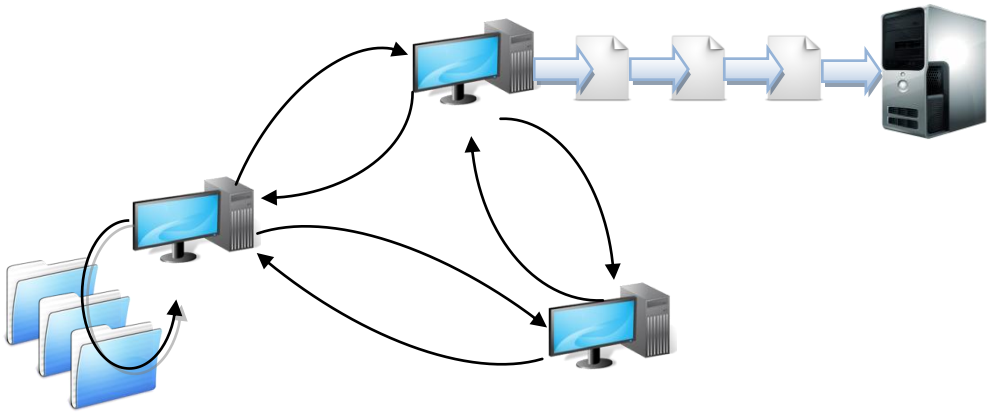
CHALMERS



Distributed Document Processing

Search index optimization by data preprocessing and workload distribution

Master of Science Thesis in the Programme Networks and Distributed Systems

JOHAN SJÖBERG
STURE SVENSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2009

Distributed Document processing
Search index optimization by data preprocessing and workload distribution

JOHAN SJÖBERG
STURE SVENSSON

Examiner: JAN JONSSON

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

Cover: The illustration depicts a system where a few file servers collaborate to effectively index the combined contents of each server to the indexing server.

Department of Computer Science and Engineering
Göteborg, Sweden June 2009

Preface

This paper is authored as a Master's Thesis on the institution of Computer Science at Chalmers University of Technology by Johan Sjöberg and Sture Svensson. 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 Karl Jansson, and Findwise executive Bengt Rodung. We would also like to thank our examiner Jan Jonsson at Chalmers.

Abstract

This thesis investigates the possible improvements to indexing files stored on servers in a local network; it is a known fact that the task of indexing is very time demanding and network consuming. At the same time the servers possess potentially unused processing capabilities. The proposed improvement given by this thesis is to distribute the tasks of text extraction and data processing to the idle processing capabilities of the servers. In addition to the theoretical basis of the improvement a working java prototype is also constructed. The prototype is designed to be capable of interoperability with virtually any existing indexing service via a unison adapter interface. It's also constructed to be able to handle any file type by an extractor interface. In addition the system also provides network synchronization and load distribution mechanisms. The result of the investigation indicates that the gains from the constructed system are substantial, especially regarding decreasing the magnitude of generated network traffic as well as reducing the overall time needed to perform the indexing operation. Relieving the index server of some work also implies that less powerful server configuration is necessary to effectively perform the indexing task.

Sammanfattning

Denna uppsats undersöker möjlig förbättring av att samla in nätverkslagrade filer för indexering. Det är ett känt faktum att indexering är en mycket tidskrävande och nätverksbetungande uppgift, samtidigt som servrar som lagrar filer har potentiellt stora mängder outnyttjade resurser. De föreslagna förbättringarna i denna uppsats baseras på att fördela ut textutvinning och databehandling till ledig processeringskapacitet på servrarna. Utöver undersökningar av den teoretiska grunden för förbättringarna har även en applikation skrivits i java. Prototypen är utformad för att vara kapabel att samverka med i princip alla befintliga indexeringstjänster via ett adaptergränssnitt. Den är också byggd för att potentiellt kunna hantera samtliga filtyper via ett textextraheringsgränssnitt. Utöver distribuering så tillhandahåller systemet även nätverkssynkronisering och belastningsdelning. Resultatet av undersökningen visar att vinsten från distribuering är betydande, särskilt när det gäller att minska nätverkstrafiken men även på den totala tidsåtgången för indexeringen. Den minskade belastningen på indexservern leder även till att en mindre kraftfull server kan användas för att utföra indexeringen.

Contents

1 Introduction

1.1 Background

Many companies and organizations today have large amounts of data spread over a multitude of locations. It is common that the set of data is continuously growing with time, in both quantity and complexity. When both users and data may be widespread, any local technique of accessing the needed documents may not be adequate; which violates the need for a document to be easily accessible by the end user. That is, it should be easy to find and access a specific piece of data that is requested. Typical examples of data are documents, images or videos. In the cases relevant for this thesis, any data is considered to be stored on any number of computers in a network. The terms *file server*, *content provider* or *node* will be used interchangeably to describe a computer containing data. A fundamental problem is that in order to acquire a specific piece of data, the file server which hosts the file will have to be identified first in order to retrieve it.

To be able to locate where a specific file is stored, it is desirable to have a simple interface to be able to search through an entire set of data, denoted as a collection. A collection can span over many file servers and may contain an arbitrary amount of data. For a search to be done effectively, the data needs to be indexed. An index is a database containing searchable information about a file along with a way of accessing it. The main approaches for generating an index is to have a centralized server either fetch or push files one by one and extract all the searchable information. This approach puts almost all the work on a single point in the system, and this is particularly bad since the server may also provide the searching service. Secondly the fetching of large amounts of data puts a lot of pressure on the network and might interfere with other services using it.

1.2 Purpose

This thesis looks at the possibility to remove a substantial amount of work from the indexing server by distributing work to the available servers in the network. This will also lead to a lowered utilization of the network since less information is propagated due to that extracted data can be sent instead of sending entire files.

The first part is to utilize the surplus processing capacity of the servers to perform information extraction and leave the index server to perform the indexing and offer the searching service. Since only searchable data is sent the amount of data on the network will decrease. The second part is to make all the file servers cooperate with the processing to maximize the utilization of the capacity of a group of file servers. Another improvement

from cooperation is the option to synchronize the network access in order to achieve a steady load on the index server. These enhancements provide a more natural boundary where the central servers can focus on providing service directly to the users. At the same time the file server resources are better utilized and network bandwidth is exploited more conservatively.

1.3 Goal

Other than this report itself the output from the thesis also includes developing a Java-based prototype to interface with a number of preexisting indexing systems with respect to network communication. This consists of algorithms for information extraction and an algorithm to share data between nodes and merge indexing results at the central node. The system will be designed primarily to be used by the file servers to be able to share data and communicate effectively. This software should be general enough to be able to be plugged in into most indexing systems via a system specific adapter. It should also be possible to add processing capability for new types of files via plug-ins.

1.4 Delimitations

The context of this thesis is concerned with how document processing can be enhanced in the scope of an indexing system. A number of generic mechanisms will be introduced and evaluated based on system- and network architecture. This dissertation is not to be considered a guide to any existing indexing system, although some major vendors will be examined for the purpose of comparison and evaluation. The thesis will be backed by software designed to provide a general mechanism to interface a number of indexing services. Each file server in the network will run an instance of the software responsible for propagating any information to the indexing server. It is assumed that any data of interest are located as a file on the file server. Although it is possible to perform indexing of data which are not files, this will not be considered as any data can always be represented in the intermediate form of a file.

It is useful to define what is considered by the term enhancement in this context. For instance, the diverse indexing system implementations, ranging from open source to enterprise indexing engines, already offer suitable environments for most scenarios. The enhancement focused on here is within the gathering and propagation of information to the indexing engine, with attributes such as workload distribution and directives to limit network traffic and thus congestion. In short, we will examine how to effectively feed the indexing engine with data over a local network. This does not imply a responsibility to provide the end user, upon a search query to the indexing engine, with the data itself. The method to retrieve any data from a file

server is for the server administrator to configure by an appropriate technique, such as network file sharing, applying a web server, or similar.

1.5 Disposition

The contents of the report are organized as follows. In section 2 we present an analysis of the problem statement and provide general theory to support some of the cornerstones in distributed computing. In section 3 we describe the choices made regarding the implementation of the system as well as identifying the chosen solution. The following section 4 examines how the system can be benchmarked in comparison to existing solutions. Section 5 provides a discussion of the key results and their validity. Finally, section 6 contains a short summary of the system key findings as well as application areas and issues of further development

2 Analysis

The introduction identifies the need of information retrieval and addresses why it may be a complicated procedure. To begin the analysis, it's important that the basic concepts are well understood. For thoroughness, the user is defined as an employee of any organization where the persons job details necessitates frequent access to documents or files spread over the organizations network. The user is presented with a consistent user interface accessible over any major browser vendor which serves as the front end of the search engine; which in turn ranges from enterprise class to well-established open source vendors. This demarcates the users' view of the system, which will not be examined further in this thesis.

Next the search engine scope is defined. It, as already presented, provides a consistent user interface for locating documents and files. A search engine is a complex system with many modules, from which we identify three key elements: the feeder, the indexer and the search front end. The feeder and indexer can be combined into the indexing server. The indexing server, visited in the following sections, provide a uniform API to transform a file into a searchable entity. Although the API's differ between vendors, they each provide a method to receive files from network. This is the point where our application interfaces with the indexing system.

The basic means to provide the indexing server with files is to arrange a method of accessing the files over a network connection and then define rules for the indexing server used to fetch data. The approach seems well supported, relatively easy to configure, and straight forward. On the downside, it's very inflexible and ineffective as each file manually needs to be retrieved from the network.

Since any data is considered to be located within an organizations internal network, the problem is not locating the file servers, but actually retrieving the data effectively. To combat the limitations of fetching data, we consider various techniques for pushing data to the indexing service by interfacing its API directly. For this purpose, the term feeding is introduced to define the operation of transferring data from a content provider to the indexing server.

From a practical perspective, the indexing system and file servers combined perform a system of data retrieval, communication, processing and indexing. The file servers that are a part of the system are considered as resources which are completely or partly available to perform tasks designed to improve system throughput. Section 2.1-2.4 examines methods for efficient communication throughout this system, while 2.5-2.6 threats techniques to which a file server can serve as preprocessors for the indexer as well as the introduction of some of the functionality of an indexing server.

The analysis section thoroughly describes some of the most important

properties of distributed system and how they can realized. The interested reader is encouraged to study those sections, however it is not necessary to be able to understand the concepts of the systems as presented in section 3 and forwards.

2.1 File- and index server communication

The primary function of the file servers is to feed the indexing server with data. The communication is inherently important in order to provide effective service. There are a number of potential approaches, each with respective drawbacks that are important to consider in order of maximizing system performance. Two basic types of communication principles will be examined below.

2.1.1 Direct Connection

The first and most straightforward approach is to let all the file servers connect to the indexing server and forward its data without any coordination to the other file servers. While simple, this technique imposes congestion issues to the network adapters. As the number of file servers' increase, connection multiplexing overhead, bandwidth contention and memory issues leaves a lot to desire. Since the system is designed with the compatibility with basically any indexing system, there can be no consideration of any server-end solutions for these problems. Naturally, each file server will be prevented to flood the indexing server by the protective mechanisms of TCP. However, given enough simultaneous open connections a server may exceed its memory boundary and in the worst case even halt the offending application. In applications not bound to the use of TCP, the network is even more susceptible to uncontrolled flooding. Consider the use of UDP with an application level resend mechanism for network data; theoretically this mechanism could fill the index servers' network buffer and cause application malfunction.

2.1.2 Intermediate Server

The second approach is an attempt to remedy some of the shortcomings of the direct connection by introducing an intermediate application that all the file servers are connected to; denoted as a *connector*.
An immediate advantage is identified in the way that the file server system becomes pluggable to virtually any indexing system by having the connector adapt any information received from the file servers into a format accepted by the indexing system. In the absence of a connector, this adapter has to be present in each file server directly.

Since the nature of the connector association allows central coordination, techniques such as buffering and sequential access control can be implemented to conform to a steady stream of data. However, this also has

some major drawbacks. In the case when the connector is located on a separate computer than the index server, the network traffic will increase due to the fact that all traffic sent to the connector also will be forwarded to the indexing system. The advantage of this setup is that some of the workload is distributed between the two computers. Another reason may be if the indexing server is bundled with proprietary hardware that does not allow the addition of new software. Due to the fact that sending batches of messages is more advantageous than sending them one by one, the connector would have to buffer data which puts a memory requirement on the machine. In the case were the connector and the index server is run on the same machine this memory requirement is added to the requirements already present for the indexing.

2.2 File server synchronization of network access

The two presented methods of interacting with the indexing server each presented flaws rooted mainly by the lack of interaction between file servers for coordination. This section examines means of synchronization between the file servers. We denote the set of file servers as a cluster . Coordination offers semantics to create a synchronous approach towards data propagation; allowing the file servers to send data sequentially, one at a time, with respect to each other. Several techniques to perform this separation will be introduced below. Further, section 2.3 will address additional exploits of cluster synchronization.

2.2.1 Time division

Time division in this context is used to create time spans in which participating nodes may perform arbitrary work in their respective slots. In practice every node is given a specific time slot, ranging from times τ_{start} to τ_{end}, where the node may utilize the network. If a node does not have anything to send the time slot is wasted if using pure time division. The method also has significant requirements on time synchronization across the network. It requires frequent clock synchronization between all nodes to limit clock skew and mitigate the effect of clock drift. To provide timely delivery of data, it's important that all nodes share the same view of the timeslots, and that no nodes exceeds its end time τ_{end} The shorter each time slot is, the increased demand is put on accurate temporal synchronization but less waste on idle operation. Assigning a timeslot needs to be performed with great care. Neither the direct connection nor the intermediate connector can propagate fragmented tasks to an index server effectively. Ideally, the length of a timeslot is proportional to the time necessary to perform a given number of complete file transfers. Unfortunately it proves a hard condition to meet due to varying file sizes, network anomalies and run time network

contention.

