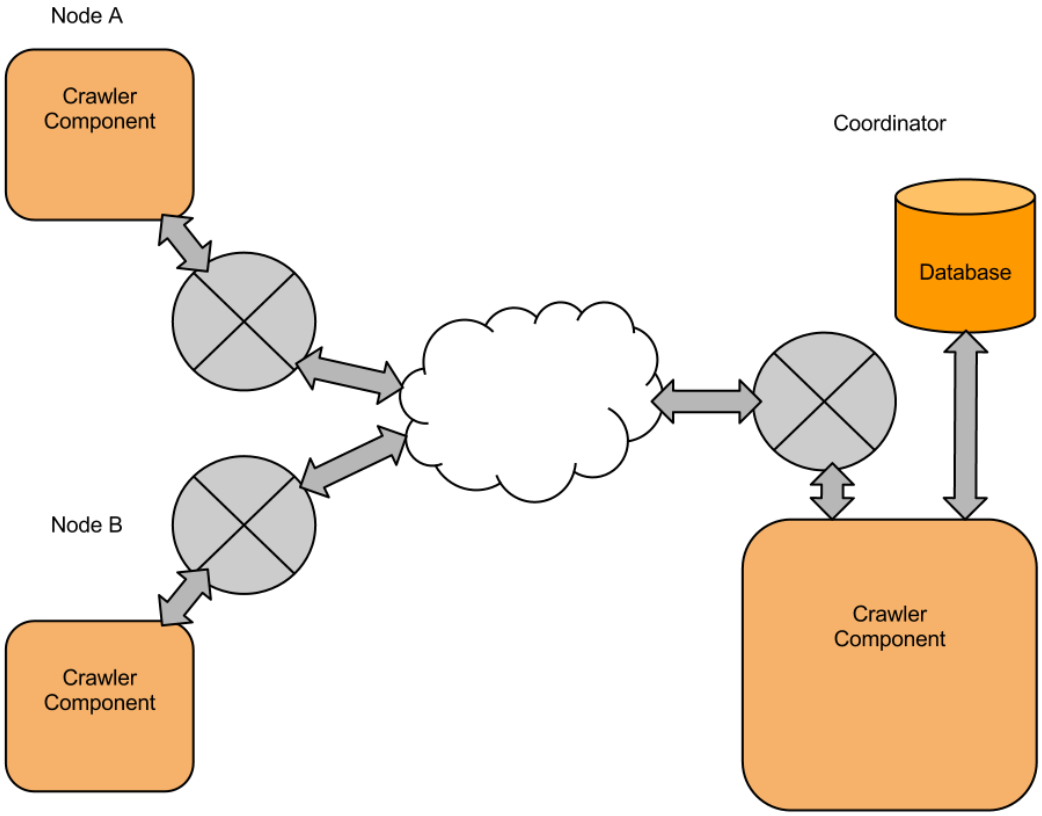
CHALMERS



Distributed Web-Crawler

Master of Science Thesis in Computer Science: Algorithms, Languages and Logic

Hans Bjerkander and Erik Karlsson

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 21/04/2013

Distributed Web-Crawler

HANS BJERKANDER
ERIK KARLSSON

Examiner: ERLAND HOLMSTRÖM

Chalmers University of Technology
Department of Computer Science & Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Cover: abstraction of the distributed workload

Deparment of Computer Science & Engineering
Göteborg, Sweden, October 26, 2013

Abstract

This thesis investigates possible improvements in distributed web-crawlers. Web-crawling is the cornerstone of search-engines and a well defined part of Internet technology. Due to the size of the Web, it is important that a web-crawler is fast and efficient, since a web-crawler should be able to find the interesting sites before they change or disappear. The thesis will focus on crawler distribution concerning modularity, fault-tolerance and group membership services. The download order of crawlers will also be covered, since this greatly influences the efficiency of a crawler. In addition to the theoretical basis of the thesis, a prototype has been constructed in Java. The prototype is efficient, modular, fault-tolerant and configurable. The result from the thesis indicates that using a membership service is a good way to distribute a crawler and conclusively, the thesis also demonstrate a way to improve the crawling order compared to a breadth-first ordering.

Sammanfatting

Denna uppsats undersöker möjligheten för förbättringar i distribuerade web-crawlers (så kallade spindlar). Web-crawlers är en fundamental del av sökmotorer och är ett väl utforskat område inom internetteknologin. Eftersom storleken på Internet är enorm är det viktigt för web-crawlers att vara effektiva och snabba, detta för att hitta relevant data innan data ändras eller försvinner. Uppsatsen fokuserar på distribuering av en web-crawler i fälten av modulärt, feltåligt och så kallade group membership tjänster. Uppsatsen fokuserar också på nerladdningsförloppet hos web-crawlers då det har en stor påverkan på effektiviteten av web-crawlern. Utöver den teoretiska delen i den här uppsatsen har en prototyp skrivits i programspråket Java. Prototypen är effektiv, modulär, feltålig och konfigurerbar. Resultatet av uppsatsen visar att användning av så kallade group membership tjänster är en bra metod för att distribuera en web-crawler. Slutligen påvisar även uppsatsen en metod för att förbättra nerladdningsordningen.

Acknowledgments

The work was conducted in collaboration with Findwise AB and we would like to express our gratitude to all Findwise employees and especially our supervisor Johan Sjöberg. We would also like to thank our examiner Erland Holmström at Chalmers.

Contents

1

Introduction

1.1 Background

Crawling the web is an essential part for feeding search engines with web-pages, enabling the engines to index the web-pages and making them searchable. Crawlers have been the favorable approach since the birth of Altavista and their web-crawler in 1995.
Since then much has happened including the entry of Google and their crawler the GoogleBot, which has changed how people regarded web-crawlers.
With the rise of cloud computing in the beginning of 2000, the need and want of distributed systems have increased[1]. Because web-crawlers handle a large set of data, the workload is ideal to distribute over a group of computers. Data considered to be crawled in the thesis are among others URL, HTML-pages and Sitemaps.
Findwise AB is a search-oriented company that realized that a distributed webcrawler would be advantageous to use instead of their regular non-distributed crawler. But since their regular crawler is part of a larger project it was more suitable to research a whole new crawler instead of improving their old crawler.
The crawler has to decide how to determine which of the resources in the cluster of clients should crawl what part of the data set. A crawler will also have to handle the issue of instability of the network and what happens if a node leaves the cluster unexpectedly.
Distribution in computer systems is a well defined part of the discipline of computer science and it has been a field of interest since the invention of the Ethernet in the 1970's.
There are many ways to achieve distribution in a network. This thesis will focus on a solution from a high level perspective. The approach of this thesis is the method of multicasting and unicasting messages and formulates that the state of the network will be the same for every member of it and no messages should be lost.

1.2 Goal & Purpose

The purpose of the thesis is to research and implement a distributed web-crawler that will divide the workload of a regular crawler evenly over a number of nodes. The number of nodes should be arbitrary and the presence of nodes should be dynamic i.e. individual nodes should be able to be change its status back and forth between absent and present in the environment at any time. The nodes should be other computer systems in the same network. This will reveal the opportunity to use the nodes previously unused data capacity with the goal of a faster and more efficient way of crawling the web. The crawler should be viewed as one entity from a outside perspective, meaning it will only have one connection for input and output. This is because the index server soley has to operate with a black-box.
Other than producing a report, the goal is to develop a distributed web-crawler, called FindDist. The software is to be configurable and effectively distributed over several nodes. It will follow the robots.txt standard and be polite, so that requests to a web-server is limited, thus not overburden the web-server.
FindDist is constructed in Java, therefore making it platform independent. The software focuses on finding links in HTML documents and is coded in such a way that it is easy to append new functions, for example a parser for a new file type.
The crawler should have an architecture sufficient for the thesis and that allows it to be easy to follow the dataflow of exchanged and extracted data of the crawled items.
It is also essential that the distributed crawling of the thesis is more efficient than an equal number of stand-alone crawlers over the same amount of nodes.

1.3 Limitations

Since the goal with the FindDist crawler is to increase the efficiency in crawling by using several computer systems in a network, it will not have resources for processing the actual data that is found by the crawling. This means that it will disregard the indexation aspect of web-crawling. This because the indexation aspect is a different scope than the actual crawling. The FindDist crawler should only try and gather more input for a index-system to handle.

