University of Zurich
Department of Informatics

ASSIGNMENT THESIS — Communication Systems Group, Prof. Dr. Burkhard

Distributed Peer-to-peer FastSS Indexing

Dalibor Peric
Zürich, Switzerland
Student ID: 01-917-657

Supervisor: Fabio Victora Hecht, Thomas Bocek
Date of Submission: February 15, 2008

University of Zurich
Department of Informatics (IFI)
Binzmuehlestrasse 14, CH—8050 Zurich, Switzerland

ifi

Assignment Thesis
Communication Systems Group
Department of Informatics (IFI)
University of Zürich
Binzmuehlestrasse 14, CH—8050 Zurich, Switzerland
URL: http://www.csg.unizh.ch/

Zusammenfassung

Im Vergleich zu Client/Server-Architekturen weisen Peer-to-Peer (P2P) Architekturen hinsichtlich der Skalierbarkeit (*scalability)*, des Lastausgleichs (*load balancing*) und der Fehlertoleranz (*robustness*) besondere Vorteile auf. Strukturierte P2P-Systeme bieten zudem effiziente *Lookup*-Mechanismen: eine exakte Suche erfolgt in logarithmischer Zeit im Verhältnis zur Anzahl Peers im System. Zusammenlegung der ungefähre Suche mit strukturierten P2P Systemen ist ein populäres Forschungsgebiet.

P2P-Fast Similarity Search (P2PFastSS) ist ein, an der Universität Zürich entwickelter, Algorithmus, der die effiziente ungefähre Suche in einem strukturierten P2P-System erlaubt. Durch spezielle Indizierung der Daten, kann er verwendet werden, die exakte Suche um die Suche ähnlicher Worte zu ergänzen. Zur Zeit ist es üblich, dass die Indizierung der Daten von einem einzigen Peer sequentiell durchgeführt wird, während die anderen Peers untätig sind. Diese Tatsache steht im widerspruch zur Anforderung eines Lastausgleichs und weicht somit vom P2P Ansatz ab. Ebenso sollte ein P2P System zusätzliche Features wie zum Beispiel, dauerhafte Speicherung von Index-Daten, Messung der Suchbegriffsrelevanz in Bezug auf den Kontext, Seitenverwaltung von Suchergebnissen und Vorschau von Dokumenten in einem Web-Interface bieten.

Mit dem Gedanken, die Lastverteilung in einem P2P-System zu verbessern, wurde beschlossen, ein Indizierungsmechanismus zu designen und zu implementieren, welcher eine Parallelisierung der Speicher- und Index-Aktivität ermöglicht. Darüber hinaus sollen die im obigen Abschnitt erwähnten zusätzlichen Features implementiert werden. In dieser Vertiefungsarbeit werden zunächst die notwendigen Konzepte von P2P-Architekturen eingeführt und im Folgenden, einen Überblick über den state-of-the-Art im Bereich der ungefähren Suche in P2P-Netzen gegeben, wobei insbesondere auf den P2PFastSS Algorithmus eingegangen wird. Anschliessend werden die Anforderungen für die Planung der einzelnen Aufgaben vorgestellt und die Motivationen den Entwurf- und Umsetzungsentscheidungen jeder Aufgabe gegeben.

Aktuelle Version von P2PFastSS führt die Indizierung und Speicherung im parallelen durch, so dass sich die Belastung auf mehrerere Peers im System verteilt. Momentan speichern die Peers die Daten auf der Festplatte anstatt im Hauptspeicher. Den Indeizierten Suchbegriffen werden Relevanzpunkte zugeordnet, welche dann für die Bestimmung der Reihenfolge der Suchergebnisse verwendet werden. Seitenverwaltung und eine Vorschau der Suchergebnisse werden jetzt durch das Web-Interface ermöglicht.

Abstract

Peer-to-peer (P2P) architectures present benefits like scalability, load balancing and fault tolerance when compared to Client/Server architectures. Structured P2P systems furthermore feature efficient lookup mechanisms: an exact search is usually performed with logarithmic complexity relative to the number of peers in the system. Integration of similarity search mechanisms in structured P2P networks is a popular research topic.

P2P Fast Similarity Search (P2PFastSS) is an algorithm developed at University of Zurich that allows efficient similarity search in any structured P2P system. By creating and using special indexes of the data in the P2P system, it can help search for similar words in addition to the exact search. Although search was made in a P2P fashion, indexing was performed sequentially by a single peer, while other peers were mostly idle. This situation lacks the load balancing property and diverges from the peer-to-peer approach. Implementation of additional features, like permanent storage of index data, measuring keyword relevance regarding its context, paging of search results and preview of documents in a web interface, is a key concern. For these reasons, a decision was made to design and implement a distributed indexing mechanism, thus parallelizing the peer's storage and index activity, with the goal of achieving a better load balancing and reliability in the P2P system.

This thesis introduces the concepts of P2P architectures and provides an overview of the state-of-the-art in the area of similarity search in P2P networks, especially of the P2PFastSS algorithm. Thereafter, the requirements for the design of various tasks and the decisions behind design and implementation choices of each task are presented.