2.2.2 Token time division

By introducing a token system the time division technique can be extended to avoid wasting time slots during periods of inactivity. The token specifies which node is allowed to have network access and supersedes the time division by clock synchronization mechanism. Associated with the token is a maximum time, or other condition, by which it may be held. For instance, a token may expire for a node after a given amount of milliseconds or whenever a network transfer limit has been reached. Whenever a token expires it needs to be forwarded to next waiting node. The advantage of token time division is that clock synchronization is of no importance in order to synchronize network access. Once a token is received, a node may utilize the network for duration specified by the token. An inactive node may thereby forward the token immediately to grant next node earlier access to the network. The approach does however increase the need for robustness against node failures, since a new token must be created if the node holding the token should fail.

2.2.3 Global queue time division

Global queue time division uses the idea of a global network arbitration queue shared among all the nodes in the cluster. When a node desires network access it places a request token at the end of the queue. Whenever a nodes request reaches the head of the queue that node is eligible to use the network. The privilege is granted for a certain time limit or until some condition is met, whereas the request token is removed from the queue and the access is transferred to the next node.
An advantage with this approach is that it doesn't pursue a strict round-robin approach, which usually is the case of the token time division technique. The queue can be an implementation which allots, for instance, a larger time window to busy nodes, or arranges the ordering of tokens based on priority constraints. At the same time, unwanted properties such as starvation must be guarded against whenever priorities are part of the access control mechanism. With an increasing amount of algorithm negotiation patterns, the implementation complexity quickly grows.

Although the approach is more versatile than the token time division technique, it also introduces *consensus* overhead. A global queue requires an implementation where all nodes in the cluster share the same view of the queue. Without a centralized solution, a continuous coordination between clients is required and thus presents some communication overhead. However, once established, it can serve as a basis to extend functionality to include complex and desirable features such as load balancing.

The queue can also be implemented centrally by revisiting the connector

procedure. This would relieve some of the communication overhead and significantly decrease algorithm complexity. Another advantage arises in the form of managementability. The connector could serve as a central location to administrate the system. Still, the queue implementation realized with a connector would introduce a single point of failure; a property which is most undesirable.

2.3 Group service

To induce the characteristics of a dynamic system which adapts to environmental changes, a number of services must act together to create a dynamic bind. A node must be able to locate other nodes in the system, detect changes in the environment and adapt to these. The orchestration and coordination of activities require communication directives resting on the principal pillars of the group membership service. Hence the group service provides an abstraction to communicate arbitrary messages between any amounts of peers in the group. To satisfy a distributed view shared among all peers in group, the network should employ a *synchronous* message passing mechanism. Synchronous in this sense refers to that there is an upper bound of the delay of the arrival times for messages in the network. It guarantees that messages sent will eventually be delivered to the receiver. The usefulness of this comes in play when a piece of information needs to be shared by all nodes in the network. An excellent implementation is to maintain a distributed table of all members currently in the group, allowing new nodes to join or leave at any time. To maintain a consistent view of the table throughout all nodes in the network, certain guarantees must be made to the messaging system. The guaranties will each be considered in the following section.

2.3.1 Discovery service

The first challenge of maintaining a dynamic group relationship arises when new nodes must locate an existing group, if one exists. This is usually not a problem if a central server can perform this lookup service, similar to how e.g. a Dynamic Naming Service (DNS) is implemented today. If a system is considered distributed in its entirety this entity cannot exist. There are two main variants to serverless discovery which we denote as push and pull. Using the push methodology, nodes already participating in a group advertise their presence regularly. A newcomer to a network can pick up this message and establish contact to the group. The immediate drawback of this approach is that messages must be sent regularly for each multicast enabled location. A larger interval induces increased delay to any node possibly listening, while shorter intervals increase network traffic. A node may initiate the first point of contact by a preconfigured set of acceptable nodes, or retrieve it from

a service like DHCP . Although using external services is a clever way to discover a group, it is also limited to physical subnet to where e.g. the DHCP server resides.

The pull method works opposite to the push implementation. The multicast expanding ring [1] exploits this scheme. A new client sends a multicast query to an address designed to locate existing groups. Any peers subscribing to this address may responde. If no response is returned, the node can form a new group with initially only itself being a member. The new node also subscribes to the multicast address to be able to respond to queries from other newcomers. The concept of such procedure can be examined in Figure 1.

The expanding ring concept entail a method of sending the multicast request multiple times with increasing time to live values (TTL) until enough responses are collected. The TTL value is a limit to the number of router-hops a message can survive before being discarded. The multicast principles examined here are applicable to the specific case of IP multicast and requires dedicated router support; namely an implementation of the Internet Group Message Protocol stack (IGMP). A known java service implementing group discovery by multicast is the Jini system [2].

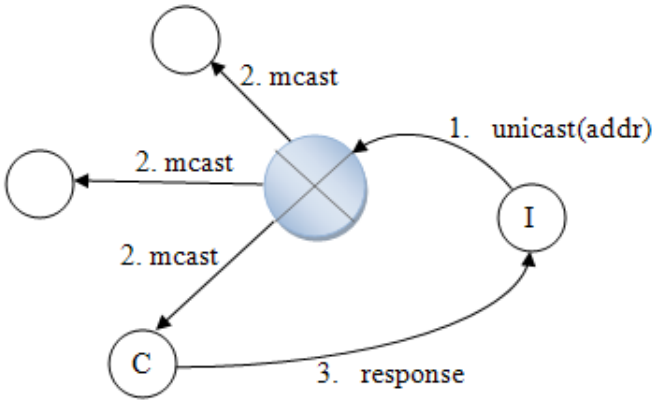


Figure 1: A demonstration IP multicast discovery mechanism. The initiator sends a unicast message to the multicast enabled router, which forwards the multicast to any registered clients. In this example a coordinator node C is responsible to reply to the initiator. After contact, the initiator can issue a join message and register itself to the multicast address

Nodes known to coexist only on a local network can employ a number of means of locating a group other than IP multicast. A simple approach is to let each node listen to messages from the network on a predefined port. Arriving nodes is then given the opportunity to contact any node in the group by performing a local broadcast. The issue becomes more complex

when nodes are physically separated by an internet connection. By the fundamental design limitation of the internet protocol and the way it utilize sub-netting, there are no wholly distributed discovery service that we know of which doesn't rely on a central naming service or predefined node contact information.

2.3.2 Message propagation

A well known algorithm family used to solve many difficult problems in distributed systems is *Propagation of Information with Feedback* (PIF)[3, 4] . It is used to propagate a message to all nodes in a group, which upon termination lets the algorithm initiator know that the message have reached all nodes in the system. When invoked by a node, a message is propagated to all its neighbors. A neighbor is considered to be a node with a physical bi-directional network connection to the node in question.

Let a *leaf* node be a node with only one neighbor. Further, let *first link* denote the communication channel where a node received its first message from. Every node, upon receiving a message, forwards it to its other respective neighbors in turn as depicted in Figure 2. When a leaf node receives the message, or when a node has received the message from all its neighbors, an acknowledgement message is generated and returned to the first link as illustrated in Figure 3. Eventually the initiating node will receive a response to the message from all its neighboring nodes. When all responses are collected the message has reached all nodes currently in the group. This type of algorithm reside under the category of *flooding* algorithms and serves as natural way to propagate a message to every node in group; constituting for the first essential part of creating a distributed group membership service.

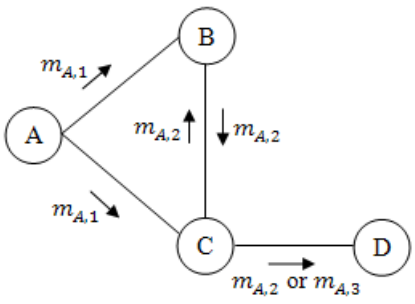


Figure 2: Illustrates the propagation path taken for a message m sent by the initiating node A.

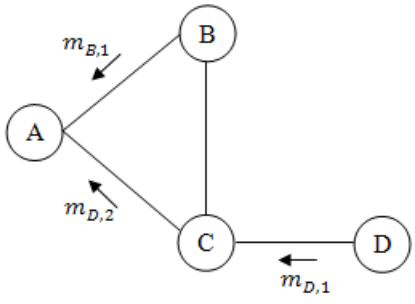


Figure 3: Note the feedback path where the acknowledgement message is only propagated via each nodes first link

2.3.3 Message ordering

Communicating a message to a group doesn't by itself provide any guarantees to ordering. Ordering is an essential property needed to provide a consistent group view. Consider a node A sending two consecutive messages, a join message followed by a leave message, to a node B. Since an Ethernet network doesn't provide any built in ordering mechanism, the leave message can arrive at node B before the join message; leading to an obvious inconsistency between how the two nodes perceive the group state. Ordering will be considered in a multicasting environment. The reason multicasting is considered is because it's a natural approach to propagating a message to multiple recipients. In the most trivial sense, multicasting is implemented as sequential routine by sending a message one by one to each intended recipient. A better use would be to employ IP multicast , which is not considered here, but provides equivalent but more effective functionality for this discussion. Let's borrow the definition from [5, Pp.484-496] to define ordering in the context of multicasting. The routine $multicast(g, m)$ represents a multicast of the message m to the group g.

FIFO ordering: If a correct process issues $multicast(g, m)$ and then $multicast(g, m')$, then every correct process that delivers m' will deliver m before m'.

Causal ordering: If $multicast(g, m) \rightarrow multicast(g, m')$, where $\rightarrow$ is the happened-before relation induced only by messages send between the members of g, then any correct process that delivers m' will deliver m before m'.

Total ordering: If a correct process delivers message m before it delivers m', then any correct process that delivers m' will deliver m before m'.

The essential property of the distributed group service is that every node shares the same view without the presence of a centralized server. Neither of the two initial ordering definitions proves sufficient to produce a consistent distributed group view. To see why, a set of events have been derived which illustrates a case where FIFO- and causal ordering are insufficient. The example can be observed in Figure 4.

Note that causal ordering implies FIFO ordering while total ordering, which is the most restrictive of the three, doesn't necessarily have to. To attain a better understanding of the causal property its semantics requires more in-depth examination. We begin by expanding the definition of the happened-before '$\rightarrow$' relation as defined by Leslie Lamport, one of the pioneers of distributed systems, using the following three conditions [6]:

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

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

The definition of concurrent implies that if there are two events a and b, and they are not causally related to each other, they are considered concurrent in a distributed system. Even though the two events can occur at two distinctly different points in time they are still considered concurrent. The problem resides in the way computers perceive the notion of time. Clock synchronization is a well known problem which explains the difficulties in synchronizing clocks between computers in a network. As a result time stamping messages using physical clocks is not sufficient to demonstrate a global absolute time. In causal systems however, a node can only decide order by observing messages communicated between nodes in the group. Thus the notion of time used in systems of this type is thereby, for a node k, based on sent and arriving messages $m_1^k, m_2^k, ..., m_n^k$.

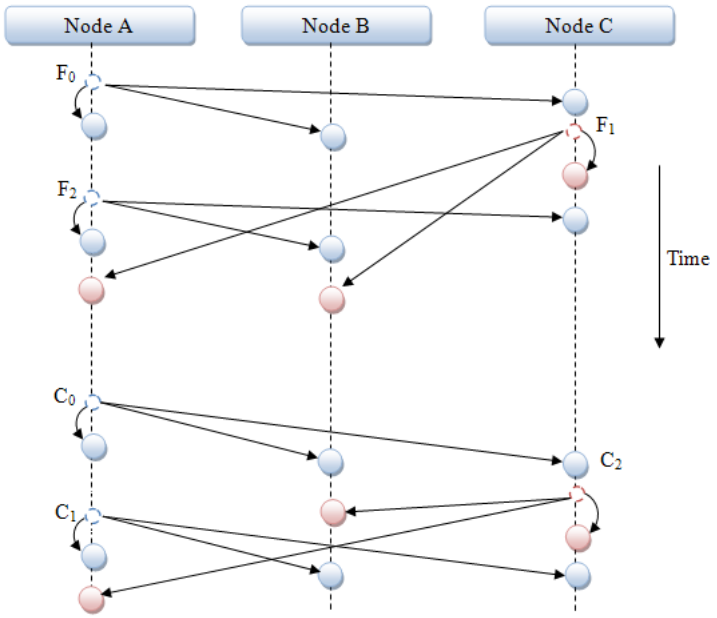


Figure 4: Demonstrates a set of events for which FIFO- and causal ordering are insufficient to provide a consistent delivery of messages such that all nodes will perceive the same order.

Total ordering protocols can deliver messages in an arbitrary order, alas with a guarantee that every node in the group delivers each message in the same order. A protocol can however adapt to respect e.g. FIFO order. Lam-

port defined a method to achieve total ordering by time stamping messages with logical clock values [6], which is also considered by e.g. [7, 8]. Let $L_i(e)$ be the logical clock value at node i for the event e. As stated before, an event can be e.g. to send or receive a message. The logical clock value does not need to relate to physical time, but can be an arbitrary number implemented using i.e. an incrementing counter. Using the notion of logical clocks we can define correctness of time based on the order in which events occur. We repeat the clock condition presented in [6]:

> *Clock condition*: For any events a, b:
> $$if\ a \rightarrow b\ then\ L(a) < L(b).$$

It follows that:

1. A node i increments L_i for each successive event.
2. If e_1 is the event of sending a message from a node i, and e_2 to receive the message at node j, $L_i(e_1) < L_j(e_2)$.

The condition is enough to satisfy the happened-before relationship, as presented above, using logical clocks. To conform to (2), each message sent must contain a timestamp $T_m = L_i(e_1)$. The receiving node j can then assign its logical clock value to $max\{T_m, L_j\} + 1$. The final condition needed to provide total order is to break ties when two events are seemingly concurrent. A deterministic approach is to extend the happened-before relationship using the following condition for nodes i and j:

$$e_1 \rightarrow e_2 \text{ if } \begin{cases} L_i(e_1) < L_j(e_2) \\ L_i(e_1) = L_j(e_2) \wedge i < j \end{cases}$$

What's been examined is a logical clock system independent of physical time. As a side note it is however possible to prove the correctness of total order multicast using synchronized physical clocks with an upper bound on the clock skew [6, 9].