2

Analysis

This section describes some of the problems and solutions of a distributed web-crawler. It is assumed that the reader has some basic understanding of what a web-crawler is and how it works.

2.1 Distribution architecture

Distributed systems are present in most modern computer systems today. Since distribution can be achieved on different levels of systems and on different parts, we need to describe some of the various ways the distribution is accomplished.
There are several different architectures of distributed systems, two of the most common architectures are examined below.

Client-server

Client-Server distribution is one of the most common architectures of distributed systems and it is often used for making a great quantity of requests to a server. In most cases, the server is static and stateless, making the interaction between client and server mostly the same for different clients. Client-Server design is elegant since it separates what would be considered client computing and server computing[2]. The most common relationship between clients and server is that the clients make requests to the server and the server fetches data for the client to use. The aim is to have a server with better and stronger hardware, so the server can handle lot of requests and do all the heavy computations, which enables the clients to be made of low end systems.
It is also designed with the presumption that client-server relations do not change and that a client will never change status. This creates a static relationship environment, meaning that it would not be very fault tolerant if the server would unseemingly crash. Taking into account the restrictions mentioned above, this approach do not seem feasible in a distributed crawler. Since it would be desirable for the server to also perform a large part of the clients task, this would mean that it would be no clear distinction between a client and a server. If the server would crash, a client could be able to assume the role of the server, which is a sought-after trait of a network.

Peer-to-Peer

Another approach is to use a peer-to-peer architecture. A peer would be a computer system of roughly the same configuration as all other peers. Peer-to-peer architecture has gained much popularity in the last couple of years, since the introduction of modern peer-to-peer systems [2]. This approach offers several benefits since a peer-to-peer architecture is designed with all ”peers” in the network being more or less on the same level of computing power. In a peer-to-peer network peers should interact between each other in a direct fashion with no intermediate manipulation. As peers can often acts as a client or server, this property makes a peer-to-peer network very robust for unseen changes in the membership of the peers. If a server would crash or otherwise not able to handle its function properly, another peer will be able to step up to the task and act as coordinator. The coordinator would act like a server and handle computations that does not concern the actual crawling *e.g.* which node that should crawl what *etc.* The difference between a server and a coordinator is that while the server is a predefined part of the network, any node should be able to act as the coordinator, thus making the membership between nodes dynamic.
This property makes it very suitable for a highly fault tolerant distributed crawler, seeing that if the coordinator node should crash or leave the network, another node could replace it. The peer-to-peers architecture property of autonomous joining and leaving [2], makes it possible for the network to achieve a dynamic membership where the efficiency of the crawler would be linear with number of nodes in the network.

2.2 Group Membership Service

Group membership service is a service in a distributed system where processes might join or leave at any time. A group membership service is often supervised by an observer that handles ”join” and ”leave” requests.
Fault tolerance is a property of high regard in the network of nodes, where the nodes should achieve autonomcy. This creates the need for a dynamic environment where the nodes can interact, detect changes and make exchanges with each other. It is of most importance that the nodes can exchange messages with statuses in the dynamic environment. A group membership service covers these needs and the service would dictate how these properties are implemented. This kind of service would have to define a concept for message passing between nodes and how discovery of nodes are done *etc.* Since the environment is dynamic, the communication should be synchronous, as it dictates a shared view of the network. Coordination is paramount, so the service will also have to implement a method for selecting a coordinator to handle a view of nodes and state of the network.

Discovery service

Since it is desirable for the network to be dynamic, it needs a discovery service to find existing groups of nodes or new nodes that are trying to join a group. If the network has a centralized server, it is common to use a protocol such as DNS for discovery of nodes. This because it stores the addresses of the nodes and a new node only has to ask the DNS server for their address. But when a distributed system is wanted, this kind of entity cannot exist[3]. This means that a server needs to implement another way of discovering new nodes with less effort. Suitable approaches of the server should be based on multicasting messages of acknowledgment[4]. There are two main approaches of this, pushing and pulling[3].
Pushing methodology is centered around the idea that nodes consistently advertises their position by multicasting it over the network in intervals. If a new node entering the network does not get the message in time, it will form a group with itself as the only member. This creates several problems as for example choosing an interval for sending messages. A too short interval would put too much weight on the network but a too long interval would create a too big a delay for a new node to find the existing ones.
The pulling methodology is basically an opposite solution that uses pulling to find the position of existing nodes. Pulling has a predefined address to which the nodes can subscribe. A new node then sends a message to the address that get multicasted to the nodes subscribing to the same address, in this way they can then respond to the new node. If a new node does not get any response, it will form a new group with itself as the only member.

Message handling

Message passing is well defined section of distributed systems and group services. Ordering is necessary for distributed systems since it indicates if there has been illegitimate use of the network. Ethernet in itself does not offer any kind of ordering structure. Ordering gives messages structure and it enables the determination of the order in which the messages has been broadcasted and in what order they should be delivered. There are several ways to handle messages propagating through the network, Défago, Schiper and Urbán in [5] gives the following definitions:

- **FIFO ordering:** Orders message in first-in/first-out fashion. If process *p1* sends a message m before it sends message m' then a correct process will deliver m before they delivers m'.
- **Casual ordering:** Uses the happened-before relation[6] denote by $\rightarrow$. Where the relation $m \rightarrow m'$ holds true no correct process will deliver message m' before they have delivered message m.
- **Source ordering:** Source ordering is a special case of FIFO ordering that take into account the messages origin, this form of ordering can only ensure total ordering if messages are broadcast from a single source.

- **Total ordering:** Guarantees that all correct processes receive all messages in the same order. If a process *p1* delivers message *m* before message *m'* then process *p2* will deliver message *m'* if and only if it has delivered message *m* before.

These are the typical definitions of message ordering in distributed systems. Source ordering will not be considered as a viable way of ordering messages since it is not applicable on the kind of network that the FindDist crawler aims to occupy.
The happened-before relation is defined by Leslie Lamport[6] as the following binary relation:

1. If a and b are events in the same process, and a comes before b, then a $\rightarrow$ b.

2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then a $\rightarrow$ b.

3. If a $\rightarrow$ b and b $\rightarrow$ c then a $\rightarrow$ c. Two distinct events a and b are said to be concurrent if a $\nrightarrow$ b and b $\nrightarrow$ a.

Both ”FIFO” and ”casual ordering” is not reliable for ordering messages, because the system has to be certain that every node has perceived that all messages have happened in the same order. As shown in [3, 5] there are several cases of this. Therefore it can be concluded that ”total ordering” is the most suitable solution in the group membership service to synchronize message in the network. ”Total ordering” is a more strict ordering than the other presented examples and it also make sure that every member of the dynamic group perceives that every message is in the same order.

Management of nodes

The number of nodes in a dynamic environment should be arbitrary, therefore the network should handle the loss of nodes without any faults. Faults that can happened when a sudden loss of a node occurs, are for example that messages may be lost or that deadlocks may appear. This can put the network in an unexpected state which it can not handle. A example of this is when Amazon.com cloud (Amazon S3) went down because of corrupt messages[7].
There are several defined methods for improving the tolerance of node loss. For example the Byzantine fault tolerance algorithms or ACID message propagation. The network also have to achieve synchronous communication and one of the simplest way to accomplish this, is using Birman and Joseph theorem of virtual synchronization. For every sent message, an acknowledgment response will be received by the sender, the response containing information indicating if the message succeeded or failed its task. Birman and Joseph's approach is not really feasible if the goal is to detect node loss though.

This because if a node crashes, it will not send a message stating that it has done so. The handling of node loss might be expensive for the network, and it has been agreed that just disregarding failed nodes and assume that they have crashed is simpler, more scalable and more efficient[3, 8].
There is only one case of node loss that needs to be accounted for and that is the loss of the coordinator node. Since the network needs to be coordinated, it can be assumed that there will be a coordinator and if the coordinator crashes there has to be a new determination/election of another node that should act as the coordinator.