Current version of P2PFastSS performs indexing and storing in parallel, and the indexing load is distributed among all peers in the system. Peers currently store data and indexes on disk instead of in memory. Indexed keys are given a relevance score, which is then used for sorting the search results. Paging and preview of search results is now implemented in the web interface.

Acknowledgments

First of all, I would like to thank Professor Burkhard Stiller, for introducing me to the exciting world of computer networks and communication systems, and for giving me the opportunity to complete this assignment for the communication system group. Very special thanks go to his assistants Fabio Victora Hecht and Thomas Bocek for their great support, valuable advices and constant presence. Their generous indications and guidance assisted me while writing the final version of this paper.

I would also like to thank my wife Sandra and my family for their incessant support and encouragement in all years of my studies.

Last, but not least, I would like to thank my friends, my colleagues and everyone who was involved in writing of this assignment paper.

Table of Content

Zusammenfassung i

Abstract ii

Acknowledgments iii

1 Introduction 1

1.1 Motivation and Problem Description 1

1.2 Assignment Objectives 2

1.3 Assignment Outline 2

2 Related Work 3

2.1 Peer-to-peer Networks 3

2.1.1 Definition of Peer-to-peer systems 3

2.1.2 P2P System Characteristics 3

2.1.3 Peer-to-Peer Overlay Architectures 4

2.2 Similarity Search in P2P Architectures 7

2.2.1 Basic Concepts for Similarity Measuring 7

2.2.2 Similarity search algorithms 8

2.2.3 Fast Similarity Search 8

2.2.4 Similarity Search in structured P2P Networks 9

3 Design and Implementation 11

3.1 Technical Requirements 11

3.2 Distributed Indexing 11

3.2.1 Motivation 11

3.2.2 Design 14

3.2.3 Implementation 14

3.3 Disk-based Storage 15

3.3.1 Motivation 15

3.3.2 Design 15

3.3.3 Implementation 16

3.4 Keyword Relevance 17

3.4.1 Motivation 17

3.4.2 Design 17

3.4.3 Implementation 18

3.5 Paging of Search Results 18

3.5.1 Motivation 18

3.5.2 Design 18

3.5.3 Implementation 19

3.6 Preview of Search Results 19

3.6.1 Motivation 19

3.6.2 Design 19

3.6.3 Implementation 19

4 Conclusion and Future Work 21

4.1 Summary 21

4.2 Conclusion 21

4.3 Future Work 21

References **23**

List of Figures **27**

List of Tables **27**

A Contents of the CD **29**

1 Introduction

Effective and efficient search for information in computer networks and distributed computer systems is an important subject, extending well beyond Computer Science research field. Google's success is a known example of what impact innovation in search technology can have on everyday life: it improved the satisfaction of World Wide Web users, while projecting a newcomer firm to the top of the IT-Industry.

While traditional search engines and other online applications mostly operate in networks implementing Client/Server-based architectures, where a clear distinction between the functionality of a client and that of a server exists, another important network design emerged towards the end of the last decade: Peer-to-peer (P2P) architecture. Although there are different P2P schemes, in its pure form all participants are connected with an *overlay network* - a virtual network running on top of an existing network - and are have equal roles (hence the word *peer*). There are no such distinctions as “client” and “server”: every peer can request resources from other peers, while offering its resources at the same time. In this thesis the term *node* is also used to indicate a computer system or application connected to a P2P network.

Initially mislabeled by mainstream media as a technology for illegal sharing of copyright material, P2P systems have so far proven themselves as a valid complement, when not alternative [1], to the Client/Server ones. As a matter of fact, P2P approaches ideally offer superior *scalability* (no central bottleneck) and *robustness* (no single point of failure) compared to the Client/Server model. But P2P is ultimately not perfect: potential misuse, missing incentives for the peers to share their resources and the fact that a (decentralized) P2P system is difficult to control by an authority, represent some of its possible drawbacks.

Different P2P overlay network configurations have been proposed and researched [7], [8], [11], [13]: from *unstructured centralized* topologies, across *fully decentralized* and *hybrid* topologies, with *structured* P2P systems such as CAN [18], Chord [21], Pastry [19] and Kademlia [20], using *distributed hash tables* (DHT), being the most recent. Structured P2P systems implement mechanisms for efficient storage and retreival of peers resources but have limited support for *similarity search.* This means, in textual search context, that if a user misspells the name of the searched resource in a structured P2P system, peers will not get the result.

1.1 Motivation and Problem Description

One incentive for implementing similarity search, or *approximate string matching*, in P2P systems is that users commit errors. A system that allows to find data with misspellings is a desirable property. Another area where similarity search can be applied is the service discovery [2] application domain because service descriptions are frequently made of informal textual information.

Former two situations are examples of text retrieval, but there is a vast number of other possible applications for approximate string matching, such as [3] signal processing (recovering the original signals after their transmission over noisy channels) and computational biology (finding DNA subsequences after possible mutations).

P2P Fast Similarity Search, or in short P2PFastSS, is an algorithm developed at University of Zurich [2] to address the deficiency of similarity search in structured P2P systems. P2PFastSS finds similar keys in any DHT using the edit distance metric, and is independent of the underlying P2P routing algorithm. P2PFastSS, unlike *online search algorithms*, doesn’t perform the search on the data itself; it uses special indexes of the data instead. Therefore data needs to be indexed in advance, before searching.