2.3.4 Multicast reliability

The semantics of total order does not assume or imply reliability. For example, if a correct node i delivers message m and then m' to a correct node j, then node j can deliver message m without delivering message m' or any other subsequent messages. An algorithm which provides total ordering as well as reliability is commonly referred to as *atomic broadcast* [10, 11]. What constitutes reliability in multicast is defined by Hadzilacos and Toueg [12] using the following three properties:

Validity: If a correct node multicast a message m, then it eventually delivers m.

Agreement: If a correct node delivers message m, then all correct nodes in the group where m was multicasted eventually delivers m.

Integrity: For any message m, every correct node in the group delivers m at most once, and only if m was previously multicasted by $sender(m)$.

The term deliver is used to denote that a message has been received by the correct process in the node and that any content in the message can be applied. For instance, on receiving, the node places the message in a hold-back buffer. After all nodes have agreed to the conditions for which the message contents can be processed, and once those requirements are fulfilled, the message is delivered and possibly consumed.

There are two primary methods used when designing totally ordered reliable multicast. The first employs the concept of a sequencer node to aid the global ordering of events. The model was first introduced by Chang and Maxemchuk [13]. An actual implementation of a sequencer protocol for the Amoeba system was proposed by Kaashoek *et al.* a few years later [14, 15]. Figure 5 and 6 illustrates the typical operation of such system for both unreliable and reliable operation. Since the message is initially multicasted to all members in the group, it can be noted that the latter of the two modes of operation allows for node failures in the group. During a failure, it is necessary for the group to agree on a new group view and assign a coordinator to it. This topic will be examined in the next subsection.
Any messages received during this recovery procedure must be stored locally by each node. Once a new group has been defined with an appointed coordinator, the nodes can synchronize their operation in time by replicating their message buffers internally in the group. The coordinator can then resume its duties by forwarding acknowledgement messages for all messages in the buffer.

The second technique considered to induce reliability to totally ordered multicasting is similar to the ISIS algorithm defined by Birman and Joseph [16, 5]. It requires no coordination efforts by a single node, but instead moves this chain of responsibility to each node in the group. A node i upon multicasting a message includes a sequence number T_i. The sequence number is calculated as the largest observed sequence number so far, plus one. For any node k receiving the message, a response is generated containing the new sequence number $T_k = max\{T_i, T_k\} + 1$. The originator collects all replies and selects the largest one m as the final sequence number. In the last step, m is multicasted to all nodes in the group by which it is used to set to the definite order of initial message sent by the originator. The procedure in general is outlined in Figure 7.

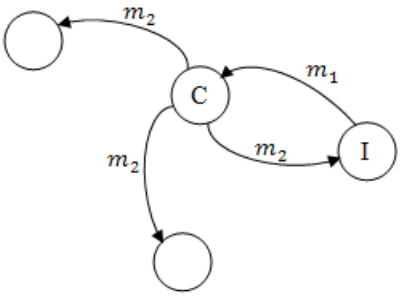


Figure 5: A typical operation of unreliable message communication where a coordinator node C acts as a sequencer and node I as the initiator.

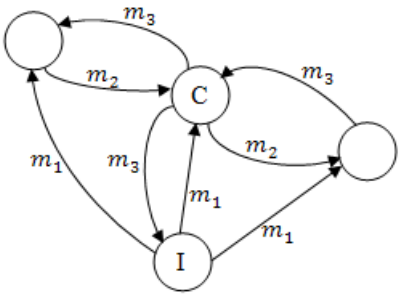


Figure 6: A reliable message path using a coordinator as sequencer. Message m_1 can be considered a request, m_2 an acknowlededgement and m_3 an accept message.

Some details have been left out explaining the algorithm. For instance, each message must be equipped with a unique timestamp to be able to separate messages apart. The final message must likewise carry an identifier to the message of the originator to be able to assign it the final agreed upon sequence number. Compared to the coordinator-influenced reliable multicasting algorithm as presented above, the algorithm induces more traffic and is thereby subject to network congestion in a more direct way. However the traffic is evenly distributed and does not introduce a bottleneck at the coordinator; neither regarding processing capabilities and contention. From a practical standpoint, the implementation of a coordinator in distributed systems is slightly more complicated than constructing a system where all nodes are considered equal in operation; particularly during failures where state recovery and coordinator assignment are considered. It is clear by observing the algorithm that for all correct nodes, the message is either consumed by all nodes or by none. A failing node will result in incompleteness of any of the protocol steps. Just like the algorithm involving a coordinator, any change to the group must result in the distribution of a new group view.

2.3.5 Group management

So far we have considered how a group can be located by new nodes and how reliable messaging can be used to provide a consistent view of shared objects. The following section examines how to maintain a dynamic shared group state in the presence of failures. We start by introducing the concept of virtual synchrony as defined by Birman and Joseph [16]. The term synchronous is used to describe an environment with totally ordered reliable multicasts, and where events such as node joining, node failures etc. occur in the same order on every node. As seen before, such properties are expensive

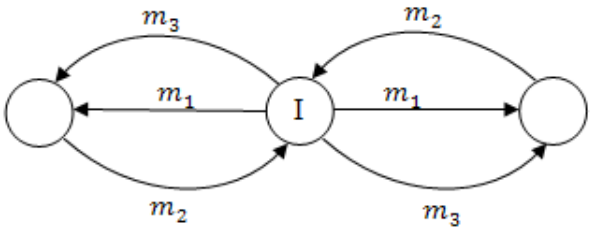


Figure 7: Reliable total order protocol without a coordinator node. The initiating node i communicates message m_1 to the members of the group. Each node generates a response message m_2. The initiator, after collecting all replies, sends a message m_3 to the members of the group containing the final sequence number to assign message m_1. At this point each node can continue and consume the message m_1.

in terms of the number of messages which has to be communicated between all nodes. It is however beneficial in terms of simplicity when processing events such as node crashes. Even keeping a request queue to access e.g. a mutably exclusive resource is no more complicated than maintaining a simple distributed FIFO queue. Equivalent logic applies to a node joining or leaving the group.

Failures in a distributed system are most of the time masked by mechanisms such as message retransmissions. To detect node failures, we make the assumption that a node can only fail by crashing, an assumption reasonable to most systems. Bring in mind from the synchronous definition that there is an upper bound on message delays, implying that a node can be considered to have crashed after exceeding this timeout due to some event. Although the synchronous property allows reliable failure detectors [5, Pp.469-472], it's not of great importance. Stelling *et al* [17] argues that unreliable failure detectors has advantages in that they are more scalable, simpler, and more effective due to that no atomic message protocol is needed to provide the service. For this purpose it is often enough to suspect that a node has failed. In a virtually synchronous system, all nodes will share this suspicion and can collectively discard the faulty node from the group.

Assuming the group has a coordinator; a special case occurs if the coordinator suffers a crash. At this point, electing a new coordinator should be given precedence. Leader election is a well known problem, and there exist a wide variety of algorithms [18, 19, 20]; although they will not be considered here.

In previous sections, it was shown that the coordinator can act as a sequencer. It may also be useful for other purposes. Consider the event of a node joining the group. Then it's the responsibility of the coordinator to

supply the new node with any shared data as well as the current state of the system. Technically this service can be performed by any node in the group. However it's often convenient for the implementer to have predefined software component perform this operation without having to decide which node should take the responsibility from case to case. This software component is only active on the node currently acting as the coordinator.

A fundamental problem using Ethernet is its unreliability. At any time an intermediate connection between peers may be lost due to e.g. congestion or the malfunction of a network router or switch. Figure 8 depicts a particular scenario where nodes become physically separated due to a malfunctioning router. Even though the nodes may reside physically close to each other, the inability to reach each other would lead to the creation of subgroups. In this case the subgroup without a coordinator would elect a new one. The problem arises when the router becomes functional again, the two groups needs to merge back into one group, discard one of the coordinators and synchronize the state. An alternative approach is to let one group continue their operation, while the other is suspended until contact can be reestablished again. One such algorithm is based on *quorum consensus* [21]. Upon a network partition, the groups must decide for themselves which group may continue operation. The privileged group can be established using majority, or in other words by acquiring quorum. More advanced usage is defined as well. In a system with replicated data, read quorum R and write quorum W can be considered as two weighted operations following two simple rules: $W >$ half of the votes and $R+W >$ total number of votes for the group. These properties ensure only one group can perform changes to shared objects.

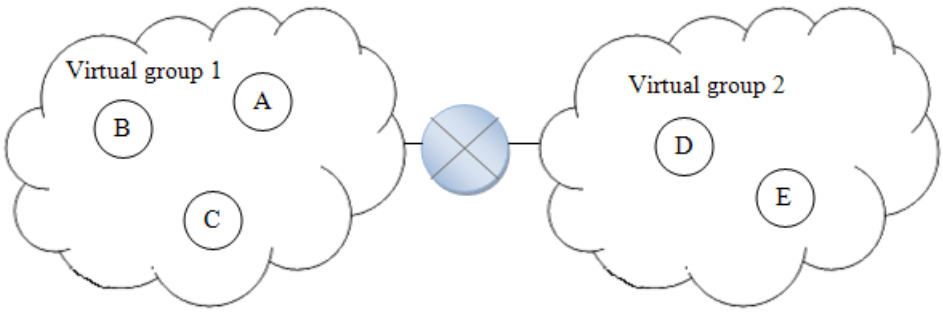


Figure 8: The malfunction of a communication link can lead to network partitions. The two virtual subgroups depicted may be the result of a malfunctioning router or switch. As a result, to groups would exist simultaneously without the possibility to communicate with each other.

2.4 Load distribution

A primary motivation to use distributed systems is to be able to exploit load distribution in any of its forms. Throughout the thesis we will use the terms *load distribution* and *load sharing* interchangeably. A distributed system can be constructed with respect to properties such as scalability, reliability, robustness etc. Consider by which means these properties are fulfilled using variants of load sharing in existing web technology, where the hosting of a single web service may employ servers in the thousands. Let's recite three desirable load distribution properties as presented by Wang and Morris [22]:

i. Optimal overall system performance-total processing capacity maximized while retaining acceptable delays.

ii. *Fairness of service* - Uniformly acceptable performance provided to jobs regardless of the source on which the job arrives.

iii. *Failure tolerance* - Robustness of performance maintained in the presence of partial failures in the system.

We classify load-sharing systems into three categories: centralized, decentralized and hierarchical. A system can possess either of these characteristics, or combine several in a hybrid model. Figure 9 depicts the topology to which each of the classes belongs. Note that we consider ring-networks to be a special case of the decentralized system. The topology sets the basic restrictions to which type of load distribution algorithm can be designed. The hierarchical model is commonly seen in web servers, where a server acting as the front end receives the initial request and forwards it down the hierarchy to an available server. An immediate drawback of this scheme is the need for load managers to arbitrate work between its children; alas nodes are not uniformly assigned the same principles of operation. We focus on the decentralized topology as it enables maximum flexibility while not introducing any single point of failure by employing a centralized server.

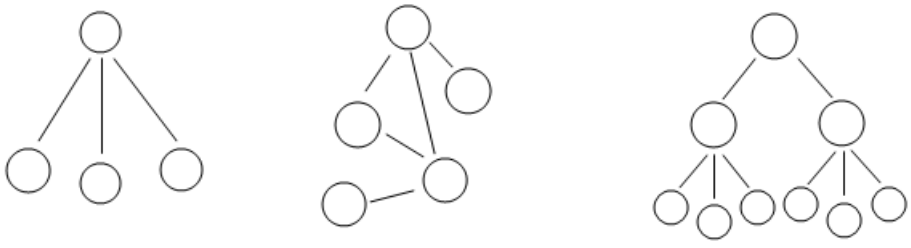


Figure 9: Network topologies for systems of (left-to-right) centralized, decentralized and hierarchical type.

The next distinction that we make to load distribution relate to how it's initiated. If a node determines which peer to distribute to, we denote this *source-initiated* distribution. On the other hand, if a node decides to acquire tasks from another peer, we call this *peer-initiated* distribution. In the latter case, a node possessing surplus processing capacity can request tasks from any nodes in the network. An advantage to this approach is that a node doesn't need to maintain a global status vector for each node in the group to decide if there exist distributable tasks, but can issue a query to a set of nodes at any time. During stale times, the operation experiences the characteristics of a polling operation. This effect is mitigated in source-initiated distribution in that a node only needs to locate a distribution peer when there are distributable tasks available. Locating a node can be performed either by a query multicast to the group or by maintaining a local vector carrying surplus capacity of each peer. The vector can be implemented based the techniques for reliable multicast visited in previous sections. Updates to the vector may be performed either by dedicated messages for changing status or by piggybacking information on existing messages.

2.4.1 Granularity

Distributing the workload between peers in a group can be performed differently based on the origins of the data characteristic, the semantics of the network connection, and the degree of separation of the problem that can be attained. Any means of distribution must be justified in order to increase efficiency. It is not intuitively obvious for which data sets a performance increase can be at all observed. If not carefully engineered, the result may have the opposite effect on performance. To examine what degree of distribution can be applied, we use the following two terms defined by Berger *et al* [23].

Load distribution - A load distribution is an assignment of work to a set of processing elements

Load balancing - Load balancing is the process of transferring units of work among processing elements during execution to maintain balance across processing elements.

We consider load distribution in the dynamic sense. That is assigning tasks to processing elements at run time, in contrast to static scheduling where tasks are assigned before the program execution is started. The rationale is that events in the back-end of the indexing system cannot be decided beforehand; files may enter or leave the system at any point during an execution. Note that the term load distribution is sufficient to describe the exposure to the degree of distribution to apply in this context. Load balancing implies a finer granularity aiming to even out (CPU) activity among the nodes in the system.