2.3 Realization of the distribution

To implement the distribution as group membership service, there are several backbone frameworks which can be used for the realization of the distribution, such as CORBA, Zookeeper and JGroups.
The CORBA standard is an older defined standard for distribution between computer systems. The standard was set by the OMG group[9]. CORBA's biggest advantage is that it enables communication between systems by using different OS and different implementation languages, *e.g.* an Unix system with a client written in C should have no problem communicating with a Windows system with a client written in Java. But even though CORBA is implemented in Java, it is not fully realized and additional work has to be done. CORBA does not have a native group service available and this have to be implemented manually.
Zookeeper is a framework for Java developed by "Yahoo! Inc" [10]. Zookeeper offers group service but is dependent on a set of predefined servers to act as a coordinator-service. This makes the service robust but it is not the behavior the FindDist crawler is striving for, since FindDist aims to achieve self-reliance, to be able to start without knowledge of the nodes in the network beforehand.
JGroups is a framework developed by Bela Ban, that offers reliable multicasting[11]. The framework offers an easy way to send message between nodes and is very configurable. Using JGroups with UDP protocol for multicasting a node's presence will contribute to make the system autonomous, since it does not need any predefined data to make itself known to other nodes. JGroups also implements "total ordering" for the structuring of messages, which is needed to ensuring that every node perceives that messages are received in the same order.
Because of its advantages over the other systems considered, the JGroups framework is chosen as provider of the group membership service to the FIndDist crawler.

Using JGroups for Group Membership

Group membership is used for the purpose of equalizing every node except the coordinator. This will create a controlled environment for the different parts of the node cluster, enabling it to be certain of which of the messages are passed between the nodes. Multicasting should be used for the discovery within the group, so that all nodes will know of the coordinator's presence. Therefore, it is essential that the multicasting service is reliable. JGroups offers an easy way for selecting the coordinator, since it uses a list of all the visible nodes that is multicasted and the nodes with index '0' is the coordinator. That means that even without any calculating, there is always a coordinator. With this in mind, the algorithm for joining the group is as following:

```
Look for cluster with same clustername as me
   If not found
     Create cluster and set myself as COORDINATOR.
     Record MY STATE as the GLOBAL STATE.
   Else
     Request JOIN from COORDINATOR.
     JOIN the cluster.
     Acquire the GLOBAL STATE from COORDINATOR.
```

When a new node joins the group, the coordinator will record a new global state and the client nodes will acquire it. This will minimize the problem with sudden node loss in the group. The global state will ensure that the nodes within a group always have the same ordering. It will also guarantee the view of other nodes, this to avoid that messages get passed to the wrong node. Messages containing data on what to crawl and what already have been crawl goes through the coordinator to keep the interaction between client nodes to a minimum. All nodes have software to send data between them, but this software is only accessible while being the coordinator. If the coordinator leaves the group in any way, the new coordinator will be the next node on the list. The node that gets assigned as the new coordinator will then activate the coordinator software and there will be almost no loss of message as long there is a node with index '0' in the list.

Workload distribution

The task to crawl specific data sets is distributed in a fashion ensuring that the coordinator calculates which node should take on which task. When starting the crawling with a fresh database, the coordinator will initiate the crawling and the client nodes will wait until the coordinator assigns them a path to be crawled. When a node has finished crawling a path, it will send a message back informing that the task is done. The coordinator will store all found data in a database and when it gets a 'task done' message from a node, it will modify the database accordingly and send the data to index server.

Data is divided between the nodes by the coordinator, who will select a node based on its index in the node list in the global state. The selection is done by computing the hash of the data then modulus it by the number of nodes, thus getting the index of the node that should process the data. This gives an even distribution of work on every node.

Client(s)	Coordinator
(1)finds new link and sends the link to coordinator	
	(2)receives link (3)inserts the link in the database (4)calculates which node should handle the link (5)sends link to the client who should handle it
(6)crawls the link and extract new links (7)sends back the new links and a check message with original link	
	(8)calculates which nodes should handle the new links (9)receives check message and modifies the database accordingly (10)sends the finished page to the index server

Table 2.1: This table shows the exchange of messages between a client and the coordinator

The coordinator is the only node that communicates with the database and the index server. In case the coordinator leaves the group, the addresses to the database and the index server are stored in the global state so that the new coordinator can fully take on the role of coordinator.

2.4 Distribution of work

There are several ways for distributing work across a distributed system. The crawler will only aim to crawl URLs, sitemaps and RSS and therefore the distribution will only take into account these file-formats. Most distributed crawlers soley distribute work

over multiple threads, and so does the FindDist crawler, which aims for distribution over several different computer systems in a network to work as one. Even though the FindDist Crawler distributes over several computer systems, the systems will also distribute over multiple threads in a thread safe way.
The distributing of the URLs can be done in many ways, as assigning one node to a specific domain or subdomain. Though if this is done, it will create some native problems as for example how to calculate which node should handle which domain or how to scale with dynamic membership of nodes in the network.
Consequently, this approach is not a satisfactory solution. In addition, there is a high probability that the new links found will be located at the same domain, since the crawling is done in a breadth-first fashion. This may create starvation on nodes that are not assigned to a selected domain, since the domains may not be evenly distributed over the nodes. This might lead the crawler to a non effective use of all the resources in the network and therefore not satisfying the objective of the thesis.
Disregarding what domain or subdomain the URL belongs to while simultaneously distributing every URL as unique URL and setting aside any relation with similar URLs, will prove to be a better solution. This way, every URL needs to be distributed and calculations for finding a node to handle the URL has to be done. This creates a greater number of calculations and more transactions between nodes. Even though this process creates more work, the calculations are relative lightweight, thus it will not weigh down the system's performance. The extra network flow is also negligible since the message going back to the coordinator still has to be sent. This creates a more robust system, since if a node would suddenly die, the only data lost would be the specific URL currently being crawled. The workload will also be more evenly distributed. For example, if one domain is overrepresented in the crawling, one node may be doing most of the work irrespective of the number of nodes in the network.

2.5 Scalability of node members

The goal is a efficiently distributed workload over the number of nodes in the cluster. To achieve this, the cluster needs to attain a form of scalability. Scalability is not only about operating more processes but doing so efficiently[12]. The application of scalability can be put in use on several parts of the distribution and crawling.

List indexation

The straightforward approach to achieve scalability over the nodes joining and leaving the cluster, is to ignore which node is which. This can be accomplished with a list of the nodes and classifying them by the index of the list and ignoring what index is pointing to which physical node, only noting how many nodes there are present. Consequently, there will be no need to make corrections if a node leaves when distributing work, except if the node that is about to receive work is the last on the list. This creates a dynamic relation between the nodes and it will not burden the computations with what nodes are

in the cluster. This would work best if all messages converged at the coordinator, since else the membership list may differ between the nodes.

2.6 Filetypes

This section contains descriptions of different file types that FindDist can parse and it focuses on how links are represented in each file-type. The different file types presented was selected due to their high usage on the Web and that they usually contains a high amount of links.

HTML

Hyper Text Markup Language (HTML) is a major building block of the Web and it is the major markup language for web-pages. One of the more important functions of a web-browser is to be able to parse and translate a HTML-document into human readable format. HTML is composed of elements of tags enclosed in angle brackets(⟨ ⟩). A link in a HTML document consist of an anchor element with the attribute href and the value of href decides what the link will point to. For a webmaster, it can sometimes be necessary to disallow crawlers from following links and indexing certain pages. By adding the attribute ”rel” with the value ”nofollow” to the anchor element, the crawler will be forbidden to follow a specific link. A more general way is to define a behavior, that will apply for the whole document. It can be done by adding a meta tag with attribute ”name” with value ”robots” and a content attribute with the value of a combination of ”noindex”,”index”,”nofollow” and ”follow”. ”nofollow” and ”follow” tells the visiting crawlers if they are allowed to follow links in the document or not. ”index” and ”noindex”, allows or disallows indexation of the page.

```
<html>
  <head>
    <meta name="robots" content="index, nofollow">
  </head>
  <body>
    ..
    <a href="http://www.findwise.com">Normal link<a>
    <a href="htpp://www.findwise.com" rel="nofollow">nofollow link<a>
    ..
  </body>
</html>
```