In the Client/Server implementation of FastSS [4], [5], only server(s) have the task of building and storing the indexes, while the clients submit search queries. On the opposite, in P2P environment all peers participate, by offering their resources (which can be memory, computing power and / or bandwidth).

P2PFastSS defined three types of tasks: storing articles, indexing articles and collecting statistical data. Despite the fact any node in the system could perform these tasks, they were performed sequentially. Besides this sequential processing, the CPU load wasn't distributed between nodes: the computation of data indexes was done by a single node. Simulations showed that indexing node had a high CPU load, while the CPU of other peers was mostly unused.

Another shortcoming was that the data and the results of data indexing were initially stored in the nodes' volatile memory (RAM). This meant that if the system was shut down, or crashed unexpectedly, both the data and the indexes would be lost and data would have to be re-indexed all over again before the system could be used. Other possible enhancements are improving the existing P2PFastSS web interface, which allowed to index and display search results, with the goal on displaying the search results in a meaningful way: the more relevant results were to be shown first. These ordered results were then to be separated in navigable sets, for easier and faster browsing. Finally, previews of the documents containing the (exact or similar) searched key, were to be visualized together with the results.

1.2 Assignment Objectives

The list below indicates the goals of this thesis considering the assignment's task description and the above additional motivations. Objectives of this assignment are:

- To provide an overview of the state-of-the-art in the areas of:
  - P2P networks: different approaches and different concepts
  - Similarity searches in structured P2P networks
- Design and implementation of a distributed indexing mechanism based on P2PFastSS
- Design and implementation of the following features:
  - Storage of data and data indexes on permanent memory
  - Relevance function to compute the "relevance" factor of a keyword regarding its importance in the document that contains it
  - Ordering of search results based on the relevance of search results
  - Paging of search results in the web interface
  - Preview of the document containing the search results

1.3 Assignment Outline

The rest of this document is arranged as follows: the next chapter will discuss the various types of P2P systems, focusing in particular on structured P2P systems, and then will introduce the basic concepts in similarity search in P2P networks, listing all available search mechanisms for those systems. The third chapter will give a more detailed description of P2PFastSS' implementation and the motivations behind distributed indexing-design and implementation. This chapter will also showt the decisions regarding design and implementation of the additional features. The concluding chapter contains a brief summary, conclusion and future work.

2 Related Work

This chapter will present an overview of various existing P2P architectures and an introduction into existing similarity search techniques, in general, and those developed for structured P2P networks. Since P2PFastSS is an algorithm that operates in *structured* P2P systems, this chapter will mainly focus on those particular systems in both sections.

2.1 Peer-to-peer Networks

The vision of early developers of ARPANET and the Internet was actually very similar to a P2P architecture [6]: in this conception every computer could access the resources of all other computers on the network, while making its own resources available.

Various events contributed in changing the Internet landscape from primarily peer-to-peer to the Client/Server architecture. According to Gradecki [6], the Internet has progressively become more commercial, while corporations limited access to their resources and information with firewalls. Home computers couldn't match the power of the computers that formed the Internet backbone. In addition, most popular web services and applications, like email, WWW or FTP are based on Client/ Server architecture.

2.1.1 Definition of Peer-to-peer systems

Client/Server organization is an asymmetrical structure because there is a clear distinction between server and client roles: a server is a centralized computer system which offers its resources or services to a group of clients. The network address of a server has to be well-known [1] and it must be reachable anytime. These restrictions do not apply for client systems, which do not have to be accessible all the time and which address must not be fixed.

On the opposite side, P2P systems have symmetry in roles [7], where each host can be, at the same time, a server and a client, producer and consumer, of services and/or resources of other hosts in the network. By literature definition [8] a P2P system is a "self-organizing system of equal, autonomous entities (peers) which aims for the shared usage of distributed resources in a networked environment avoiding central services

2.1.2 P2P System Characteristics

Most P2P systems provide at least some of the following benefits [9], [10]:

- Workload can be spread to all peers; this can deliver massive resources and computing power.
- Centralized control and management is not required.
- Without centralization there is no "Single point of failure" and no central bottleneck.
- New peers can easily be added to the system; the ability to expand the network is also called *scalability.*
- P2P network will still function when some of its peers are not working properly or leave the network temporarily at will. In consequence it is more fault tolerant than other systems.

However, P2P systems also have some drawbacks [9], [10]:

- P2P systems, especially decentralized ones, are difficult, for an authority or a company, to control or shut down.
- In some P2P overlay organizations, lookup requests generate a huge communication overhead and may lead to false negative results due to lookup timeouts.
- Consistency of information is difficult to maintain, since this can be replicated in many peers.
- Egoistic behavior of peers: some peers may want to access other's resources without sharing proper ones.
- Malicious behaviour: a peer may behave improperly by replying innacurately, providing wrong content, or denying responsibility for data it is designed to keep.

Table 1 summarizes the main benefits and drawbacks of P2P and Client/Server paradigms according to [10].