2.4.2 Distributable tasks

At this point it's useful to remember that the indexing engine likely is of proprietary license and thus cannot be modified, which means that the system backing this report is concerned with the gathering of files from any file servers in the system. Previous sections outline the main tasks of the system. In short, the steps can be summarized as data retrieval, extraction, processing, and finally the propagation of the result to the indexing server using a well-defined structure. To this attaches the desire to reduce bandwidth. The two actions, data extraction and data processing, can immediately be recognized as tasks possibly being subject to load distribution. The extraction process can logically only be split up into several smaller tasks by separation of its input data. At the finest granularity, this has the impact that a file is split into several small pieces and handed out to *free* nodes in the system. On the opposite end, files are communicated as a whole to other nodes in the system. Both imply sending at least the full content of the file over the network, working against the bandwidth reduction requirement. In addition there is the overhead associated with creating a connection to a neighboring node as well as the overhead induced by the packet-header; which becomes increasingly prevalent as the data size is reduced.

The second task which has the potential to gain from load distribution is the data processing task. Without further investigation of what constitutes a processing step, we can conclude that it performs some data manipulation and thus requires the complete data set as generated from the extraction phase. If this task is to be distributed combined with the extraction process, it must thereby logically be a requirement to facilitate both steps at the same node with the granularity of complete files.

To conserve as much bandwidth as possible, we expose only the processing steps to load distribution; sending only the minor dataset produced by the extractors onto the network. The alternative approach would be to distribute the extraction process as well, however the additional network traffic induced is severly increased. As the processing step grows in complexity and execution time, the magnitude of performance gains by load distribution is increased.

A typical index system usually returns a response message to each file received, expressing the status of the indexing operation. This message needs to reach the originator of a file, regardless of if load distribution is enabled or not. Figure 10 illustrates a typical routing behavior supporting this action, following the characteristics of a peer-initiated distribution. It is assumed that originator node is aware of the status of the neighboring nodes and can utilize the information to establish a distribution channel to a peer.

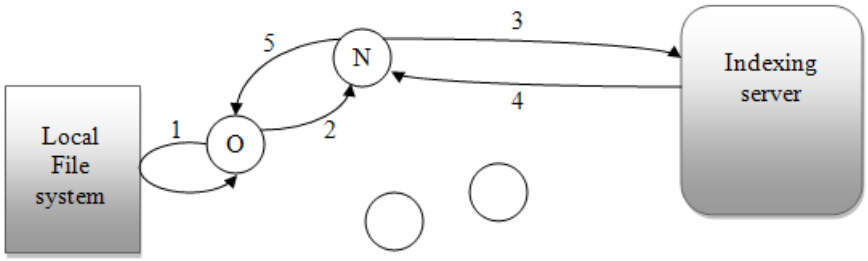


Figure 10: Illustrates a scenario where the result of a message must be routed back to the originator node O after it's been processed by neighboring node N. In the first step, a file is retrieved from the local file system by node O. Any desired content is extracted and the resulting text propagated to the neighboring node. A number of processing steps for the content may commence before the file is sent to the indexing server which also returns a response message upon indexing the file. This response message is forwarded to the originator in the final step which can upon examining the message content decide if the operation was successful or not.

2.5 Information extraction

As was mentioned in the introduction, the thesis aspires to reduce network load and offload the indexing server by exporting as much of its work elsewhere. Information extraction is a technique to address both these issues. A file stored on a computer contains alot of diverse information and a great deal of this might be of little or no interest for the purpose of indexing. For example a PowerPoint document contains a lot of style information, images, backgrounds etc. The main information of interest is the text component and various elements like author, file size, creation date etc. The term metadata will be used for this collection of information about a document.

There are two fundamental approaches to retrieving the desired text components. The first technique is to remove all irrelevant information from the file before passing it to indexing server. This method is called *information stripping*. We define this term as *to remove all the irrelevant data from a file*. When supplying information to an indexer the information stripping method have some disadvantages. First of all the relevant information is considered as the data content in a file, but this is unfortunately not a well defined term for this purpose. Many file formats are just containers which host a combination of textual data and binary data such as an image. The main problem associated with information stripping is that indexing a file should not violate the properties defined for that file type format; neither should the original file be subject to any changes. Hence, the information stripping method will have to create a copy of the file either in memory or disk before any data can be discarded. Alternatively the file can be copied

while ignoring the undesirable parts of the file if they are known beforehand.

The second approach involves locating and extracting relevant information from a file and to store this in some internal data structure used as a representation of the file. We denote this procedure as *information extraction* and define it *as to extract relevant information and store it in a specified structure*. This effectively removes any need to create a copy of a file. Moreover it is often possible to better control memory consumption. Consider a file combining textual data with binary data. The textual data can be located using a search routine without allocating any memory buffers, and thus suffers less overhead than the information stripping approach. This method is arguably more intuitive as well in terms of implementation.

2.5.1 Character encoding

The text component of a file is stored using a character encoding scheme such as UTF-8 or ISO 8859-1 for Swedish text literals. The primary problem is that it is essential to interpret the contents of a file using the correct character encoding scheme. Fortunately this information is commonly stored in the file format container. At a low level, a character is described by a sequence of bytes. Each character encoding scheme gives different meaning to how this sequence should be interpreted. Moreover, each character may be realized by varying length for different encodings. Some schemes requires each character to be of 16 bits in length, allowing up to 36,536 unique character combinations, while others are limited to 8 bits allowing only 256 unique combinations. The 16 bit (2 bytes) encoding requires twice the amount of physical storage to represent one character compared to an 8 bit (1 byte) encoding scheme. Character encoding has been a problem for a long time which led to the birth of Unicode.

The Unicode standard was devised to contain multilingual support, and does so very well with more than 100,000 different characters defined [24]. Unicode is commonly implemented by the UTF-8 encoding scheme which is widely supported. The problem is that many aging documents are often produced using legacy encoding schemes. Unfortunately it is never a trivial operation to convert characters between different encoding styles. Although any character defined in e.g. the 8-bit ASCII code set can be represented using i.e. UTF-8 encoding, the converse is not true. An implication to this is that the indexing server must accept documents encoded using a modern scheme such as UTF-8 to be able to express the many special characters unique for many languages. This forces the need to perform proper character encoding of file content already during the parsing phase.

2.5.2 Different file structures

Regardless of which method is used to retrieve information from a file there is need for some method to differentiate the relevant information from the irrelevant. In order to decide which sections of a file contain useful information, the structure of the file type must be known in advance. We will examine the major types of structures used today to represent files: *raw text*, *binary data*, *formatted text* and *xml structured text*. Each of these structures requires different types of techniques in order to differentiate relevant data from irrelevant data. The process of separating data is henceforth called parsing.

Raw text. A file stored in a file system by the structure of *raw text* comprises of a stream of characters terminated by an end-of-file marker. Files of this type are intended to be directly human readable and contains only textual data. This structure is very advantageous to parse since all information is already stored as a sequence unformatted text. The main component of work behind files of this type is simply to extract the desired metadata and send the full contents of the file to the indexing server. Examples of files stored in raw text are files with the .txt prefix and various log- and configuration files.

Binary data files are a files designed to be interpreted by some software before any useful information can be attained. Each binary format has its own, often proprietary, container format which needs to be interpreted. Examples of files that fall under this category are the Microsoft Office (97 - 2003) file formats: doc, xls, and ppt to mention a few. The Microsoft Office files are of special interest since this is a binary format designed for containing text data and are very commonly used. The files use a technology called OLE2 (Object Linking and Embedding) which itself is built upon Compound files [25]. Both of these structures is developed internally by Microsoft and were previously closed standards. However the specifications were released in public on June 30, 2008 [26]. The structure in these files can be compared to a miniature file system containing two basic elements, storages and streams which correspond to directories and files respectively. A storage object may contain other storage objects or streams while a stream is a container for data only. The entire file could be viewed upon as one large array. The storage object contains a collection of pointers to the contained streams and storages. A stream is a sequence of elements each ending with a pointer to the next element in the stream as illustrated in Figure 11.

Consider a PowerPoint document which file type container include five streams [27]:

1. *Current User* Contains information about which user last accessed the document

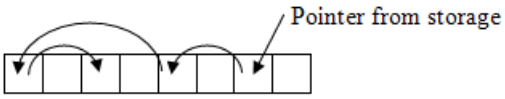


Figure 11: An illustration of the relationship between elements in a stream for a MS Office document. At the end of each element a pointer is used to indicate the start of the next element.

2. *PowerPoint Document* Contains the text and document layout properties
3. *Pictures* Data in the form of embedded objects
4. *Summary Information* A set of metadata elements describing the document
5. *DocumentSummaryInformation* An optional extra information about the document

While Summary Information, DocumentSummaryInformation and Current User contain metadata that might be of interest to an indexing system, it requires a relatively high amount of effort to locate and extract. The buildings blocks of a stream is quite complex but can, just like the document type itself, be viewed upon as a miniature file system. It contains a number of *container* objects which in turn host objects called *atoms*. The atom-objects can be of many different types and can contain information about texts, layout, animations etc. The resemblance between other file type formats in the Microsoft Office family prior to 2007 is very closely interrelated and offer similar characteristics. The intuitive way to gather the text content from such a file is to find the atoms that are defined as containing text and extract their content. This would of course presume that an interpreter with knowledge of the binary format is being used.

Formatted text files are files which commonly are human-readable text surrounded by tags or object descriptors used to define attributes such as fonts, weight, style etc. The files can be viewed in any text editor although it may be impractical to read them this way. This type of files can be compared to the source code in a programming language, in the sense that they provide code to be compiled by the viewer in order to view it correctly. A document type which conforms to the metrics of formatted text is the Portable Document Format (PDF) [28]; although the text components in PDF are usually stored as hexadecimal form or embedded into a type of stream concept which can accommodate text in many formats.

The source code of a PDF-document consists of a number of objects which can be of 8 types: booleans, numbers, strings, names, arrays, dictionaries, streams and the null object. The object types most relevant to text

extraction are the string and stream objects. It should be noted that some objects can contain nested objects; hence an array- or dictionary object still has to be examined for relevant textual content. Strings in the PDF source code are written in literal form as (text) or <text> in hexadecimal form. The simplest form of text gathering would consist of collecting these areas. However this would ignore the actual order the text is presented, as text component need not necessarily be written in the order they appear in the document. For completeness, it would also require that any textual content are extracted from the mentioned stream containers.

Objects in the PDF format are defined as $10obj$, $20obj$ and so on and can later be referenced to as $10R$ and $20R$ respectively. The main structure of a PDF document starts with a Catalog object. The object in turn contains a Pages object which is a collection of Page objects. Each of these Page objects have a variable called contents which is a collection of content-holding objects such as strings. Below is an example of a very simple PDF document structure with one page containing one string. Note that the code has been stripped to provide only the essential information for text representation and cannot be considered complete in the sense that it's not interpretable by a PDF viewer.

```
1 0 obj
        << /Type /Catalog
        /Pages 2 0 R>>
endobj
2 0 obj
        <</Type /Pages
        /Kids [ 3 0 R ]>>
endobj
4 0 obj
        << /Type /Page
        /Contents 5 0 R >>
endobj
5 0 obj
        (Hello World)
Endobj
```

The '«»' notation represents the Dictionary object and is a method of associating an object with a key. The other notation '[]' is the Array structure which is an ordered collection of objects. With this level of knowledge is should be possible to interpret the source code and gather the document text components in the correct order, presuming that all text content is stored as human readable format. It is however much more common to save the text content as streams.

```
4 0 obj
<</Filter/FlateDecode/Length 1726>>
        stream
                xWŢXÛÕÛF}_‘ßĄŔRQÏê2žA rE [...]
        endstream
endobj
```

In order to convert the stream-content to text the flag '/Filter' must be present which specifies what filters have been applied to the stream. The filters need to be applied in reverse order to extract the actual content of the stream. There are a number of standard filters defined in the PDF reference manual, and the specification of these must be known in order to interpret PDF streams objects. The exact specification of each filter is however beyond the scope of this summary.

XML structured text is text formatted using a specific structure. XML stands for eXtensible Markup Language and is a specification to create custom data storage languages. The syntax of the language is very simple, comprising of one start element with a set of nested sub tags. An element can be extended to create elements suitable to the situation. The basic element structure can be outlined like: *<name attribute="attr"> data </name>*, or in the more verbose form of *<name><attribute>attr</attribute><value> data </value></name>*. Hence the data content can be either the text itself or an arbitrary structure of nested tags. By using custom elements to build e.g. a tree-structure, the language can be extended to handle complicated data structures. Consider the following example where a basic library system is represented by XML notation.

```
<Library>
    <Book language="english">
        <Title>Building Search Applications</Title>
        <Author>M. Konchady</Author>
    </Book>
    <Book language="english">
        <Title>Distributed Systems: concept and Design</Title>
        <Author>G. Couloris, J. Dollimore, T. Kindenberg</Author>
    </Book>
     [...]
</Library>
```

XML can also be used to create structures very similar to formatted text where the text content is the data surrounded by xml-tags with attributes such as bold text or a custom font. An example of a file type employing this methodology is the Microsoft Office Open XML suite. Instead of creating a rich data structure with a very high level of complexity, the approach has been to split information into separate XML files. These files are compressed

into a single ZIP archive with a specific folder structure. For example, a docx file includes a folder named *docProps* which stores XML files conveying document metadata. Most of the contained files specify styles and other settings, but there is also a file named document.xml which outlines the general document structure by an XML representation. The document is structured so that text content is always surrounded by a *<w:t></w:t>* tag. This is very useful fact as it makes the text-extraction process next to trivial. Similar metrics can be observed for the other file types in the OOXML suite as well.

2.6 Indexing server

An indexer is one of the three major parts of a search engine, the other two being the feeder and the search front end. The file feeder is the mechanism that provides the indexer with files for indexing and the search front end is a user interface which supplies the search service to the user.
The purpose of the indexer is to generate what is known as an *index* , which can be defined as creating structured information from unstructured text. The exact nature of an index depends on its implementation, but more generally it can be said that it's the storage of document information optimized for fast and accurate lookup. The information in the index is utilized by the search engine to find matching documents on the users request. The search engine strives to give the user the most relevant document by searching for documents matching a set of given keywords, combined with factors such as ranking or popularity ratings. The purpose of the index is to simplify and speed up the search process by supplying an organized structure that can speedily be scanned instead of iterating the contents of a large set of documents directly.