Figure 2.1: HTML example

Sitemaps

Sitemaps is an easy way to inform web-crawlers and search engines about the content of a domain. They are usually linked in robots.txt or in a link to it sent to search engines. A Sitemap is a XML-document that list URLs and possible meta-data for each URL. The meta-data contains information about when the page was last updated, how often it is changed and how important it is, *i.e* its priority to be indexed relative to the other URLs on the domain.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ...
  <url>
    <loc>http://www.aftonbladet.se/debatt/article10250770.ab</loc>
    <lastmod>2011-03-08T06:53:23+01:00</lastmod>
    <changefreq>daily</changefreq>
  </url>
  ...
</urlset>
```

Figure 2.2: Sitemap example

A Sitemap is not allowed to contain more than 50,000 URLs and be larger then 10MB and it can be compressed in the gzip format to be able to fit more data. If there is a need for a larger Sitemap, it can be done by a Sitemap index, which can list several Sitemaps in a file. The Sitemaps index syntax is similar to Sitemaps but using Sitemap elements instead of url elements[13].

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ...
  <sitemap>
    <loc>http://www.aftonbladet.se/svn/sitemap/2001/12.xml.gz</loc>
    <lastmod>2012-05-04T16:11:28+02:00</lastmod>
  </sitemap>
  ...
</sitemapindex>
```

Figure 2.3: Sitemap index example

RSS

RDF Site Summary or Really Simple Syndication is a easy way to publish a web-feed of frequently updated pages and is often used by blogs and news headlines. The RSS file is a XML-document describing a channel and its items. It can be seen as a Sitemap specialized on a small part of the domain with a high frequency of update.

```
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <link>http://www.dn.se/m/rss/senaste-nytt</link>
    <title><DN.se - Nyheter - Senaste nytt></title>
    <description><DN.se - Nyheter - Senaste nytt></description>
    <ex:version>1.0</ex:version>
    ...
    <item>
      <title><Sommaren närmar sig med stormsteg></title>
      <link>http://www.dn.se/nyheter/sverige/sommaren-narmar-sig-med-stormsteg</link>
      <pubDate>Fri, 18 May 2012 13:18:00 GMT</pubDate>
    </item>
    ...
  </channel>
</rss>
```

Figure 2.4: RSS example

2.7 Ordering

A big part of web-crawling is deciding in which order the newly found links should be downloaded. Due to the size of the Web and its volatile nature, there is a high risk that the crawler will not be able to find all pages before they are changed or removed[14]. New pages can be created and disappearing faster then a crawler can find and process them, if a ineffective ordering is chosen [15] [16].
Depending on what data is going to be crawled, several different strategies can be applied when deciding the crawling order. Examined below are some of the major ones.

Focused crawling

The explore order of the links is decided by an algorithm, which is based on the metric of the links. A well known ranking algorithm is pageRank, invented by the founders of Google, Sergey Brin and Lawrence Page. Each page is given a pageRank value depending on the pageRank of other pages pointing to it and the number of outbounds link it have [17]. This way a site with many inbound links usually gets a high pageRank. The

pageRank is defined by the equation 2.1.

$$PR(A) = (1-d) + d * \sum_{l \in B_A} \frac{PR(l)}{L(l)} \tag{2.1}$$

PR is the pageRank value, d is a value for scaling the values, B_A is a set containing all inbound links to the page A and L is the number of outbound links from l. The pageRank algorithm is fairly easy to compute but with the size of the Web, it could led to a hefty constraints on the web-crawler. Additionally, to be able to compute the final value of each page, most of the Web must be known, which given a web-crawler is usually not the case. Therefore the page-rank algorithm is mostly used for re-crawling data. Another way can be to prioritize the links by sorting them on how many times they have been found in downloaded pages. When crawling for a specific type of data, such as audio-files, it would be simpler to give links priorities depending on the number of times it is found on pages with audio-files.

Breadth-first crawling

Breadth-first crawling is the simplest and most straightforward way of crawling. Starting with a small set of links and exploring links from them as they come, thus following the links in a ”breadth-first” fashion [18]. According to [19] bread-first ordering yields high pageRank values for the pages found during the early stages of the crawl. As the crawl continues the page qualities deteriorates. This could be due to pages with high quality has many pages linking to them and therefore is easily found fast but smaller and important sites is harder to find.

Random crawling

Random crawling is as indicated, crawling carried out in a random fashion. There has been some usage of random crawl orders with good result Bennouas and de Montgolfier in [20] shows that it is possible for a random order to achieve just as good results as breadth-first ordering. If the links to be crawled are spread on many different web-servers, the selected links will also be spread. This makes it possible to minimize the need to often connect to the same web-servers.

2.8 Robot Exclusion Protocol

The Robot Exclusion Protocol or robots.txt is a standard method since 1994. It is a rather simple way of specifying to web-crawlers what they are allowed to do on a domain. It is important to remember that the standard is voluntary and only well-behaved crawlers will follow it. The robots.txt file, placed in the root-level, provides different

access policies for different crawlers with the user-agent term. A user-agent with name * denotes all crawlers. A number of directives can be followed after a user-agent, the most common ones are Allowed and Disallowed, which denotes what the crawler are allowed and not allowed to access on the domain[21]. A unofficial rule is the Crawl-delay, which specifies the allowed frequency of requests to the domain [22]. Another rule is the Sitemap which specifies where a Sitemap for the domain can be found. The directive is not associated with a user-agent and is usually placed in the beginning of the file [13]. Lines starting with # are treated as a comment.

```
# 80legs
User-agent: 008
Disallow: /

# MSNBot
User-Agent: msnbot
Crawl-Delay: 20

User-Agent: *
Disallow: /*.json
```

Figure 2.5: robots.txt example

In the example above, the robots.txt specifies that the crawler with name 008 is not allowed to access anything on the domain, the crawler with name msnbot must limit its visits to the domain to one in every 20 seconds and that no crawler are allowed to access files ending with .json.

2.9 Etiquette

One of the more important part of web-crawling is to not overburden the web-servers which are crawled. By sending too many request in a short interval, the server might interpret the crawlers requests as a denial-of-service attack. Usually web-crawlers decides the delay themself but some domains specifies a minimum delay between requests in their robots.txt. For a non-distributed web-crawler, it is not too complicated to be polite. For each sent request, save the time it happened and make sure that next request to same web-server happens after the delay. When a crawler is distributed, it could become a problem to share the policy over several nodes with ”the last request time”, since it would not be very effective. To keep using a local policy in a distribution system, the URLs have to be distributed only on its domain part, thus making sure that only one node contacts a specific web-server [14]. A downside to this system would be risk of uneven distribution of the data, for example one node could get only small domains with not much to crawl while another node could get large domains. Another method could to

be to use local policies but with multiplying each delay with the number of nodes. Thus over time being polite, but with a good and even distribution of the URLs. Instead of using static delays, the delay could be decided on the servers capabilities. This could be done by measuring the time it took to download a page and then using a delay, for instance 10 times that value. This would limit the number of request to servers according to its status and only burden the server when it can handle it [14].

2.10 Database

A record of each link crawled needs to be kept somewhere to make sure that the crawler only downloads the same page once. Since FindDist will be able to crawl millions of links, is it practical to save this information in a database. The data to be saved is only an URL and a value indicating if it is downloaded or not, so there is no need to use a graph database, since it is specialized on data with many relationships[23]. The data must be queryable to ensure that the links which has not been fetched yet will be found. Key value stores has no native support for this but it could be implemented in the application[23]. But with increased risk for bugs and the slowing down of the system, it would be better to use a document or a family column store. The majority of all operations on the database would be of the insert type. Therefore it is essential that the database should be able to handle small and fast writes. Due to the fact that the coordinator node is the only node connected to the database, there is no need for concurrency. The worst that could happened is that a fetched link is resaved in the databases as not fetched. The consequence would be that the node would be refetched in the future. This is very rare and the risk is worth taking for a write big speed boost. Since the web-crawler is implemented in Java, the database should have a good Java API or good java bindings. it is also important that the database should be developer friendly in that sense that it should be easy to use and setup. The database chosen was the document store database MongoDB. Due to that its concurrency strategy is last-update wins, it has fast writes [23]. It is also easy to query, has a great Java API and is easily scalable horizontally [24].

2.11 Indexing server

This section contains information on how an indexing server works. This is outside the scope of our thesis but is included to give a better understanding of what the output from the web-crawler is used for.