Table 1: P2P and C/S - Benefits and Drawbacks

	Benefits	***Drawbacks***
Client/Server	• Trust / Security • Manageability • Consistency	• Single Point of Failure • Scalability • Costs
Peer-to-Peer	• Extensibility / Scalability • Fault Tolerance • Resistance to lawsuits	• Imperfect incentive schemes • Maliciousness • Overhead

2.1.3 Peer-to-Peer Overlay Architectures

A common property for all P2P architectures is that the actual data transfer between two peers is completed through a direct connection *on a network level* between the peer offering a resource and the peer requesting for it [11]. While there can exist various network topologies between the nodes on the level of network connections (*centralized, ring*, *hierarchical*, *decentralized and hybrid* [12]), a P2P network is actually an *overlay network* [13], operating at the application layer.

An overlay network is a “virtual” network created on top of an existing network. The nodes in the overlay network use each other as routers to send data [14]. They implement a network abstraction on top of the network provided by the underlying substrate network [15], for example the Internet.

P2P overlays can generally be grouped in *structured* and in *unstructured* networks, depending if the overlay topology takes form of a regular structure or not. In the following text, when not specified otherwise, the term *network* is used as a synonym for *overlay network*.

An unstructured P2P system consists of nodes joining the network, usually with some loose rules, and without any previous information about the topology [7]. If nodes leave the network, no specific reorganization activity takes place, thus the topology does not take the form of a regular structure, hence the *unstructured* keyword.

The routing mechanism used for the nodes to find resources is flooding, or broadcasting. Each request from a peer is flooded to directly connected peers, which themselves flood their peers and so on, until the request is answered or a maximum number of flooding steps occur, and the query is terminated [16].

While flooding-based techniques are effective for locating, popular, highly replicated items and are resilient to peers joining and leaving the system, they are inadequate for locating rare resources. Another problem is that his approach shows poor scalability as the load on each peer *grows linearly* with the total number of queries and the system size [7]. The number of lookup messages grows exponentially relative to the number of peers in the system.

Unstructured P2P systems may in addition have centralized elements, like a central directory index (as early Napster), or "superpeer" elements (like Gnutella 0.6) but those versions will not be investigated because their influence on this assignment is minor.

Structured Peer-to-peer (P2P) systems represent the latest subject in research of P2P systems. In those systems, the overlay topology is predetermined and the overlay network is always maintaining a structure. If peers leave, the network structure will temporarily become imperfect [17]. But the connections will be readjusted, as soon as the system recognizes the structural flaw. Likewise, if new peers join, they are assigned a position in the network which does not violate the foreseen structure.

Although structured P2P systems like CAN [18], Pastry [19], Kademlia [20], and Chord [21] have different designs, the following approach is common to all of them: both peer and their content are mapped into the same *abstract space*, and this space is divided by a *partitioning scheme* into ranges, with each peer (node) being responsible for a particular range. An *overlay network* then connects the nodes, allowing finding the responsible node for any given key.

The abstract key space, the partitioning scheme and the overlay network are components of a *distributed hash table* (or DHT). Hence, distributed systems that provide lookup service using a DHT can be defined as structured P2P systems.

Like an ordinary hash table, a DHT provides the basic operations *put(key, value)* for storage and *get(key)* for retrieval. When data is to be stored, a hash function is initially applied to the key and the put method is called. A search for given data is performed using the get method, where the same hash function is used again on the key of the searched data. Figure 1 illustrates the use of DHT operations by a distributed P2P Application. In this case the distributed application is P2PFastSS.

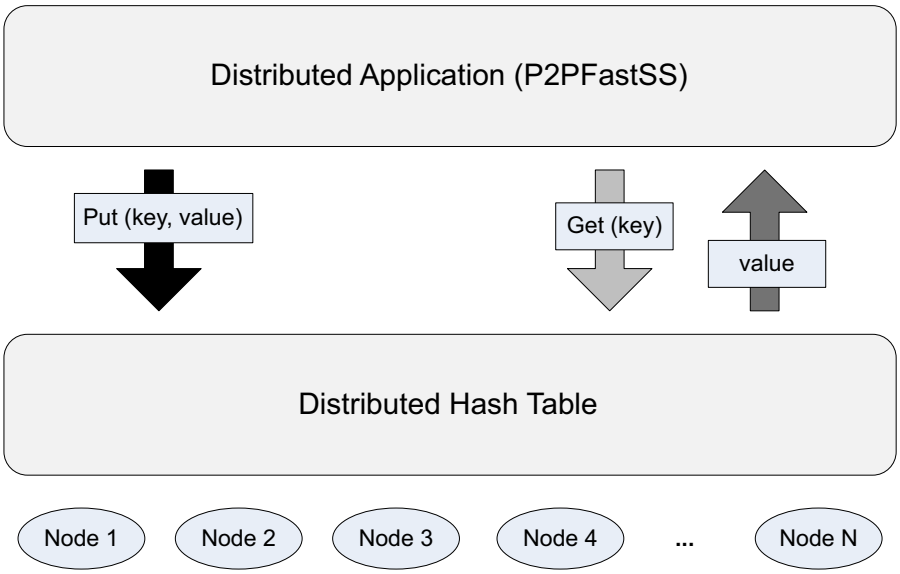

Figure 1: Distributed P2P Application running on top of a DHT

Search for information in structured P2P systems is relatively efficient: as a matter of fact, a key lookup typically requires *O(log n)* routing steps, with *n* peers present in the system. This efficiency can be achieved because each peer "knows" the network structure and can forward queries just in the right direction [17]. Unfortunately, this efficiency also has a drawback: systems using DHTs intrinsically do not support *similarity search*. The main reason for this deficiency is that calculating a hash function on two keys results in different hash values, even for greatly similar keys.