Before the index is generated the text are organized into a set of weighted *terms*. The common approach is to locate reoccurring words and phrases. Certain words that are interpreted as having a special meaning can be assigned higher priority; for example names or email addresses. The structuring of the text into a set of relevant words and phrases is called tokenizing. Certain document types contain data supplied only for the only purpose of providing indexers with predefined relevant words. This information is called *metadata*.

Some of the first indexers and search engines were devised to operate solely based on the metadata while ignoring the rest of the document. For example, metadata tags were included in the HTML-format specifically for the purpose of accommodating indexing services [29] . Although the technique using predefined metadata tags for indexing is straightforward, it proves fallacious in the sense that it's easily abused. A major impact is how HTML documents could be designed to appear on top of many search results by clever engineering standpoints such as keyword spamming. As a result search

engine ranking system has become increasingly intelligent over the years, in some cases, including several hundredths aspects in each decision. In internal systems however, this flaw might be of less relevance as metadata can be deterministically generated from the content of a document itself.

There is a multitude of possibilities to rank the set of *terms* for each document in order to construct a highly searchable index. One of the more common approaches is based on the generation of a list of all words in each document combined with their frequency. The selection of words should exclude the most common words such as *a, and, of, the, is* etc., as they usually pose no value to the search query. To exclude irrelevant terms an interval of terms somewhere in the middle to high- range in the frequency list is selected. Each term in the set are given a value representing its frequency in the document, where higher values implies more frequently occurring terms. The method of ranking words based on frequency is called *Term Frequency* (TF) .

The outcome of applying the Term Frequency technique alone tends to inflict poor results since ignoring low frequency words reduces precision for queries regarding uncommon words and ignoring high frequency words potentially discards expressions of relevance. To create a better balanced output, the term frequency can be combined with *Inverse Document Frequency* (IDF): which is the inverse of the number of documents that the word occurs in. Konchady [30] suggests using a formula to deterministically calculate the IDF:

$$IDF = log\left(\frac{N}{n}\right) + 1$$

where N is the total number of documents in the set and n is the number of documents where the word occurs in. Each document is associated a weight for each term based on for the product of TF for the document and the IDF for the set.

$$W_{i,j,k} = TF_{i,j} * IDF_{i,k}$$
$$i = term, j = document, k = documentset$$

The method is commonly known as *TF/IDF indexing*. There exist also other methods for generating indexes worth mentioning; among these are *Latent Semantic Indexing* (LSI) which is based on building matrixes of terms, documents and relations between words in similar contexts [31], and *Okapi BM25* which utilizes probabilistic models [32].

The generated weights ($W_{i,j,k}$) are used when calculating the relevance of a document for a specific user query. The most conspicuous method commonly used together with TF/IDF, is called the *vector space model* [33]. Let each queried term be a dimension in a graph and draw a vector from

origin to the calculated point for each document. Let the length of each vector denote the relevance for the corresponding document. The document that receives the highest relevance value is considered as the most suitable answer for the query. Figure 12 illustrates a query for the words *relevant* and *document*. We observe that since the relevance value of 6.3 for document B is more prominent than the relevance value of 5.09 for document A, document B would be the best suited answer to the query.

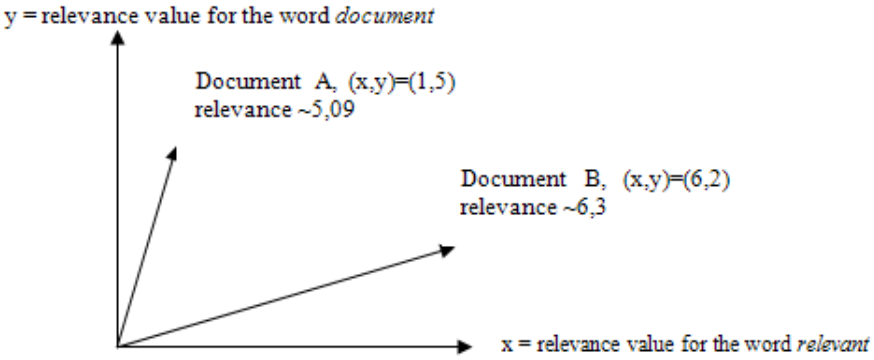


Figure 12: A query for the words *relevant* and *document* in a set consisting of two documents.

3 Method

During the analysis, a number of principles have been addressed to appoint general distributed techniques to the current problem domain. At this point the reader is assumed to have an understanding of which logical entities of the indexing service this thesis is focused on. We will continue to study the realization of the distributed system by careful examination of each logical subsystem; starting with file crawling and ending with the handover procedure to the indexing server. All aspects of the system is implemented using the Java programming language and is thereby operating system independent.

To aid the understanding of the system in general, an illustration is devised to depict the key components of the system as can be observed in Figure 13. It's produced in a custom format intended to give a brief overview of the system without including the full set of entities and their complex relationships. Note that the arrows represent part of the program flow rather than dependencies. We let this illustration guide each of the following subsections.

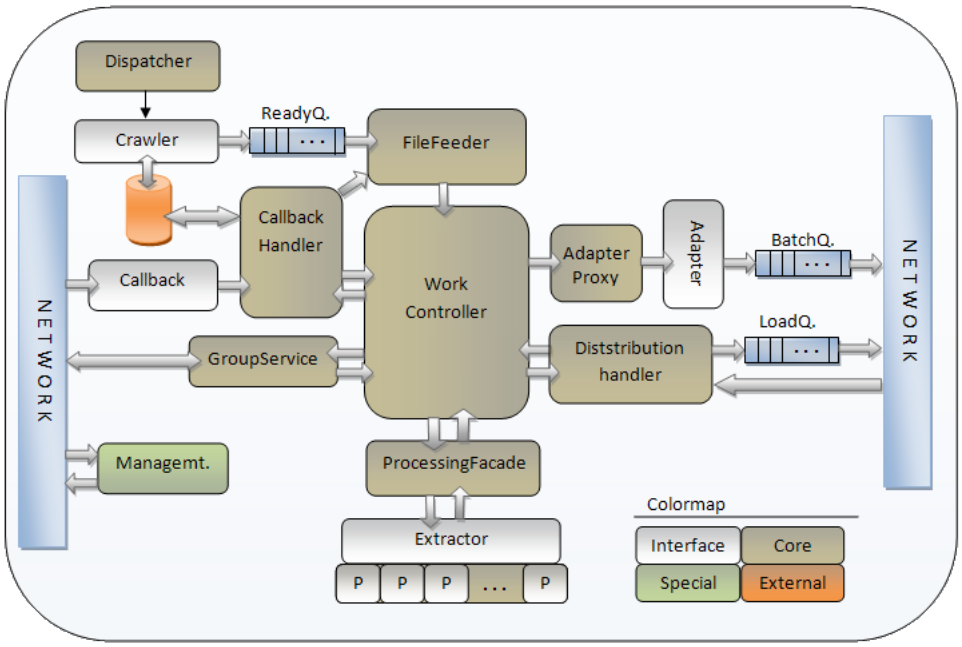


Figure 13: A conceptual view of the developed system. The illustration is largely simplified where only key flow components are outlined.

3.1 System summary

It's a difficult task to trivialize a complex system. Still, we feel that a short summary would aid the illustration of Figure 13. The starting point of the system is the dispatcher which on a regular basis schedules new crawler jobs. The crawler iterates the local file system and passes the files by a shared queue to the file feeder which in turn forwards the file to the work controller. Assume for completeness a configuration where all modules are active. The controller contains the plumbing to route the file to different locations based on a large number of factors. First, the processing facade is contacted which performs the information extraction and processing of all files. Once complete, the file can either be forwarded to the distribution handler or an adapter based on decisions from e.g. the group membership service module. If passed to the distribution handler, the contents of the file are shared with other members in the group which collaborates in performing additional computations. The alternative is to forward the file to the adapter proxy which provides a uniform interface to accessing an indexing server as well as a batching routine. The proxy in turn controls an adapter which is the direct interface to the currently used index server.

Once a file has been pushed onto the network, a callback will be returned and caught by the callback handler. The contents are examined and a decision is made to either start the re-processing mechanism upon failure, or to mark the file as indexed in the database. Further, there is management component which is intended to provide remote control of the application and to facilitate mechanisms to generate e.g. remote logging events.

3.2 Crawling

The term crawling is used interchangeably with traversing, and is considered in the context of iterating files in the local file system. We are only concerned with file crawling, although there are other potential applications like web- or email crawling. In the end each document or element is inherently a file, and as such a file crawler is sufficient to locate this information.

There are two approaches to efficiently crawl a file system. The process can be performed *iteratively* or *recursively*. The latter can be considered more intuitive to implement, however it also impose the risk of stack overflow during deep recursion. Since a file system can be of arbitrary depth in terms of directories, and each directory can contain an arbitrary amount of files, the risk of overflow is abundant. Thereby an iterative approach to traversing the file system is chosen as the primary method.

The crawling can be controlled by a number of configurable options supplied in advance of a system execution. The relevant options are: *publicDirs*, *localStartdirs*, *skipDirs*, *skipFiles* and *readyQueueSize*, where the former are comma-separated list items. The variable publicDirs is used to select the

entry point(s) to the file system, allowing the user to hand-pick the directories to crawl. The second variable localStartDirs is intended to be used in conjunction with publicDirs to control how files should be exposed externally; i.e. to be accessible to a user upon querying the search front-end. This enables the freedom to select whether files should be accessible by e.g. HTTP-requests, FTP transfers or network file sharing. The relationship is best explained by observing Table 1. The two properties skipDirs and skipFiles are used to exclude files and directories from the file traversal. Both are powered by regular-expressions, which is useful to ignore files based on their pattern. Consider the usefulness of skipping files such as *.jpg, Thumbs.db or [Pp]rivate*. Once a file has been captured by the crawler, it's forwarded onto a queue instance shared with the rest of the system. The property readyQueueSize defines the size of the queue and thus limits the number of files crawled at a time. It acts as a guarantee to not crawl more files than the system can serve in any time frame.

Table 1: Mapping of resources from local to public access.

LocalDir	File	PublicDir	Resulting location
C:/path1/	C:/path1/a.txt	http://host/foo	http://host/foo/a.txt
C:/path2/	C:/path1/b.txt	ftp://host/foo	ftp://host/foo/b.txt
C:/d1/d2	C:/d1/d2/d3/a.txt	file://host/foo	file://host/foo/d3/a.txt
C:/d1/d2	C:/d1/d2/d3/b.txt	file://host/foo	file://host/foo/d3/b.txt

The crawler is essentially its own system executed by a dedicated thread. It's instantiated regularly by the dispatcher based on predetermined input of the user. The scheduling employs the UNIX familiar Cron expressions [34, 35] to launch the crawler with a high level of configurability. It's internally configured to not issue multiple crawling instances simultaneously. Instead missed events will up to a certain point become scheduled in a sequence following the termination of the currently violating job.

A desirable property is to obtain a *stateful* notion of the file system which corresponds to maintaining state information for each crawled file. The records can be used to track which files have already been submitted to the index server and thus requires no additional action. The current implementation is backed by a SQLite [36] database storing records of file location, indexed status and a MD5 digest. The digest algorithm spans the size of a file combined with last modified timestamp, thereby producing an effective method to discover changes to a file. The database records enable the discovery of missing files, inclusion of files, and the modification of files. A mapping is created to generate corresponding instructions to the indexing server. Hence,

the crawler is able to intelligently distinguish state records to only inform the indexing server of changes to a document when necessary. The technique is based on the assertion that database records are not persisted until a response has been received from the index server. This notion of robustness has the direct implication that a server failure will not result in incorrect file records or missed documents in subsequent crawls.

3.3 Internal file routing

The crawler exposes files to the rest of the system by an internal routing mechanism. The mechanism resides in the work controller as depicted in Figure 13, and is activated whenever the file feeder forwards file objects to it. The routing mechanism is layered into three tiers based on if the user has enabled the network synchronization or load sharing mechanisms. The choice of active modules should reflect the needs of the user, e.g. if there only exist one file server there is no need to enable group services or load distribution. However the highest level of complexity arises when both options are enabled. In this scenario the controller needs to collect advice from the group service in order to decide where to route the file. It may be instructed to fill up a queue of files to pass to the load distribution mechanism, or it may be temporally suspended until it can access the network, or the file can be enqueued to be a part of a batch routine intended for communicating files to the index server.

One of the main responsibilities of the controller is to synchronize the access for more than a handful of threads at worst. Internally this is implemented by a series of java monitors. Some of the threads which require synchronization are the feeder, the re-processing mechanism, membership services, the distribution mechanism etc.

3.4 Information extraction

As defined in the analysis, information extraction is the task of extracting relevant information and storing it in some internal data structure. Since the system should be able to facilitate all document types the data structure comprises of a modular design which enables associating any type of document to it. The system currently supports some of the more commonly found formats, namely pdf, doc, docx, xls, xlsx, ppt, pptx, html, txt and images. Each file type is associated with a dedicated *extractor* implementation. An extractor is based on a well-defined interface providing a uniform data-retrieval routine. File types with no pre-defined extractor are subject to a default extractor providing generic routines to extract metadata from a file. The default extractor offers two modes of operation based on if the index server supports receiving files as binary content. If enabled, the entire file is stored in KB-sized chunks in a buffer and transmitted to the index

server. Otherwise only the metadata component is used to represent the file.

The interface for creating a new extractor poses no limitations to what types of metadata can be stored. Each element is represented by a *<key, value>* pair. Before passing the metadata elements to the indexing server they are subject to another mapping, allowing the administrator to configure exactly which name each element should be stored by. Using this highly configurable approach, adding or removing metadata elements brings no further implications than adding or removing a line of text in the mappings document. Moreover, the extractor configuration itself is largely flexible. We've developed a framework where a file type can be mapped to an extractor using an external XML configuration file. The extractor implementation can be provided either as a java class or as a java jar file, in the sense of build once use anywhere. A syntactic excerpt outlining this procedure is given by the following example. Note that there must exist a HTMLExtractor.class or a HTMLExtractor.jar in the predefined class path.

```
<extractorconfiguration>
    <extractor>
        <filetype>html</filetype>
                <filetype>htm</filetype>
                <filetype>jsp</filetype>
        <class>HTMLExtractor</class>
   </extractor>
    [...]
</extractorconfiguration>
```

A restrictive approach to memory utilization is an integral part of the system design philosophy, and the extractor interface is no different. Two quantifiers are supplied with the interface to control contrasting aspects of the memory utilization in terms of data size for the working set as well as the maximum data size for each file. Although the restrictions are provided, the implementer is not obligated to enforce them. The entire system is constructed with Postel's law in mind: "*Be conservative in what you do; be liberal in what you accept from others*" [37]. The law is mirrored in our design philosophy; the system provides a strict rules set, however if a module disobeys one or more of these rules, the system shall still retain operation as close as possible to the intended specification.