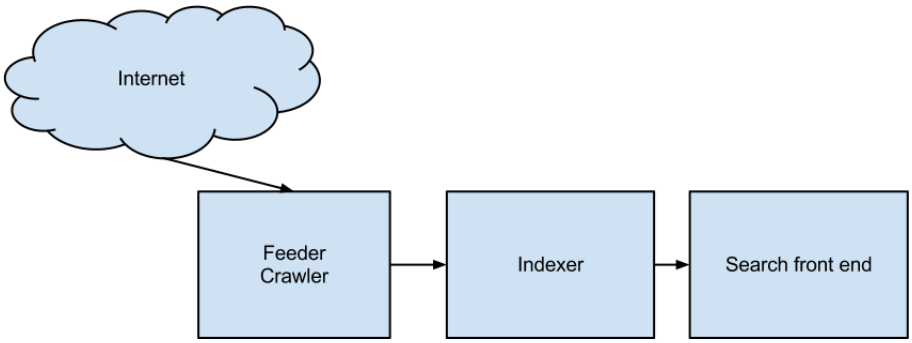


Figure 2.6: A simple abstraction of a typical search engine

A search engine consists of three major parts: a feeder, an indexer and a search front end. The feeder is the mechanics that provides the indexer with the content and most often, the feeder is a web-crawler. The search front end is the user interface of the search engine and compares user queries against indexed content provided by the index server. The indexers purpose is to generate a so called index for each document, which then can be used to find relevant documents easier. This process can be seen as making unstructured data into structured and categorized data. The index depends on the implementation of the indexing server, but the purpose is to optimize the data for fast and accurate look-ups. Without indexing, the search engine would need to scan all data for the answer to the query, which would require substantial processing power and time.
A computer can not understand the structure of a natural language and therefore not automatically understand what part is words and sentences in a text. A text must therefore be split into words before being indexed. Such a program is usually called tokenizer, lexer or parser. The tokenizer can also discover entities, such as telephone numbers, email address and URLs, which is not regular words in a natural language. There is also the possibility to save additional data for each found token, such as sentence number, sentence position and lexical category(noun or verb). Some documents contains only data for the purpose of describing the document to a search engine, this type of data is called meta-data.
With HTML it is possible with the meta-tag to provide meta-data. The tags are useful to provide information about the site, but can easily be abused to get in the top findings of many search results, by so called keyword spamming. Though search engines have become more intelligent over the years and are now able to handle this. In an internal system, meta-tags are very useful due to the lack of interest to abuse it.
In order to give a good result of a query, the documents needs to be ranked depending on how relevant their terms are to the search queries terms. There are many different ways of doing this ranking, one of the more common methods is a combination of Term Frequency(TF) and Inverse Document Frequency(IDF). TF is the number of times a term occurs in a document. In itself, it is bad to use as ranking, words that occurs often in a text will get a heavy weight but the word itself is not taking into account. For example, in the query ”the blue horse”, ”the” and ”blue” will probably get a heavy

weight compared to ”horse”, due to their higher frequency in the English language. This could make the result skewed with many results lacking the most relevant word, ”horse”. IDF is based on counting the number of documents in the collection containing the term in question. A term which occurs in many documents is not a good discriminator and should be given less weight to a term found in less documents. There's many different versions of IDF but the most common form is in equation 2.2.

$$IDF(T_i) = log_2 \frac{N}{n_i} \tag{2.2}$$

Where N is the number of documents in the collection and the term T_i occurs in n_i documents. The weights are computed for each document and term pair, by taking the product of the TF and the IDF [25].

$$W_{t,d} = TF_{t,d} * IDF_t \tag{2.3}$$

Where W is the weight, t the term and d the document, the weights can then be used to calculate the relevance of documents for a specific query.
A good model is the vector space model. Each document represents a vector, where each dimension correspond to a different term and the document vector's value is the weight of that term. Create a vector Q representing the query, in which the dimensions corresponds to the same terms as in the document vectors, with the weight of that term, using the search query as document. Then the result with the smallest degree between itself and Q is the most relevant document [26]. The simplest way is computing the cosine with the equation 2.4:

$$sim(d_j,q) = \frac{\sum_{i=1}^{t} w_{i,j} * w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} * \sqrt{\sum_{i=1}^{t} w_{i,q}^2}} \tag{2.4}$$

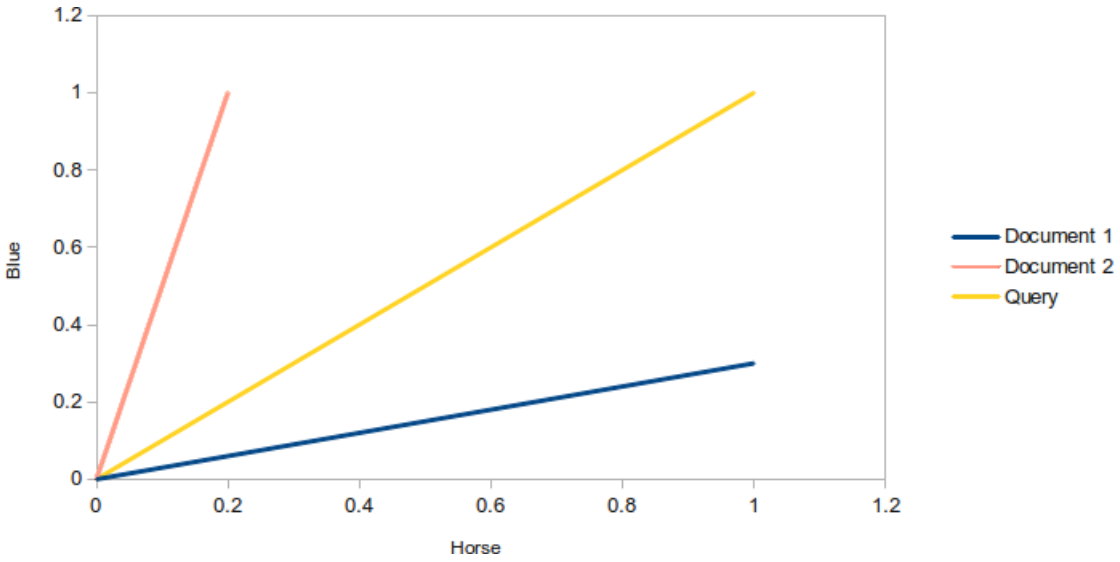


Figure 2.7: Example with two documents and a query for the terms blue and horse, where the document 1 is most similar to the query.