Table 2 illustrates the results of computation of SHA-1, one common cryptographic hash function, on four very similar strings. We notice that the resulting values are completely different, even if the variation is minimal. This property is desirable for cryptographic algorithms and is called the *avalanche effect* [22].

Table 2: Example of the avalanche effect

Keyword	***Result of SHA-1 applied on the keyword***
structure	`2d309ce90c549a12a47588a74687315a7c6b51a7`
structured	`52a6391f3d211759d7c537fd8ea86d52317c15f9`
structures	`41b426fb69c819c1f3a57bdc7ae4fe543fd971f1`
Structures	`c64b6d8ab6a3167af6b211df05198e555a8951eb`

Since P2PFastSS implementation uses a DHT based on the Kademlia routing algorithm, a basic description of the algorithm is given below. For a detailed description of Kademlia, and its formal proof, readers can refer to [20].

Kademlia

In Kademlia keys for the DHT are 160-bit quantities. Each node has a unique node ID in this 160-bit key space.

To publish and find (key, value) pairs, the system relies on the notion of distance between two identifiers. In Kademlia this distance is defined as their bitwise exclusive or (XOR). Given two 160-bit identifiers, x and y, it would be interpreted as an integer, $d(x, y) = x \oplus y$. For each *i*, $0 <= i < 160$, every node has a list of descriptors for other nodes (usually a {IP, UDP port number, node ID}-triple) of distance between 2^i and 2^{i+1} from itself. These list are called *k*-buckets because their size can grow up to *k*. *k* is chosen as a system-wide replication parameter and it represents a number of nodes that are very unlikely to fail within an hour of time.

Some important Kademlia's protocol characteristics are:

- Minimization of configuration messages sent by nodes to learn about each other.
- Information configuration spreads as a side-effect of key lookups.
- Nodes have the possibility to select routes with low-latency.
- Use of parallel, asynchronous queries to avoid timeout delays from failed nodes.
- It implements mechanisms to protect the network from some denial of service attacks and to ensure the persistence of (key,value) pairs.

2.2 Similarity Search in P2P Architectures

This section will introduce the main concepts used for approximate string matching and present some similarity search algorithms. Then it will provide a description of FastSS' operation, and finally it will show some approaches of implementing similarity search in structured P2P networks.

2.2.1 Basic Concepts for Similarity Measuring

In computer science and information theory, to determine the similarity of two strings of characters, the concept of *distance* is used. Distance between two strings x and y is informally defined as the "minimal cost of a sequence of operations that transforms x into y" [3]. Common used operations are insertion, deletion, substitution and transposition (swapping of adjacent letters). [3] also gives a formal definition of distance concept.

Most frequently used distance functions are, by Navarro [3]:

- *Levenshtein* or *edit distance* [23], which allows insertion, deletion and substitution operations.
- *Hamming distance* [24]: allows only substitutions
- *Episode distance* [25]: allows only insertions
- *Longest common subsequence distance* [26]: allows only insertions and deletions

2.2.2 Similarity search algorithms

Bocek et al. [2] categorize similarity search algorithms as *online* or *offline*, *exhaustive* or *heuristic*, and using a *global* or a *local metric of similarity.*

Online algorithms perform the search without prior data processing, and all data must be traversed during the search. Offline algorithms, on the opposite, pre-process (or, *index*) the target data and may store the results in memory or on disk. Main intention behind this approach is to speed up query processing.

Exhaustive algorithms guarantee to find all existing occurrences of the search query, in contrast to heuristic algorithms which offer a reduction of search time by evaluating only statistically interesting patterns, but without guarantee that all similar data will be found.

Algorithms implementing global metric consider all target data when searching for similarity, while those using local metric determine the similarity between a part of the target data and the search query.

The list below report names and short descriptions of mostly used similarity search algorithms. Their detailed review can be found in [3], while a comparison with Fast Similarity Searching (FastSS) algorithm, can be found in [4].

- **Edit distance (ED)** [27]: which uses *dynamic programming* (DP) - a method of solving problems having the properties of overlapping subproblems and optimal substructure - to compute the minimum number of operations (deletion, insertion or replacement) required to transform one string into another.
- **NR-grep** [28]: uses bit-parallelism and forward and backward searching.
- **N-grams** [3]: n-gram index is created by sliding a window of length n over the data and saving the content and position of all such windows; since indexes are being used, it is an offline algorithm.
- **Keyword Trees** and **Suffix Indexes** [29]: uses tree-shaped indexes in combination with DP to find similar words under ED model; if suffix links are used the tree data structure is even smaller and the search is terminated as soon as the one can see that a match is impossible.
- **Search with Neighborhood Generation** [30]: indexes target data using n-grams, and with a hashing function, maps each word to an integer. A search query is split into words of constant length and for each such word a neighborhood of all words with a certain ED is generated and the candidate words are obtained from the index.
- **Index Misspellings**, like **FLASH** [31]: patterns with misspellings are purposely indexed, and matching is based on statistics; a common type of algorithms implemented by search engines.