The text extraction process is for some file types an operation of high complexity. Where due, we've decided to not reinvent the wheel again. Hence a number of open source libraries have been used to aid the text extraction for some of the more complex file format specifications. As an example the open source PDFBox [38], under the Apache License v2 [39], is used to interface with pdf files. Further Apache POI [40], also released under the same license, is used as a façade for files in the MS Office suit based on the OLE2 standard. It is of course also possible to implement extractors without the use of an external library.

The OOXML based MS Office 2007 series is an excellent example of how the java standard library is sufficient to perform the text extraction. The format is more closely examined in the previous analysis section; however we'll formulate a summary of the necessary steps to perform the text extraction:

1. Extract the ZIP archive
2. Locate files with text contents, i.e. word/document.xml for docx
3. Parse the selected file for areas surrounded by *<w:t></w:t>* tags
4. Return the extracted areas in the same order of occurrence as in the file

There are also other files of relevance. For instance to locate the title metadata element in a docx document, the file docProps/core.xml can be scanned for the *<dc:title></dc:title>* tag.

3.5 Information processing

After the textual information is extracted from a document it may be desirable to perform additional processing on the contents. Processing steps are based on receiving text and metadata in an internal data structure, create or alter some of its contents and return it to the caller. They are characteristically computational intensive and may perform actions such as character encoding conversion, sorting, or directly act as a replacement for any of the possible steps traditionally performed at the indexing server; such as spellchecking, stemming, ranking or categorization.

Analogous to the extractor configuration, processing steps as well supports dynamic invocation using an xml configuration file. The usefulness appears in that the application can be tailored to the needs of each individual customer while maximizing component reuse. The configuration is however a little more elaborate this time. The order in which processing steps appear in does make a difference. Secondly, processors are categorized based on if their operation must be performed locally or if they can be distributed to another server in the group. Consider the following configuration:

```
<pipelineconfiguration>
    <pipeline>
        <filetype>doc</filetype>
                <filetype>docx</filetype>
        <mstep>ACLProcessor</mstep>
        <dstep>UTF8Processor</dstep>
        <dstep>WordCountProcessor</dstep>
   </pipeline>
    [...]
</pipelineconfiguration>
```

The pipeline step associates files with the doc or docx extension to three processors. The ACLProcessor, which extracts ACL information from a file,

is defined as a *mandatory* step and must be performed locally. There are mainly two motivations to define a mandatory step. Either the processor needs direct access to the file itself, not just its previously extracted contents, or the step requires very little processing time and would thereby not benefit from being distributed. Performing character decoding and encoding is however a more demanding task and can be performed elsewhere, hence the UTF8Processor is defined as a *distributable* step. The distribution benefits increase as the number of distributable steps grows. It's been identified that in a real scenario, the number of distributable steps may be significant, as well as the associated computational demands. There's also a mechanism to assign default processing steps to all file types which has no explicit processor declaration. The routine responsible to perform the distribution is studied in more detail in the load sharing section.

3.6 Adapters

Whether a file is distributed or not it will eventually be forwarded to an adapter. The adapter is an implementation specifically intended to bridge the communication with the indexing server. Each index server thereby requires a custom adapter implementation to perform the necessary adaptation of data into a form interpretable by that server. An adapter implements an interface allowing it to be controlled by a proxy class. The proxy employs a batching routine which is useful for two reasons. It offers a more effective approach to propagate data onto the network and secondly it's used in conjunction with the network synchronization mechanism. That is, while waiting to access the network the batch is filled. Once accessed, the batch is emptied and its contents sent to the index server before passing on the network access token. The adapter proxy also contains routines to handle network failures. If the network becomes unavailable the proxy will enter a state where it's attempting to detect network access again. Once the network is available a routine will be invoked in the adapter which reestablishes connection to the index server again. Hence the act of unplugging the network cable and inserting it again will only cause a delay in the software.

Before passing the metadata for a file to the indexing server a mapping is performed to transform each entry into an acceptable format for that specific index sever. Each of the internal metadata elements requires a matching entry in the mapping table or they will be automatically discarded by the system.

The system supplies two adapter implementations bridging the communication between the application and indexing server. They will be considered in the following sections.

3.6.1 FAST ESP

FAST ESP [41] (Fast Search & Transfer Enterprise Search Platform) is a proprietary search platform for corporate search solutions. ESP supplies a complete search engine by supplying feeding capabilities, an indexer and a search front end. There are a number of different components for feeding the index server available but the one relevant to this thesis is the file traverser. It works by fetching files from content providers over a network connection and exposing these to the indexing server which performs data extraction, processing and indexing. The ESP index server accepts most file types directly using built in text extraction techniques. It's also possible to supply extracted text contents or metadata directly to the indexing server. ESP provides a java API for connecting via an object which accepts files in either binary form or as pairs of <key, value> elements representing a file. Using the latter approach, a mandatory key representing the file contents need to be supplied. Regardless of method, ESP utilizes a container object known as IDocument to store the contents of a file. The following code snippet gives a brief overview of its usage:

```
IDocument document = DocumentFactory.newDocument("id");
document.addElement(DocumentFactory.newString("data", "text example");
document.addElement(DocumentFactory.newString("author", "Sture")
[...]
```

The IDocument is upon completion forwarded to the indexing server via predefined API routines. ESP performs the indexing according to a predefined pipeline on the indexing server. The functionality to alter the pipeline is provided by a web interface. Once all files have been indexed, they will be subject to a routine responsible to make the file searchable through the search front end. When an object has been processed the indexer will return a callback-object specifying the result of the indexing. The callback contains a variable which denotes if the indexing was successful or not as well as a collection of possible errors and warnings. The callback object is adapted to a standardized internal representation of a callback object used within our system and forwarded to a callback handling routine capable of interpreting the results and taking appropriate action.

After a batch of data is sent the main adapter thread will block until all callbacks for that particular batch has been received. Disconnects and connection errors are reported as exceptions and can occur at any time either when sending or during the blocking operation. In the event of an exception the adapter will automatically generate failed callbacks for all files which no callback was received for prior to the exception. After the exception is handled locally the adapter will continuously try to re-establish the connection and resume operation.

3.6.2 Apache Solr

Apache Solr [42] is an open source enterprise search server based on the Apache Lucene search library. In contrast to ESP, which is backed by a major company offering a complete range of functionality, Solr offers just the basic. It provides an API to propagate formatted or unformatted text to the indexing server which performs the index operation by a set of predefined rules. An XML representation of the matching data is returned upon a search query. An excerpt of a search result is outlined below:

```
<doc>
    <str name="filename_s">pretty sunset.jpg</str>
    <str name="id">C:/share/pretty sunset.jpg</str>
    <str name="lastmodified_s">Wed Feb 11 15:51:10 CET 2009</str>
    <str name="link_s">ftp://host/pretty sunset.jpg</str>
    <int name="popularity">0</int>
    <str name="size_s">71189</str>
    <str name="sku">ftp://host/pretty sunset.jpg</str>
    <date name="timestamp">2009-04-23T14:23:27.489Z</date>
    <str name="title_t">pretty sunset.jpg</str>
    <str name="truncated_s">false</str>
    <str name="content_t">This is the textual content[...]</str>
</doc>
```

As obvious by now, Solr doesn't provide much usability for free. In order to prove useful a search front end needs to be constructed to wrap the XML data; which is not treated in this thesis. Most entries are custom created by appending a symbol after the name. For instance filename_s corresponds to an entry of the type string, title_t of type text etc., due to the control sequences '_s', '_t'.

The communication with Solr is in essence non-stateful. Transferring content to Solr is nothing more complicated than sending a HTTP request containing the data content. Thereby there is no active connection needing to be maintained which simplifies handling network outages. Solr differs from ESP in yet another aspect as it doesn't provide any callback mechanism. Data errors are instead masked in the process of locally translating the content into Solr XML format. Communication errors are notified as exceptions when issuing the HTTP request. The implication is that the Solr adapter itself creates virtual callbacks which are passed directly to the callback handler. The implementation specific challenge turns out to be to avoid deadlocks in the system. In contrast to ESP where the client spawn new threads as incoming callbacks are received, the Solr callbacks are generated by the thread which executes adapter code. Technically this thread resides in the file feeder, thereby offering quite a different approach to assigning thread responsibility.

3.7 Callback handling

Handling callbacks are performed by the callback handler denoted in Figure D. A generic structure is used to represent a callback internally. Each successful event will result in the file associated with the callback to be persisted in the database, which is the first time a file is considered to be indexed. Upon a failure, a state machine approach is initiated to reprocess the file. The first failed event will result in the re-extraction and reprocessing of the file in the same manner as was performed for the file the first time. During a second or third failure, two levels of base extractors will be used which succeedingly extracts more primitive data from the file. The receipt of yet another failed callback will result in the file being dropped from the current crawl operation entirely.

A dedicated thread is used to perform the re-extraction and reprocessing routines. The callback handler merely adds a failed callback to the scheduling mechanism after updating its internal state. After enough failed callbacks have been scheduled, or after a timeout, the actual mechanism will start which synchronizes the reprocessing operation with respect to the regular batch routine. The reason is that treating a failed callback is equivalent to using the framework already in place for processing and extraction, only with a control parameter used to decide which extractor configuration to employ for the current case.

3.8 Communication

When deciding the communication backbone technology for a cluster of file servers there are a number of available options. The first option, to write a custom implementation, is discarded due to the complexity of the task as it would likely make up an entire thesis by itself. Since there already exist a number of available middleware systems designed for distributed systems, one of these would likely offer a more befitting choice.

One option is to use the popular CORBA (Common Object Request Broker Architecture) [43] for communicating within a group of clients. CORBA supplies a wide variety of functions; most prominent are the ability of remote method invocation and programming language interoperability. Due to its generic object structure and communication mechanisms, it's an excellent choice to provide communication between e.g. a C- and java client regardless of operating system. There are however no built in group membership capabilities, alas the entire group service protocol would need to be devised on top of the CORBA stack.

Another toolkit is Ensemble [44] which offers satisfactory membership capabilities as well as a wide range of programming language interoperability. However the major disadvantage is that it must be run as an external service on every participating node. This is both its greatest strength and greatest

weakness depending on the situation. Combined with its academic nature, the usage of Ensemble is rejected as well in this project.

A third alternative is the Internet Communication Engine (ICE) [45] which closely resembles the CORBA architecture but with less complexity. Although Ice seems to provide all necessary functionality it has an open source license which doesn't permit it to be interleaved with proprietary software. A commercial license is offered as well, but this alternative is discarded in favor of an open source solution.

JGroups [46] is a toolkit specifically designed for reliable multicast and group handling. Its protocol stack is configured externally by an XML file which can easily be adapted to the situation. Total order protocols, with or without a coordinator, is supported. Due to its multifaceted operation, JGroups is chosen as the communication provider for group services in the system. To reduce network contention, a coordinator controlled approach to message delivery is selected. It's a variant of the reliable coordinator scheme presented in the analysis section 2.3.4. A node wanting to propagate a message sends a unicast transmission to the controller which sequences the incoming messages and performs the multicast on behalf of the peer. As the number of nodes increase, the controller will be faced by an increased amount of work. This is not an immediate problem in a server environment where not more than a dozen servers are considered to be part of the system at any one time.

Another advantage of JGroups is that it's considerably less complex to use than e.g. CORBA. All that's needed is to construct a class which implements an interface consisting of a few routines for sending and receiving messages. Any other routines, such as delivering a new group view or passing control messages are already taken care of by the respective layer in the defined protocol stack. Hence the implemented class can be compared to the application layer of the OSI model, commonly used to represent the different protocol layers a message is exposed to before being transmitted over the network.

3.8.1 Membership service

The purpose of group membership is to provide controlled communication between all nodes participating in the system. It is desirable to synchronize access to the network and to perform services like load sharing. Each of these implementations requires a node to maintain a few data structures which must be kept synchronized between all nodes in the group. A group service offering *ordered reliable multicast* can facilitate this need. The JGroups membership service is configured to be handled internally by the *sequencer* protocol which employs the concept of a coordinator to accomplish total ordering. The method for choosing a coordinator is simply to pick the node that appears first in the view, e.g. the node that created the group. A new

node joins the group by the following algorithm:

```
find existing members
    if no members found:
        create a singleton group and become coordinator
    else:
        determine the coordinator of the group
        send JOIN request to coordinator
        wait for JOIN response
        if JOIN response received:
            node has joined the group
        else:
            wait for 5 seconds and restart
```

Each node receives a new view object each time the group is updated. The object is delivered to the implementing class which compares the new view to the last view received to detect changes to the group. Detecting group changes is necessary to maintain a correct view of the distributed data structures. The nature of these data structures will be considered in the network synchronization and load sharing chapters respectively.

In the special case were there exist two or more coordinators, each in a subgroup, the coordinators need resolve this as soon as the other group is discovered; leading to a merge of the subgroups. The details of this action are at the discretion of the JGroups implementation which performs this service internally.

3.8.2 Network synchronization

In addition to maintaining a list of all nodes every member of the group stores its own copy of a global request queue. The queue is used to control network access to nodes by granting access only to the node at the head of the queue according to the global queue time division described in section 2.2.3. Somewhat confusing, the network access mechanism is used to control access to the indexing server and not the network per say. When a node desires to forward all processed files from the batch queue to the index server, a *request* multicast message is issued which appends the request at the tail of the queue. The node which request is at the head of the queue is currently eligible to communicate with the index server. Once the operation is complete the node revokes its own access by issuing a *release* message, allowing the next node in the queue to access the network.