3

System Overview

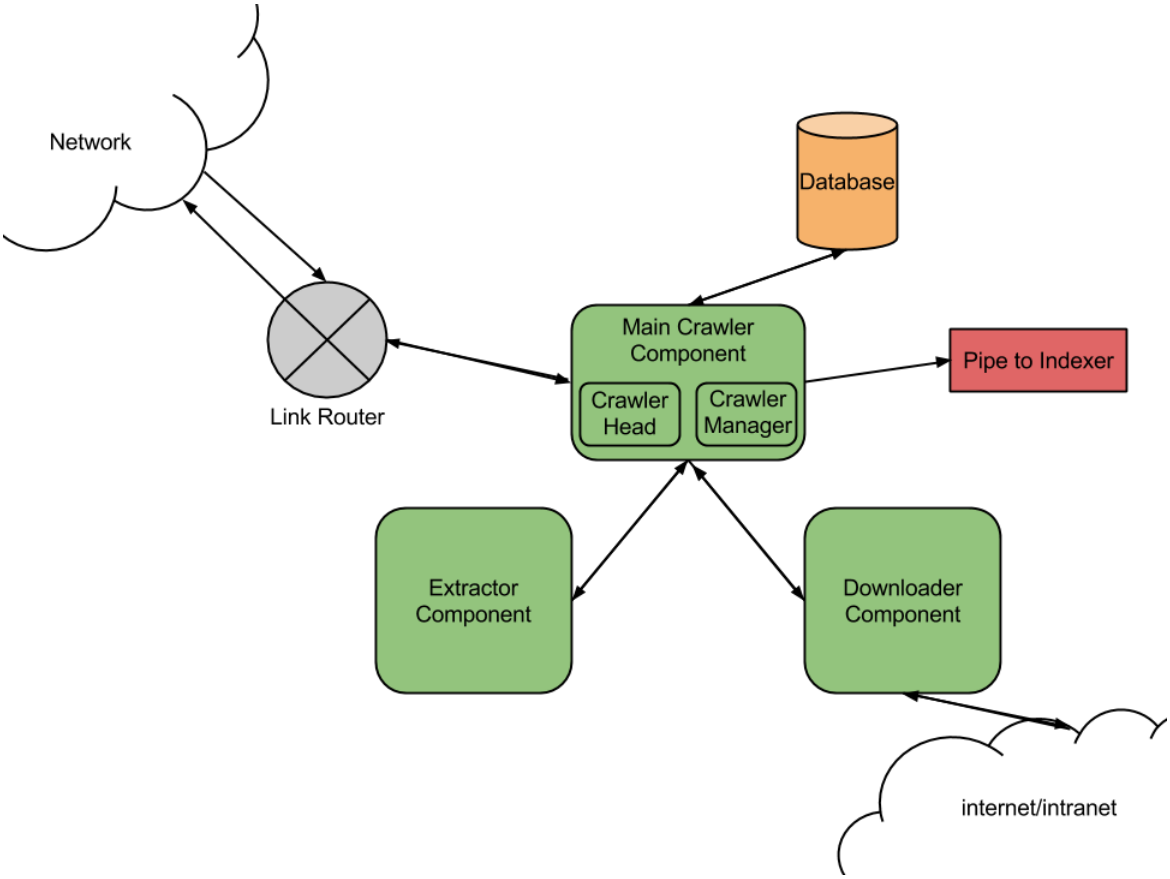


Figure 3.1: Abstract overview of FindDist

This section gives an overview of FindDist, an abstract view is shown in figure 3. The figure is simplified and its arrows represent the flow of data, not any dependencies. To avoid too much details, the overview is composed of several different components with similar functionality. The system is started by a call to the Crawler Manager which initiates the Crawler Head, which then starts the rest of the system. The system then

continues till the stop function in Crawler Manager is called. The Downloader requests a link from the Download Queue, then downloads the associated page and adds the page to the page queue via a callback to the Crawler Head. The Extractor requests a page from the page queue, extracts links from the page and then sends them to the link router. If the node is not the coordinator, the page with the found links are sent to the coordinator for processing. The Link Router checks which found links are unique and saves them in the database. For each newly found unique link, the owner is computed and sent to be downloaded.

Crawler Manager

The Crawler Manager together with the Crawler Head is the Main crawler component in the figure 3. This since both of them are rather small and handle the start and stop of the application. The Crawler Manager is responsible for handling the communication with the crawler. It can initiate the crawler and its components and starts and stops the application. The manager functions as an interface to the crawler and makes it easy to append a GUI.

Crawler Head

The Crawler Head is the component of the crawler that keeps everything in the crawler connected. It implements several interfaces and can therefore be used to callback from other components in the crawler. The Crawler Head can be seen as a data hub, where the data is received and directed to the right component. It also contains the page queue, where the downloaded pages are stored before being parsed for links.

Downloader

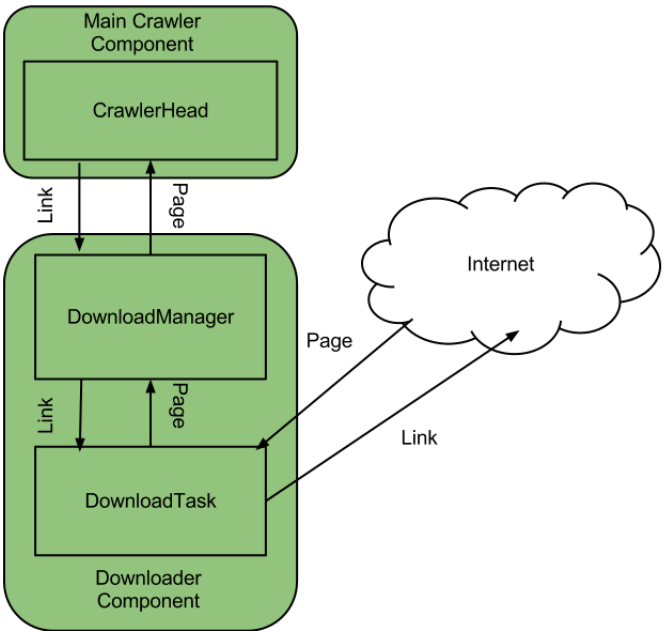


Figure 3.2: Downloader

The Downloader is the component of the crawler responsible for handling the downloading of web-pages in a polite manner. It is composed of several different classes, for a simpler overview it is combined in two separate parts: a manager and a download task. The manager fetches links from the download queue, via a callback to the Crawler Head, and creates tasks, that downloads the page pointed to by the link. Each task is run on a separate thread and there is a limit to the number of threads that the download manager is allowed to run at the same time. The number is configurable and makes the crawler effective in scaling vertical on both low and high end systems. Before a task is created, the links web-server is checked to ensure that not too many threads are downloading from the same web-server. If there are more than half of the maximum number of threads downloading from the same web-server, the fetched link is put back and a new one is fetched. The tasks consists of downloading a page and if it succeeds, it adds it to the page queue through a callback to the Crawl Head. If the download of the page fails, the task first checks how many times it has failed before. If the error count is more than a configurable number, the task drops the link and removes it from the database. If the error count is less than the configurable number, the link is added back to the download queue and will eventually be downloaded again. The downloading is done with help of Apache httpclient, an efficient and feature rich client-side http-package [27]. Before a download starts, it has to first check if it is polite to start the download. It is polite to begin the download when a specified time has passed since the web-server associated with the link, was last called. The polite time is specified either by the web-servers roobots.txt or by a configurable default value. The default value should not be to low,

else it risks that the crawler will be mistaken for a denial-of-service attack.

Download Queue

The download queue is a ordinary First-In-First-Out queue with some modifications. To ease the burden on the underlying system memory, the queue has a fixed maximum size. When the download queue reaches a quarter of its maximum size, the queue asks the database for new undownloaded links, so the queue becomes full. Due to the fact that the database keeps its content sorted by the contents hash-value, the fetched links from the database becomes almost randomly chosen. If the database contains many links from different domains and web-servers, the new links will likely have the same spread as in the database. When the links are downloaded, they will not create many waiting periods due to politeness constraints. After found links have passed the validation, they are sent to the download queue. First they are checked against the database to verify if they have been found before, those who are unique are saved in the database and marked as not downloaded. Then all the unique links which the queue has space for, are added to the queue while the rest are dropped. With the use of this type of queue, the download ordering becomes a mixture of breadth-first and random ordering. Breadth first for the links that are added on the end of the queue, but sometimes the queue will be filled with randomly chosen links from the database. The queue size is an indication of how much FindDist is breadth-first and how random it is. A queue with unlimited size will be pure breadth-first, a queue with size 1 will be pure random. As shown in the section 4 this kind of ordering yields a good result compared to a pure breadth-first ordering.

Extractor

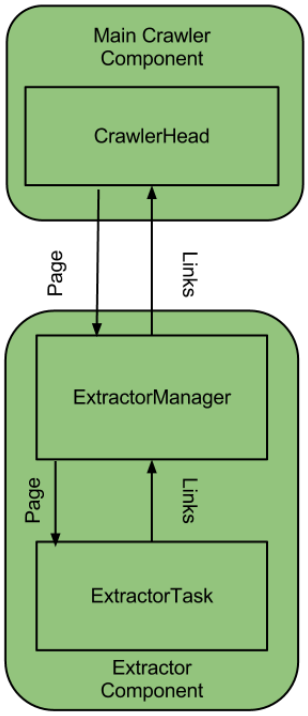


Figure 3.3: Extractor

The extractor is responsible for the extraction of links from downloaded pages. It is structured as the Downloader component, with essentially a manager and tasks running on threads. The manager fetches pages from the page queue and creates a task, responsible for extracting links from the page. As with the Downloaders manager, the extractor manager has a maximum number of allowed threads running at the same time, to be able effectively scale the crawler. The tasks is to parse a page, collect all URLs from it and send each valid link to the LinkRouter. The validation is done by first making the found URL absolute, *e.g.* to convert `/index.html` to `http://www.findwise.com/index.html`. The second step is to check if the URL follows the configurable rules, such as a maximum link depth and that the URL has a allowed protocol. The last step is to compare the URL against the associated web-servers robots.txt, if there are any. If the page is discovered to be corrupt during the process, the page is dropped. The page will eventually be redownloaded due to that its source link is still marked as undownloaded in the database. When an extraction is done, all valid links and the page are sent to the LinkRouter. In the present situation, FindDist contains adapters that can parse file types of HTML, Sitemaps, Sitemap index and RSS. These file-types is described in more detail in section 2.6.