2.2.3 Fast Similarity Search

Fast Similarity Search is an exhaustive, offline search algorithm, based on the local similarity model.

The main suggestion of FastSS "consists in using an efficient variant of the neighborhood generation algorithm [..], adapted to use deletions only". In [4] the authors provide a formal proof that the edit distance problem can be solved by applying deletions only.

The indexing procedure consists of creating a deletion dictionary *of the target data*, word by word. When a search query is issued, it is transformed to generate a deletion neighborhood *of the query*, which is then compared to the indexed deletion dictionary. Since a deletion neighborhood is smaller than a regular neighborhood, it contributes to a faster search. For a dictionary with *n* words, look up

time for FastSS is independent of *n* in case of a hash-based index, or proportional to the logarithm of *n*, for a tree-based index.

2.2.4 Similarity Search in structured P2P Networks

Pier [32] combines a DHT with prefix trees and uses special path query primitive to locate nodes based on a prefix string. AlvisP2P [33] trades a "marginal loss in retrieval qualities for rare queries" for manageable index size and network traffic. Other approaches, like LSH Forest [34], include using several overlays with various hash function combinations, treated as a forest. In [35], Aekaterinidis and Triantafillou make use of substring matching, based on a subdivision of a query string into all possible substrings, and querying for those substrings too; n-grams can also be used as an alternative. Some researchers address the problem of complex queries [36] in the P-Grid based DHT by splitting relational tables vertically and index attribute values in a DHT. Some propose [37] using algebraic signatures on n-grams, ensuring data privacy, but still incurs a large cost in string searching. Ahmed et al. [38] describe a distributed pattern matching method. By their solution flexible queries such as partial and multiple keywords are supported and the algorithm is based on Bloom filters [39].

Main benefits of P2P-version of FastSS search are, according to its authors [4]:

- Efficient operation with small words in distributed system (unlike Karnsted et al. [36], Aekaterinidis et al. [35]).
- Independence of the routing algorithm, any DHT can be used (unlike Ahmed et al. [38]) .

3 Design and Implementation

This chapter reports the modifications P2PFastSS' software underwent. Each major change has a dedicate section, containing a description of previous state, list of design choices for the new version and finally, the implementation details.

3.1 Technical Requirements

The following list enumerates the technical requirements for the implementations according to the assignment task description:

- The Java version should be at least Java 1.6 and the existing P2P framework shall be used.
- Available design/implementation frameworks and libraries are to be used where possible. If necessary, the libraries are to be adapted if possible.
- JavaDoc should be used to document the code. JUnit [40] or similar tools should be used for testing purposes.
- An open source license (BSD, GPL license [41]) for the code should be used.

3.2 Distributed Indexing

3.2.1 Motivation

P2PFastSS was implemented in Java and uses a DHT based on the Kademlia routing algorithm.

An important feature of the DHT on top of which P2PFastSS works, is the possibility of using asynchronous, non blocking, communication. This feature allows implementing and using the asynchronous, parallel queries described by the Kademlia algorithm. Main advantage of asynchroneous queries is that they "tollerate node failures without imposing timeout delays on users." [20]

The following example illustrates the effects and possible benefits of asynchronous communication compared to synchronous one: node *A* needs to query node *B*; if the communication is synchronous (blocking) then after the node A has established communication with node *B*, node *A* will block itself until node *B* completes his tasks and answers. By staying idle until *B* answers, node A may waste significant time, which could be spent for other tasks. If the communication is asynchronous (non blocking), then after the data has been sent to *B*, node *A* can immediately return to do other, unrelated, tasks. *A* can later willingly check if the communication was successful or if *B* has returned a result. The latter approach is clearly the most flexible, but it requires careful programming to cover potential resource conflicts and possible failure of communication.

P2PFastSS' underlying framework implicitly takes care of all communications, implementing the bootstrap phase (the first connection of a peer to a P2P network) and the DHT operations *put*(*key, value*) and *get*(*key*). Those operations (or methods, in Java) return objects (e.g. *FutureBootstrap*, *FutureRouting*, *FutureDHT*, and so on), which can be saved and accessed later (polled) to check if the communication was successful or failed, or if the contacted party returned a result or not. An

even more powerful way of using “future” objects is the possibility to attach anonymous listener objects to them. These listeners must implement some predefined methods (declared in an interface) but the code inside the methods can be custom-made, so that when a special condition becomes true in the “future” object, it triggers a given method in the listener. This frees the programmer from the burden of deciding when and how frequently to poll the “future” object. If the programmer whishes to block on the connection result for some purpose, it can also call a “join” method on that object.