When a new node connects to the group it receives a copy of the queue from the coordinator. From that point on, the queue is kept synchronized across the group by the mechanism provided by the totally ordered multicast. In the event of a node failure, or if a node leaves the group, a routine will

be performed on each remaining node in the group to remove the faulting node from any data structures. The routine relates to the view delivery mechanism mentioned in the previous section. If a node is no longer found in the current group upon a view delivery, and the node is present in e.g. the network access queue, it can safely be removed.

3.8.3 Load distribution

In the Information Extraction section the term distributable processing steps are defined. During an execution, the contents of a file are first extracted before the mandatory processing steps are performed. At this point a *load queue* is used to store all elements which have not yet had its distributable steps processed. Once the load queue reaches its capacity ceiling a load distribution request can commence.

The first step requires the node to locate an available peer. The technique used to perform this lookup is integrated into the implementation of the membership service. This time a distributed hash table is used to track the status of each member in the group. A status entry is defined to carry either the value *free* or *busy*. The value free corresponds to that a node is currently not performing a crawl or processing job of any kind, while busy relates to the opposite. Each node that finds itself in a position where it is able to perform work for another node sends out a multicast message notifying that it is free to the group. The status entries are kept synchronized by dedicated *changeStatus* multicast messages.

Since the set of available peers are known, a distribution request is forwarded to one of the peers according to the source-initiated distribution technique visited in the analysis. The originator collects the response and accepts it upon a positive reply. If an answer is negative or missing, the node will continue to listen to changeStatus messages until another node is marked as free and the request procedure can be repeated. Once a peer is verified to perform the distributable steps, the contents of the load queue will be forwarded to that peer and the node can continue processing elements locally as well as preparing to perform another distribution batch. The details of the distribution protocol are illustrated in Table 2. Note that distribution mechanism corresponds to the technique visualized in Figure 10.

Sockets are selected as the primary means of distributing workload. Although the membership service supports unicast messaging, there are arguably less overhead associated with socket communication. Lastly, if the connection should fail in any way the distributor resumes the responsibility for all the files that it did not receive any callback for.

Table 2: The file distribution procedure as viewed by the initiator.

Distributor	Peer
(1) Open socket on specified port	
	(2) Accept connection
(3) Send specified amount of data followed by end-of-queue	
	(4) Receive all data
	(5) Start processing of data and forward the results to the index server
	(6) For each callback from the index server, send callback to distributor
(7) Receive callbacks	
	(8) When all files are processed and all callbacks from the index server are forwarded, send end-of-queue
(9) Close connection	
	(10) Close connection

3.9 Security

The system is foremost intended to be used in a local area network where security implications are not as severe as were there an intermediary internet connection between the servers. Nevertheless the ambition regarding security is to implement the highest level possible for each communication component throughout the system. By the term *security* we mean security in the aspect of avoiding eavesdropping (confidentiality) and unauthorized modification (integrity) of the network traffic. The communication system can be divided into three categories regarding security: *internal data transfer*, *group communication* and *index server specific communication* where each category has its own security implementation.

The internal data transfer communication, or *socket based communication*, is the traffic generated by distributing data between nodes. It has the option to enable SSL/TLS encryption by the option *Security.ssl* in the systems primary configuration file. Since SSL/TLS employ the concept of asymmetrical keys, every node needs to have its own private key and a copy of every other nodes public key. By using SSL/TSL, the communication is considered secure in the aspect of both confidentiality and integrity; translating to that only an authorized node is able to read and write data while being aware of any modifications to the original contents. However the procedure of generating and distributing all keys might be a tedious task and in order to simplify the process we have created a tool that generates keys and helps import the public keys on every node. For group communication there is the option to use symmetrical encryption. In order to enable encryption

in the system the configuration variable *Security.groupComm* can be set to true.

Since the encryption is symmetrical an identical key must be present at every partaking node. Having symmetrical encryption means that every node in the group can read every message, but this is not regarded as any weakness since multicast is used as the basis of operation for the group communication. The communication with the index server is naturally dependent on the current choice of index server, leaving the security issue to the implementer of the specific adapter. Certain index servers such as Solr provide no security functionality while FAST ESP is fully capable of communicating over SSL/TLS. Implementation of security over these links is beyond the scope of this thesis.

4 Results

As the system is intended to replace existing crawler implementations, a comparative benchmark analysis is useful to interpret the system efficiency. Of the two indexing systems examined in the thesis, only one provides crawler software. The FAST ESP system supplies proprietary software named File Traverser, denoted simply as *ESP* in the following tests. We refer to our implementation as *DDP*, short for Distributed Document Processing. The DDP crawler is intended to be run on each content provider in the network, while the ESP traverser is always performing its execution on the indexing server itself. The data set selected to be used for crawling are chosen as a pseudo-random collection of documents somewhat representable to a real world scenario. All documents have been retrieved from Google by using context search. For example the query *filetype:docx* returns a number of documents in the docx-format. The number of files, their file size and distribution can be observed in table 3.

Table 3: Ratio of documents used to produce the collection of test data.

File type	Num. files	Avg. size (KB)	Total size (MB)
Doc	85	156	13
docx	73	301	21.6
html	20	19	0.38
jpg	21	2004	4.19
pdf	40	7498	289
ppt	120	2176	255
pptx	44	1198	51.5
txt	16	266	4.17
xls	41	191	7.65
xlsx	23	74	1.68
Total	484	1375	650

Our goal is to perform tests which can be used as a representative means to compare key aspects of the system. Different aspects hold varying importance depending on the deployment scenario. The two we consider of greatest importance are the document throughput and network bandwidth. We use the total time to index the entire collection of files as a measurement of the document throughput. Network bandwidth is in both systems measured at the amount of network traffic which passes the network interface card located at the indexing server.

The tests are conducted in a local area network using up to five computers, where four computers are available to perform crawling and the fifth

being the Fast ESP indexing server. All communications are performed by cable offering speeds of 100 Mbps with a 1000 Mbps enabled switch. The tests take place in an office environment where traffic from other components than the test system is negligible. During the testing, all non-essential software is disabled to provide reliable results during the replicated test runs. Furthermore, both ESP and DDP are configured to not utilize any security modules, such as SSL, throughout the test runs. The computer specifications as well as the architecture and operations systems are specified in table 4.

Both the ESP- and DDP-system has the requirement that all files in the collection should be indexed; no files are allowed to be omitted to provide for comparable results. During all test runs, the DDP system was restricted to heap space of a maximum of 500 MB. ESP had no configurable counterpart and is left unknown. Each test is performed a number of times to achieve reliable results. As the system relies on an operating system to run, a large number of tests are required to observe the accuracy of the test runs. Although events such as operating system context switches never can be controlled, we expect that the impact will be of less importance to the crawling operation which extends over a large enough period of time. The test will be performed with varying number of participants and configurations and will be treated by a sub section each.

Table 4: The configuration of the test servers.

Role	CPU family	CPU freq.	Memory	OS (32-bit)
File server 1	Intel Pentium Dual	2 GHz	3 GB	Win. Vista Home Basic
File server 2	Intel Core 2 Duo	1,66 GHz	1 GB	Windows XP Pro
File server 3	Intel Pentium M.	1,86 GHz	1 GB	Archlinux
File server 4	AMD Turion 64 X2 Mobile	1,81 GHz	2 GB	Windows XP Pro
Index server	Intel Pentium 4	3 GHz	4 GB	Windows Server 2003

4.1 Testing with one file server

Both systems have been tested independently of each other a number of times using different configuration settings. The purpose of these tests has been to derive a configuration optimal to each of the systems for the current data set. Although the tests were conducted only on file server 1, we consider

the configuration settings to be satisfactory for any of the other file server as well. The resulting values will be used on all servers when conducting the tests, and they can be observed for ESP and DDP respectively in tables 5 and 6. Note in the DDP system that since the test runs are performed by a single content provider only, both network synchronization and load distribution are disabled. The ESP traverser is set up to perform its operation sequentially, as running two copies of the program simultaneously resulted poor results for the index server which was not equipped with multiprocessor technology.

The tables 5 and 6 also outline the resulting indexing time and bandwidth usage for the derived configuration. The most significant observation is the conservation of bandwidth by the DDP system of approximately 60 times, or 1.65% of the total traffic generated by ESP. In addition, the total execution time for the ESP file crawler is on average 1.42 times longer than the DDP execution time, by the added length of 1 minute and 20 seconds. It should be noted that the results are specific to the chosen data set and may well provide different results for different collections of data. The DDP crawler endured an average CPU load of 40-60% during the execution with a memory usage of approximately 230-240 MB. Since the ESP traverser is running on the same server which performs the indexing, we have not deduced a reliable metric for the CPU load which can be used for comparison.

Table 5: ESP configuration and test results using one file server.

Variable	Value
queue	100 MB
batch length	300
batchSize	75 MB
max fileSize	5000 KB
Time for indexing	4:31 Min
Network traffic	676 MB

Table 6: DDP configuration and test results using one file server. Note, the first three elements should be prefixed with 'fileCrawler.'.

Variable	Value
readyQueueSize	200
maxDataSizeMB	50 MB
maxFileSizeMB	50 MB
BatchQueue.size	200
BatchQueue.sizeMB	200 MB
Time for indexing	3:11 Min
Network traffic	11.16 MB

4.2 Testing with multiple file servers

The configuration used for each of the file servers is the same configuration which was found close to optimal during the tests with a single server. Tests performed with additional content providers are configured using file server setups {1,2}, {1,2,3} and {1,2,3,4}. The requirements are the same as the case with one file server; all files must be indexed at the end of the run by

all participating nodes. DDP is specifically designed to require the network synchronization module when multiple servers coexist on the same network, hence this module will be active throughout the test runs where more than one server is utilized. A test was also conducted using two file servers simultaneously without network synchronization. The result in terms of indexing time was as expected worse since performing several tasks at the same time is usually inefficient on uniprocessor systems. The load sharing functionality was tested to the extent that it fulfilled its purpose in routing the data and performing processing steps on another file server. There is however no relevant gains to be determined by enabling load distribution as there currently are no distributable processing steps which compares to functionality of the ESP implementation. Hence, any load sharing tests are deferred from the scope of this thesis.

Estimation of indexing time. The total indexing time for the FAST traverser is roughly equivalent to the indexing time of one content provider multiplied by the total number of content providers in the system. This is intuitively realized as the basic traverser implementation is only concerned with retrieving files as a whole and forwarding them directly to the indexing server. Fluctuations in the execution time of the FAST system originate mainly from network contention, disk speed and the underlying protocol used for file retrieval. The network propagation time is based on the size of the entire collection, as each file is transferred over the network as a whole. We define the following formula to derive an estimate of the indexing time for the ESP system using n content providers:

$$t_{FAST} = \sum_{i=1}^{n} [T_i + \alpha * \beta]$$

T_i denotes the time to required for the index server to index the data set located at node i, α is the time required to send one byte on the network (assuming negligible packet overhead) and β is the total size in bytes for the data collection.

Since the DDP system performs work at each content provider before passing each file to the indexing server, it becomes a little more complex to estimate the time of a test run. Hardware plays a larger role as the total time taken for the run is depending on the CPU capabilities, memory assets, disk speed and network capacity. The content provider which is the slowest at performing its extraction will define the base time T, where an additional T_i' will be added for each other content provider in the system. As opposed to the ESP traverser, the network propagation time will be based on the size of the extracted data instead of entire files. We derive for the DDP system the upper bound to perform the total indexing and crawler operation by the formula:

$$\hat{t}_{DDP} \leq max\left\{T_1 ... T_n\right\} + \sum_{i=1}^{n} \left[T_i' + \alpha * \beta_i\right]$$

T_i is the total extraction time for file server i, T_i' is the time it takes for the indexing server to index the data set at node i, α is the time for sending one byte on the network and β_i is the total size in bytes for the data set at node i.

4.3 Test results

Using the defined configuration, the result in terms of total time taken to perform the indexing is illustrated in Figure 14. It's clear that the total efficiency for the DDP system increase drastically as the numbers of content providers are introduced in the system. More interesting is the bandwidth ratio observed in Figure 15. Although the bandwidth requirement for the DDP crawler appears to be constant, it is only the effect of major differences presented by the two systems. Since load distribution is disabled, the traffic between the file servers is considered negligible. Note also that both curves are close to linear. This is natural since in the test case, adding more file servers' acts as a linear increase of identical collections. Further, the average amount of extracted text compared to the entire file size is illustrated for each document type of the test collection in Figure 16.

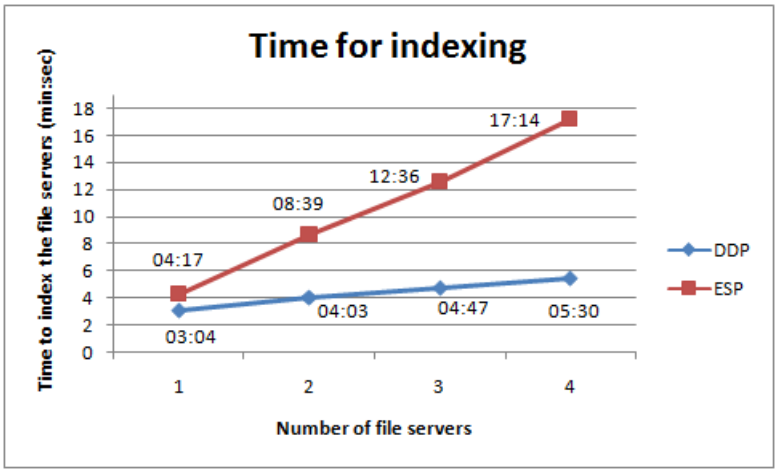


Figure 14: Statistics comparing the indexing times for ESP and DDP for the given data set.