LinkRouter

The Linkrouters main function is to implement the distribution part of the crawler. As the distribution of the crawler covers several areas, the LinkRouter also have several functions. The tasks that the LinkRouter execute are: delegation of links, receiving and transmitting messages, handling links back to the crawler. The LinkRouter needs an interface to send the links it receives, it also implements an interface that a receiver part uses *i.e.* the actual receiving is done by a different part (called MessageReceiver) that parses the message so it can determine what kind of message it is. The JGroups framework is used for communication calls and message propagation.
When the coordinator gets or finds a new link, the link is sent to the LinkRouter, so it can calculate which node that should handle it. When a node has finished crawling a link, the LinkRouter will send it back to the coordinator. The coordinators MessageReciver will parse the message as a message containing a crawled link and the links found on that page. The Linkrouter will then send the links to the interface RouterHandler, which the crawler implements so it can handle these new found links accordingly. If a node is a coordinator, it can also initiate its LinkRouter to send a kill message to one or more nodes in the network ordering them to shut down. The LinkRouter will also acquire the state of the network or set depending if the node is coordinator or not. If the MessageReceiver receives a message that is not of a specified type, the message is simply discarded.
The messages propagating through the network has several different uses. The distribution service of the crawler uses five different types of messages for invoking different aspects of the nodes. The types of messages that the crawler has defined are:

- **CRAWLLINKREQUEST** - A message containing a link that is sent from the coordinator to a node. The message dictates the node that it should crawl and the link that is contained in the message.
- **CRAWLLINKRESULT** - A message containing a page that had has its links extracted, that is sent from a node to the coordinator. The message informs the coordinator that the node has finished crawling a link and sends the new found links back to the coordinator so they can be delegated.
- **QUIT** - Sent from a node to the coordinator when a node requests to leave the network.
- **RESPONSE** - Response from the coordinator to a node that has requested to leave the network.
- **SHUTDOWN** - Sent to a node from the coordinator, saying that the node should shutdown.

Database Interface

The Database Interface handles the connection to a database and with the help of different classes, a specific database type can be chosen. The data that will be saved in the database will only be links. From each link a URL, a value representing if the link has been downloaded and some meta-data will be stored. As described in the Download Queue, the database sorts its content after the contents hash-value.

4

Benchmarking

As the system is intended to be able to replace an existing crawler, there is a need to know how effective FindDist is. Two comprehensive benchmarks was done. In order not to burden the system to much, FindDist was set to use a low end configuration. Which meant that the crawler was using 8 download threads and 4 extractor threads which made the average CPU load to be under 10%. The first benchmarks tested how effective FindDists distribution is. The test was done with 1,2 and 4 computers of the computer type 1 in table 4.1. The second benchmark tested how how effective FindDist ordering is, with a compression to how the FindDist perform with its download order and with a pure breadth-first ordering.

All crawling started from `http://www.findwise.com` and was run on computers described in table 4.1. The database used was located close by to the other computers, so latency was mostly around 1ms and did not affect the result.

Name	Computer 1	Computer 2
CPU Family	Intel Pentium B960 Dual Core	Intel Core Duo E8400
CPU freq.	2.2GHz	3GHz
Memory	4GB	4GB
OS 64bit	Ubuntu 12.04	Red Hat 5.6

Table 4.1: Different computers used in the benchmarks

A problem with running benchmark against the Web is that the Web is not static and changes over time. Even if the web-pages are the same, the time to access them will differ from time to time. A solution could be to take a small copy of the Web and perform the test against the copy. But that would take too much time and resources, as many different local web-servers needs to be created. FindDist is made for the purpose of crawling the Web, with its advantages and disadvantages, so benchmarking a static copy would remove some parts of the test, such as delays.

4.1 Benchmark with multiple servers

An important reason for doing this benchmark is to get numbers on how effective FindDist is using its distribution. A distributed node will never be able to work at the same speed as a non distributed node, due to delays in the network and extra computation time needed for knowing how to share the workload. All delays are relatively small but will add up and slow down the nodes. In this benchmark, it is examined how FindDist perform with 1,2,4 and 8 nodes and how well it scales.

	1 Node	2 Nodes	4 Nodes	8 Nodes
Found links	989426	1054445	1610383	3314431
Downloaded links	23985	34162	34911	57813
Found links/Node	989426	527222.5	402595.75	414303.875

Table 4.2: Benchmark of multiple nodes on FindDist

The data from the benchmark is shown in the table 4.2 and demonstrates how many links that was found, downloaded and in average how many links that each node found. One important observation is that the small difference between how many links were found when using one and two nodes. Two nodes finds only a small amount of more links. It could be explained by that by using only one node, the network part of FindDist is not used and therefore lot of delays are removed. Looking at the number of links found, it can be seen that the number of links are increased in a linear fashion and can be described with the equation 4.1. The equation is shown with the the data from table 4.2 in the graph 4.1.

$$\begin{aligned} \alpha + 0.44\alpha * nrClientNodes \\ \alpha = 792787 \end{aligned} \tag{4.1}$$

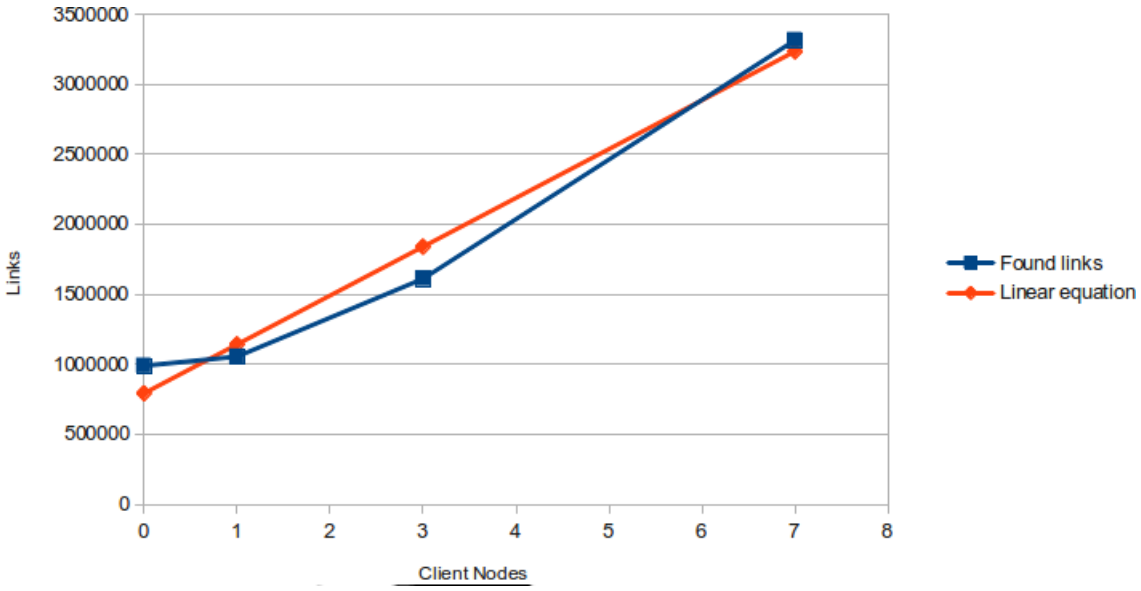


Figure 4.1: Graph with equation 4.1 and the data from table 4.2. The X-axis represent the number of client nodes and the Y-axis the number of found links.

An interesting fact is that a node in a distribution only find around 44% of the amount of links that a single node without a distribution finds. This is due to that without any distribution, a lot of the calculations are not needed and are therefore not done.

4.2 Ordering benchmark

As described in section 3, FindDist uses a modified breadth-first ordering when deciding which links to download. Breadth-first ordering is simple to implement and does not use much of the systems resources. FindDist uses a modified version which focuses on spreading out its request to several different web-servers.
In this benchmark, the performance of the crawler is implemented in a pure breadth-first fashion.
The benchmark was done on the computer 1, in table 4.1 and with FindDist ordering with a maximum queue of size 20. As is was most interesting to know how many domains FindDist finds and how much the download task has to wait due to etiquette, the server was used with a low-end configuration. This means that only 12 download threads and 8 extractor threads was used. The most interesting data is how much time was spent waiting to download a link.