P2PFastSS can work with any type of textual information. To test its functionality, its authors decided to use abstracts of Wikipedia [4] articles as base documents. Both the title and the content of these articles did provide a base for similarity search. A number of articles were downloaded from Wikipedia in a structured XML file format and it was then provided to one of nodes in the P2PFastSS network.

Although each P2PFastSS node in the network had the features to:

- parse the XML file and extract articles from it,
- process the articles, word by word, for deletion neighborhood generation,
- store the extracted articles and the index data (in two separate data structures),

only the *initiating* node (the node which had access to the XML file) would start extracting a given number of articles from the XML file and index them. Other nodes in the network did not share their resources (CPU computing power, bandwidth) for the indexing. Once the deletion neighborhood was computed, a *key* containing the result of a hash function applied on the article’s title was computed and the article was stored into the DHT by using *put*(*key, article*). A normal DHT operation, put routes and stores the article in a node with node ID closest to the *key.*

A slightly different procedure was used to store the indexed data. For each deletion neighbor of a given word, a special object, called *KeyAndKeyWord*, was created. This object contained the original word, from which the deletion neighbor was generated, and the title of the article from which the word belongs. The reason behind this approach is that different words can generate equal deletion neighbors; hence, a way to connect the index data with its original target data is needed. Once the *KeyAndKeyWord* object was created and initialized, it was *put* into the DHT.

There was no logical separation in code for article storing in DHT and their indexing: it was all done by a single method *storeArticles*(). The path to the XML file could only be passed as argument before running the program: once the XML file was processed and the articles indexed, no other articles could be added while the program was running.

During this assignment work, the software was extended with a web interface which allowed selecting of the XML file to process, manual insertion of articles into the system, submitting of search queries and displaying the search results. Figure 2 shows a screenshot of the web interface’s indexing page. Two differend indexing schemes are allowed: indexing of a single article, by typing its title and content and indexing of multiple articles, by giving the path of the XML file. A screenshot of the search page is shown with Figure 3. A link to the index page can be noticed in top left corner.

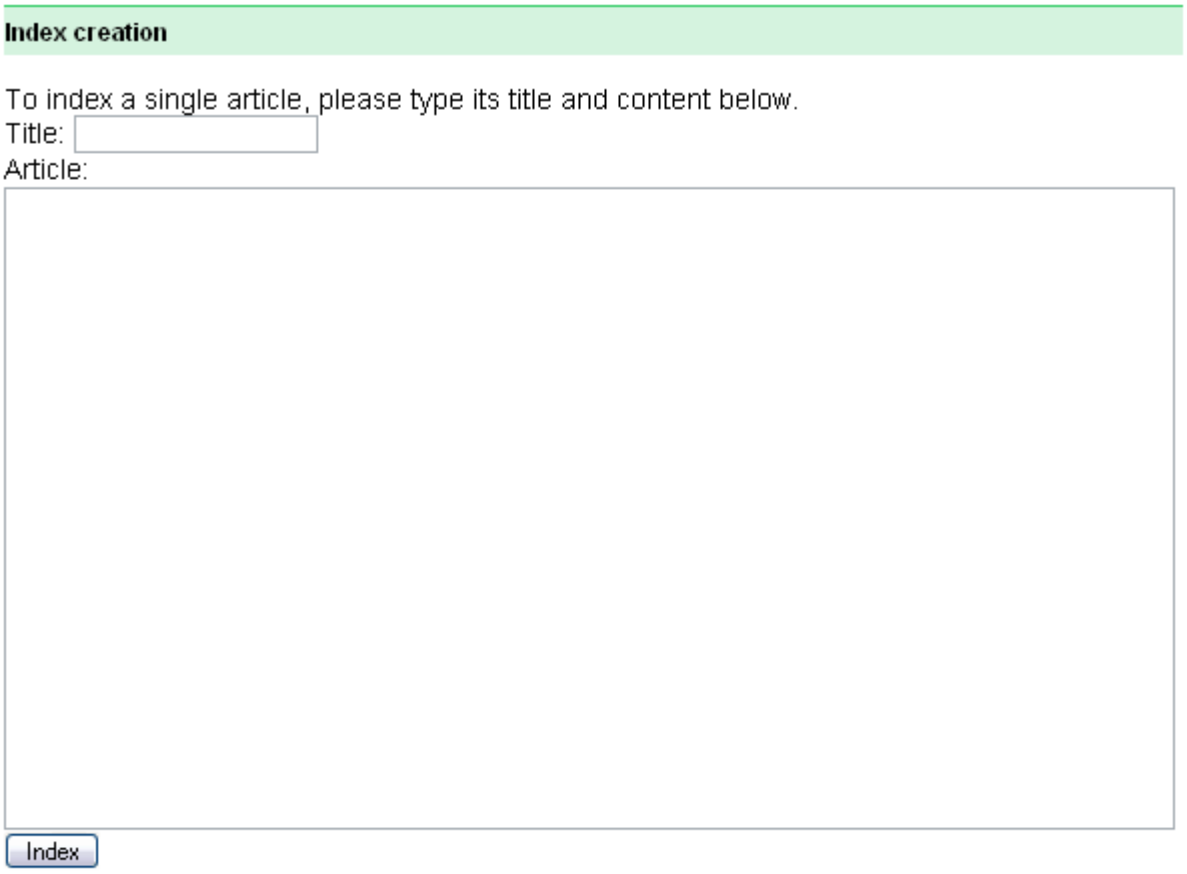

Figure 2: P2PFastSS web interface indexing page

Figure 3: P2PFastSS web interface search page

Each node had separated data structures for managing the storing procedure of articles and of the index data. Both structures used implemented the *Map* interface and therefore map objects to values.