The values obtained for the tests are the average values obtained over several test runs. These are statistically verified using the measure of standard deviation σ. Using four servers with an average sample mean of $\mu = 5.30$ minutes gives the standard deviation $\sigma = 17.47$ seconds over 10 test runs for

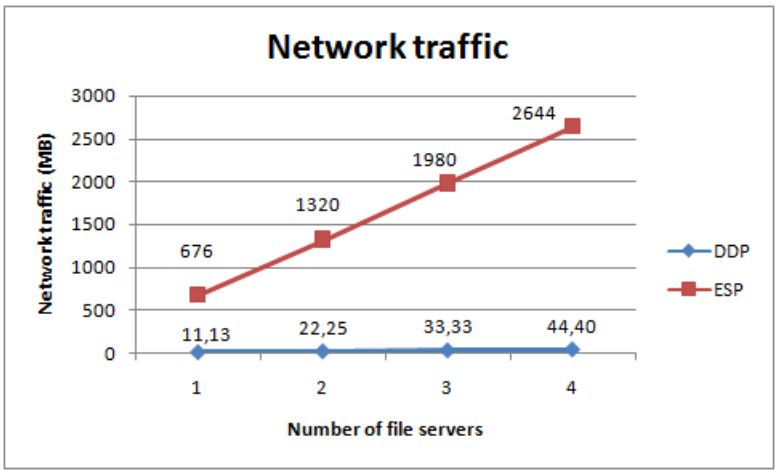


Figure 15: Statistics comparing the network traffic generated by FAST and DDP for the given data set.

the DDP system. The corresponding values for ESP using four servers are $\mu = 17.14$ minutes and $\sigma = 26.7$ seconds over 5 simulations.

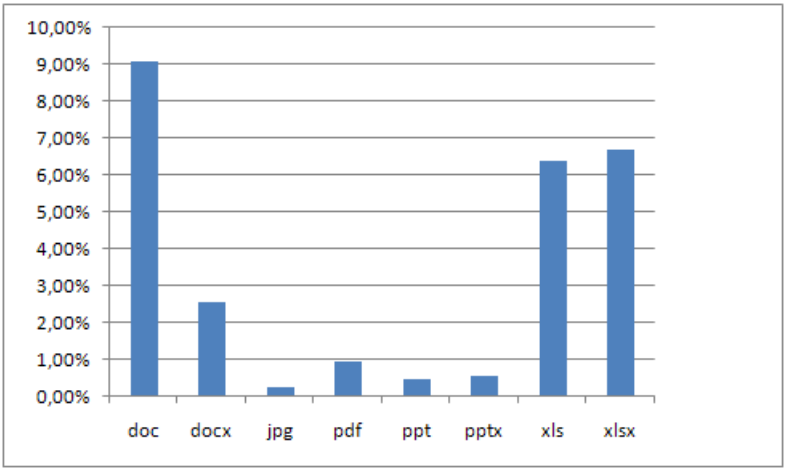


Figure 16: Illustrates the text-to-filesize ratio for the documents in the tested data set. Note that txt- and html files are not included in the figure, but that their ratio is approximately 99.8% and 67% respectively.

4.4 Program status

The constructed software is in essence a complete work. All necessary steps to produce network synchronization, load distribution, batching, indexing etc. are finished. There is support for approximately a dozen file formats.

Both index servers FAST ESP and Solr are fully compatible with the system. It's a recommendation upon deployment to produce additional extractor implementations to support an increased variety of file formats. There are some basic processors in place, like UTF-8 encoder, which can be used with any project. Although the scheme of processors is in no way required for the system to perform its work, it may still be desirable to produce processors to match the needs of each individual customer. A final remark is that the system has been stress-tested for continuous operation during four days with all features enabled without system failure. Still, it's our sincerest recommendation to test the system in a *real* production environment to work out any possible quirks prior to deploying the system commercially.

5 Discussion

This thesis is concerned with the improvement of indexing files stored on file servers; it's a known fact that the task of indexing is very time consuming and network exhaustive. The proposed improvement given by this thesis is to distribute the tasks of text extraction and data processing to file servers in the system which retain surplus processing capacity.

A series of tests have been conducted to justify the effectiveness of the system. The first observation is that the tests are very specific for the collection of test data used. Although we've strived to create a document diversity that represents that of an actual company, there is no doubt that a different data set would yield divergent results. Still, the margins of more than 300% speed increase and 1.67% network utilization seen by our constructed system compared to the existing ESP system using four file servers still conveys the likelihood of favourism towards our system. Reducing the total time taken to perform the indexing is an important goal in itself; however the emphasis lies on the vast bandwidth reduction, as bandwidth contention is an essential problem in many corporations. Consider the benefits from applying the system over a virtual private network (VPN) where file servers may spread over several distant sites.

During the test suite, bandwidth never became a major limitation to the system. It is clear that the performance gains would be substantially higher where the network poses a more severe limitation. The trend for both time consumption and network utilization presented in the results section is theoretically not limited to the four computers used for testing. The limitation was due to a practical standpoint rather than of physical nature. We make the theoretical assumption that the roughly linear characteristics will be retained to some degree when adding more file servers to the system, up to the point where the number becomes unmanageable due to network overhead for group communication etc.

Even though tests indicate benefits for the constructed system, it is still necessary to deploy the application on each file server to obtain the best system wide characteristics. Servers suffering of already heavy load will reduce the benefits of the system. The vast majority of all processing capabilities are needed during the initial indexing run. If no files are updated or there is no addition of files to the system, any subsequent crawler iterations will produce neither more work nor require any bandwidth. Generally the system obtains its most favorable characteristics, compared to the ESP file traverser, when all nodes desire access to the indexing server. Obtaining an even spread of data across the file servers also serves the purpose of limiting the impact of a node failure.

The system offers additional functionality beyond the scope of the test results. The load distribution module was deactivated during all test runs

due to the lack of useful distributable processing steps. Tasks can be derived to offer functionality unchallenged by similar systems. If desirable, some of the processing steps traditionally performed internally by an indexing server can be performed as a distributable step, thereby reducing the amount of work that needs to be performed centrally by the indexing server.

The system maintenance is considered relatively low as once the system is started it requires little manual intervention. Occasionally the logs need to be accessed to determine the state of the application and to detect possible errors. The deployment procedure is nothing more complicated than copying the bundle of files to the desired location and adapt a few variables in the projects main configuration file. Some complexity is added to the task if the system is intended to be used with SSL security as a set of keys must be distributed among the servers.

6 Conclusion

We have seen some of the problems found in indexing a set of files located on different servers in a network, and the benefits found in applying distributed techniques to mitigate those. The theory is backed by a program which has been the entry point to all tests performed. The system, which foremost employs distributed text extraction, shows how the bandwidth requirement is reduced by more than 60 times for the selected data collection of documents with a total size of 650 MB. Compared to the commercial system of the Microsoft subsidiary FAST ESP, the distributed system completes the entire indexing operation 1.42 times faster using one content provider. Moreover the system offers suberb scalability as using two content providers provide 2.14 times faster operation, 3 content providers 2.63 times faster, etc. In fact the performance figures are derived without taking full advantage of the constructed load sharing mechanism.

Although the performance figures are derived using close to optimal settings for both systems, they are still subject to the hardware configuration of the servers and to the data set used throughout the tests. However, the collection of data is selected to be somewhat representative to that of an average corporation. We consider that the constructed system can favorably be used to replace virtually any existing system concerned with gathering data for an indexing server presuming the servers have surplus processing capabilities; particularly as the application provides a unison interface allowing compatibility with multiple types of index servers.

Naturally there is room for improvements to the system. The constructed system is basically only applicable to file types with a matching extractor defined. It's possible that increased performance characteristics can be obtained by providing slightly overlapping network access rather than strictly sequential. Moreover the effects of WAN multicasts are untested for the current implementation. It may also be considered to provide a guided user interface to the application, allowing easy configuration as well as a graceful restart mechanism. Lastly, a monitoring server could be used with the system to provide alarm generation on e.g. database failures and to provide a point of central administration.

We conclude that the techniques of distributed systems well can be utilized to improve the quality and performance of many traditional systems and in particular systems which provides its service using a communication network.

References

[1] S. Dixit and T. Wu. *Content networking in the mobile internet.* Wiley-Interscience, John & Sons, 2004.

[2] Jim Waldo. The jini architecture for network-centric computing. *Commun. ACM*, 42(7):76–82, 1999.

[3] A. Cournier, A.k. Datta, F. Petit, and V. Villain. Self-stabilizing pif algorithm in arbitrary rooted networks. pages 91–98, Apr 2001.

[4] Wim Hesselink. A mechanical proof of segall's pif algorithm, 1997.

[5] G. Colouris, J. Dollimore, and T. Kindberg. *Distributed Systems, Concepts and design.* Addison-Wesley, 2005.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[7] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[8] M. Friedemann. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[9] Manuel Scholz, Frank Bregulla, and Annika Hinze. Using physical clocks for replication in manets. In *PERCOMW '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 114–119, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[11] Ajei Gopal, Ray Strong, Sam Toueg, and Flaviu Cristian. Early-delivery atomic broadcast. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 297–309, New York, NY, USA, 1990. ACM.

[12] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[13] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.

[14] M. Frans Kaashoek, A. S. Tanenbaum, and S. F. Hummel. An efficient reliable broadcast protocol. *SIGOPS Oper. Syst. Rev.*, 23(4):5–19, 1989.

[15] G. Colouris, J. Dollimore, and T. Kindberg. Archive material from edition 2 of distributed systems: Concepts and design, 1994.

[16] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[17] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. Von Laszewski. A fault detection service for wide area distributed computations. pages 268–278, Jul 1998.

[18] Mina Shirali, Abolfazl Haghighat Toroghi, and Mehdi Vojdani. Leader election algorithms: History and novel schemes. In *ICCIT '08: Proceedings of the 2008 Third International Conference on Convergence and Hybrid Information Technology*, pages 1001–1006, Washington, DC, USA, 2008. IEEE Computer Society.

[19] J. Villadangos, A. Cordoba, F. Farina, and M. Prieto. Efficient leader election in complete networks. pages 136–143, Feb. 2005.

[20] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 990–999, New York, NY, USA, 2006. ACM.

[21] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM.

[22] Yung-Terng Wang and R.J.T. Morris. Load sharing in distributed systems. *Computers, IEEE Transactions on*, C-34(3):204–217, March 1985.

[23] Berger, Browne, E. Berger, and J. C. Browne. Scalable load distribution and load balancing for dynamic parallel programs. In *In Proceedings of the International Workshop on Cluster-Based Computing 99, Rhodes/-Greece*, 1999.

[24] The Unicode standard v. 5.1. `http://www.unicode.org/versions/Unicode5.1.0/`.

[25] Microsoft Corporation. Compound binary file format specification., 2007.

[26] Microsoft Corporation. Microsoft office binary (doc, xls, ppt) file formats. `http://www.microsoft.com/interop/docs/OfficeBinaryFormats.mspx`.

[27] Microsoft Corporation. Microsoft office powerpoint 97-2007 binary file format specification [*.ppt], 2007.

[28] Adobe System Incorporated. Pdf reference, 6th edition: Adobe portable document format, version 1.7, nov 2007.

[29] Network Working Group. Hypertext markup language -2.0, 1995. http://tools.ietf.org/html/rfc1866.

[30] M. Konchady. *Building Search Applications: Lucene, LingPipe and gate.* Mustru Publishing, Oakton (VA), 2008.

[31] S. et al Deerwester. Improving information retrieval with latent semantic indexing. volume Proceedings of the 51st Annual Meeting of the American Society for Information Science, pages 36–40, 1988.

[32] K. Sparck Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808, 2000.

[33] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[34] Cron Javadoc API by Opensymphony (2000-2005). http://www.opensymphony.com/quartz/api/org/quartz/CronTrigger.html.

[35] Cron Tutorial by Opensymphony (2000-2005). http://www.opensymphony.com/quartz/wikidocs/CronTriggers%20Tutorial.html.

[36] SQLite database. http://www.sqlite.org/.

[37] J. Postel. DoD standard Internet Protocol. RFC 760, January 1980. Obsoleted by RFC 791, updated by RFC 777.

[38] Apache PDFBox Java PDF Library. http://incubator.apache.org/pdfbox/.

[39] Version 2.0 Apache License. http://www.apache.org/licenses/LICENSE-2.0.

[40] Apache POI Java API To Access Microsoft Format Files. http://poi.apache.org/.

[41] FAST ESP. http://www.fastsearch.com/.

[42] Apache Solr. http://lucene.apache.org/solr/.

[43] CORBA FAQ. http://www.omg.org/gettingstarted/corbafaq.htm.

[44] The Ensemble distributed communication system. http://dsl.cs.technion.ac.il/projects/Ensemble/.

[45] The Internet Communications Engine. http://www.zeroc.com/ice.html.

[46] JGroups A Toolkit for Reliable Multicast Communication. http://www.jgroups.org/.

Glossary

Apache License v2	A free-software license authored by the Apache Software Foundation (ASF), 39
API	Application Programming Interface, 9
ASCII	American Standard Code for Information Interchange, 27
Consensus	To reach agreement in a decision, 12
Deadlock	A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, 43
DHCP	Dynamic Host Configuration Protocol, 13
Facade	A design pattern which provides a unified interface to a set of interfaces in a subsystem, 35
File crawler	A computer program traversing a filesystem in a methodical, automated manner, 35
Flooding Algorithms	Algorithm for distributing material to every part of a connected network, 15
FTP	File Transfer Protocol, 36
HTML	HyperText Markup Langugage, 32
MD5	Message-Digest algorithm 5, a widely used cryptographic hash function with a 128-bit hash value, 37
Metadata	Data about data, 25
Multicast	Sending data to a specific group of destinations at once, 13
Network congestion	Occurs when a link or node is carrying so much data that its quality of service deteriorates, 22
Piggybacking	To attatch contents to a message already scheduled for delivery, 23
Proxy	A design pattern which provides a surrogate or placeholder for another object to control access to it, 35

State machine	The state in a state machine is only dependent on the series of previous states and inputs, 43
UTF-8	Unicode Transformation Format-8, 27
ZIP	The ZIP file format is a data compression and archive format., 31

Index