	FindDist Ordering	Pure Ordering
Time	22:46:45	22:15:50
Found links	2 446 507	1 331 280
Different domains	32 032	14 785
Downloaded links	42119	30910
Wait time	32998s	26580s
Wait time per downloaded link	0.78s	0.86s

Table 4.3: Benchmark of orderings on FindDist

The resulting values from the test can be found in the table 4.3. The table shows how long the crawl lasted and how many different domains was found, which is a good value to see how breadth the crawl was. The wait-time is the sum of the time each thread spent waiting to access a web-server. The final and most important result in the table, is the wait-time per downloaded link and it shows how much time was spent on waiting before starting the downloading of a link. In average, FindDist with pure ordering had to spend 10% more time waiting before starting to download a link then FindDist with its original ordering.

5

Results

This thesis is concerned with the performance of a distributed web-crawler, due to the scope of the thesis, only some specific problems were targeted, for example how to efficiently distribute the data in an evenly fashion and minimizing the delay between downloads due to politeness constraints. The proposed improvements and features by this thesis will show how to increase the effectiveness of a distributed web-crawler. Two tests were also done to justify the improvements see chapter 4.

The first test shows the effectiveness of the distribution of the system and how much a new node contributes to the whole system. The benchmark showed that the number of found links increases in a linear fashion as the number of nodes increases. From the data a linear equation was extracted, the equation itself is not that interesting but with the equation, it is possible to get a number on how much more performance is gained by adding a new node. From the equation we get that a new node adds around 44% more links than what a single node finds.

The second test examined how the download order might be improved. The goal was to reduce the time needed to enforce politeness policies given by different web-servers without adding too much extra computations to the web-crawler. FindDist uses a variant of the breadth-first ordering with some traits from random ordering. The benchmark compared in average how much waiting time was spent for each downloaded link. In average FindDist with pure breadth-first ordering had to spend 10% more time waiting before starting to download a link then FindDist with its original ordering. When crawling through millions of links, this yields a really good result, decreasing the crawling time significantly.
The thesis has also shown and described how to efficiently use a group membership service to distribute the workload of a web crawler. The prototype FindDist is very fault-tolerant and it can collectively handle crashes of nodes in a controlled way. When the coordinator node crashes, another node will take over the responsibility. If the contact between nodes is lost, the crawler will split into several parts for each group of nodes which still have contact. When the contact is restored, the group will be reform and continue as before.

6

Conclusion and further work

There are several ways to implement an efficient distributed web-crawler, this thesis demonstrates how to overcome several obstacles and shows some improvements that can be applied. A prototype was made that is fast, has a good distribution and it proves that with a dynamic distribution it is possible to make a robust and fault-tolerant system.

A problem with web-crawling is to follow etiquette rules when downloading pages. This thesis has shown how to implement a simple download order that reduces the waiting period and yields good results. A benchmark was made that compared how well FindDist download order performed with a breadth-first ordering. The test concluded that the breadth-first ordering had to wait 10% more than the FindDists combination of breadth-first and random ordering. When crawling through millions of links this result would save lot of time. The value of 10% should not be taken as a absolute value but more as a pointer that FindDist reduces the waiting time compared to breadth-first ordering. To get a more precise value more tests have to be performed. Still the FindDist download order creates a lot of waiting time. A different solution could be to transverse all links to be download and pick the first one with a web-server that could be access directly without any waiting time. This would probably work well when crawling with a few download threads, since it would not be too long before a good link is found. With many download threads, probably more time would be used to find a good link than just waiting, specially since most of the links is stored externally on a database.

The thesis proves that for each new node added to the FindDist crawler, around 44% more links can be found compared to what a single node finds. In a perfect case the number would be 100% but lot of computation power is lost due to the transfer of data between the coordinator and the nodes. A possible improvement to this could be to change the number of links each node is allowed to handle at a time, as of now it is a static number decided by the coordinator. A better result would probably be given if the node would decide the number of links itself, depending on its hardware and network connection. Another possible refinement could be that each node ask for data to crawl instead as of now it being decided by the coordinator. If all the nodes were given the ability to access the database the crawling could be more effective, but then an interface

for the database must be created to handle all the request for the nodes. This would have the possibility to create concurrency problems, which today FindDist does not need to handle.

The distribution of FindDist is robust and fault-tolerant, much do to the membership service. It was also found that if the coordinator node would to suddenly disappear or crash, the crawler would assign a new coordinator without fault and minimum data loss. The distribution creates some overhead that probably could be reduced, which probably would speed up the crawling.

The best way to continue improving the web-crawler would to do more test, to be able to pinpoint a good number of the size of the download queue. The authors of this thesis recommend to continue to work with JGroups, since it offers options for fault tolerance and modular design.

To conclude, there are ways to improve the state of a distributed web crawler. As shown in this thesis, one way is with a robust membership service and with a new download order.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and et al., "Above the clouds: A berkeley view of cloud computing," *EECS Department University of California Berkeley Tech Rep UCBEECS200928*, no. UCB/EECS-2009-28, pp. 1–23, 2009.

[2] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2001.

[3] J. Sjöberg and S. Svensson, "Distributed document processing: Search index optimization by data preprocessing and workload distribution," 2009.

[4] R. Lass, D. Nguyen, D. Millar, W. Regli, J. Macker, and R. Adamson, "An evaluation of serverless group chat," in *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, pp. 1639 –1644, nov. 2011.

[5] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[7] The Amazon S3 Team, "Amazon s3 availability event: July 20, 2008," June 2012.

[8] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid information services for distributed resource sharing," in *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pp. 181–194, IEEE, 2001.

[9] Object Management Group Inc., "Documents associated with corba," June 2012.

[10] The Apache Software Foundation., "Apache zookeeper," June 2012.

[11] B. Ban, "Jgroups - the jgroups project," May 2012.

[12] P. Jogalekar and M. Woodside, "Evaluating the scalability of distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, pp. 589–603, 2000.

[13] U. Schonfeld and N. Shivakumar, "Sitemaps: above and beyond the crawl of duty," in *Proceedings of the 18th international conference on World wide web*, WWW '09, (New York, NY, USA), pp. 991–1000, ACM, 2009.

[14] C. Olston and M. Najork, "Web crawling," *Foundations and Trends in Information Retrieval*, vol. 4, no. 3, pp. 175–246, 2010.

[15] A. Ntoulas, J. Cho, and C. Olston, "What's new on the web?: the evolution of the web from a search engine perspective," in *Proceedings of the 13th international conference on World Wide Web*, WWW '04, (New York, NY, USA), pp. 1–12, ACM, 2004.

[16] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins, "The discoverability of the web," in *Proceedings of the 16th international conference on World Wide Web*, WWW '07, (New York, NY, USA), pp. 421–430, ACM, 2007.

[17] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, pp. 107–117, Apr. 1998.

[18] V. Shkapenyuk and T. Suel, "Design and implementation of a high-performance distributed web crawler," in *In Proc. of the Int. Conf. on Data Engineering*, pp. 357–368, 2002.

[19] M. Najork and J. L. Wiener, "Breadth-first crawling yields high-quality pages," in *Proceedings of the 10th international conference on World Wide Web*, WWW '01, (New York, NY, USA), pp. 114–118, ACM, 2001.

[20] T. Bennouas and F. de Montgolfier, "Random web crawls," in *WWW*, pp. 451–460, 2007.

[21] C. M. Drott, "Indexing aids at corporate websites: the use of robots.txt and meta tags," *Inf. Process. Manage.*, vol. 38, pp. 209–219, Mar. 2002.

[22] Y. Sun, I. G. Councill, and C. L. Giles, "Botseer: An automated information system for analyzing web robots."

[23] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," *Cloud and Service Computing, International Conference on*, vol. 0, pp. 336–341, 2011.

[24] 10gen, Inc, "Mongodb." `http://www.mongodb.org/`, May 2012.

[25] S. Robertson, "Understanding inverse document frequency: On theoretical arguments for idf," in *Journal of Documentation*, vol. 60, pp. 503–520, 2004.

[26] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[27] The Apache Software Foundation, "Httpcomponents httpclient overview." `http://hc.apache.org/httpcomponents-client-ga/`, May 2012.