For the articles, two *ConcurrentHashMap*s (concurrency-supporting hash tables) were used, together with following *Types* as (key, value) pair: (*BigInteger*, *Map<String*, *Object>*). The reason behind

this non-trivial "map-in-map" approach was to allow the user to specify an additional *domain* as a *String*, allowing storing more objects for a single *key*. There were two maps used for the articles, one for objects larger than 1'200 bytes, and the other for smaller objects. The motivation was performance-driven: if a packet greater than 1'200 bytes must be sent, TCP protocol is used, otherwise UDP.

For the index data a *ConcurrentHashMap* was employed as well, but the types used as (key, value) pair were (*BigInteger*, *List<Object>*). Thus, a deletion neighbor was stored together with a list containing all articles generating that given neighbor.

3.2.2 Design

In the previous version of P2PFastSS the final distribution of articles and of the index data was correct, but since only a single node was responsible for all the indexing, CPU time of other nodes was not fully taken advantage of.

An *initiating* node is still needed for the new version, because only that node has access to the XML file. Even if all nodes had access to the data, a coordinating entity would still be needed, to avoid multiple indexing of the same data in different nodes, with consequent waste of resources. Therefore, the behavior of the *initiating* node was changed: it would only extract articles from the XML file and *put* them into the DHT. The indexing is done by the nodes which would become responsible for the article. After indexing an article, they also call *put* on the generated deletion neighborhood, to store it in the DHT.

One important issue on this approach is that nodes have to be able to differentiate between receiving an article and something else (keyword), and to start indexing them at that point.

3.2.3 Implementation

The first action was to logically separate the various tasks, by using different (Java) methods for each of them. So the *storeArticles*() method was replaced by removing the code that was not involved with article storing. A new method for article indexing purpose, *indexArticle*(), was created. The method parameters are the article to index, and an integer to indicate the edit distance desired for the indexing. This method creates *KeyAndKeyWord* objects with generated deletion neighbors and passes them to the DHT for storing.

To avoid blocking a node while it is indexing articles, a dedicated *thread*, called *IndexRunner,* is added to the P2PFastSS node. It takes care of indexing and, since it is a separate thread, it runs in parallel with the node. When there are no articles to index it is put in *sleep* mode.

Different solutions were considered regarding how could a node could differentiate what type of objects it stores in his map. Initially, message payload analysis and reengineering of the class managing the article's map (*StoreMultiMapMemory*), to allow callbacks on the node's main class, was considered, but those solutions would have been complex and inelegant. A much more elegant solution is to use the *listener pattern*: it allows classes to notify each other about events. *StoreMultiMapMemory* was the slightly modified, to allow *listeners* to be added to it. A last update to this class' code was to notify all its listeners whenever an article was going to be stored on the map. Finally, node's constructor was modified, so that an anonymous listener is created and added to the node's *StoreMultiMapMemory.* This way, whenever an article is stored, eventual listeners are notified. They then send the article to the *IndexRunner* for indexing and *putting* the index data in the DHT.

3.3 Disk-based Storage

3.3.1 Motivation

The Original implementation of P2PFastSS used peer's volatile memory to store the data. While this allowed fast access to the stored information, it had two main drawbacks:

- Given the nature of RAM, once the P2PFastSS application running on a node was closed (intentionally or not), all article and index data stored in that node were lost. If the whole P2PFastSS network was shut down, all articles would have to be processed again to return to the previous state.
- Index data size tends to be multiple times the size of the text. In [4] it is stated that a 388 KB dictionary results in an approximate 100 MB index for an edit distance of 2, which is more than 250 times larger. This situation causes the P2PFastSS application to run out of system memory.

By implementing a disk-based storage, these drawbacks could be countered: nodes could rejoin the P2P system after failure, and the system could resume after a shutdown, without need to re-index all the data.

3.3.2 Design

Before implementing the disk-based storage for P2PFastSS, an important decision had to be made. It concerned the choice of storage system to be used: the file system (FS) or a database management system (DBMS). Table 3 shows some key benefits and drawbacks considered for both systems.

It was decided to use a relational database manager. HSQLDB [42] was used for the following reasons:

- It is written in Java and offers Java Database Connectivity APIs,
- it is available under BSD license,
- library size is small (around 600 KB for the standard edition),
- no need to start the server in separate Java Virtual Machines (can run in In-Process mode).

The final design step was to design the relational schemes which are created at the first run of the P2PFastSS application. Table 4 illustrates the relational schema used for storing of articles and Table 5 the relational schema used to store index data.

Attribute hash in Table 4 represent the hash code of the article's title, the title and abstract store the title respectively abstract text data. Since articles have unique titles, their hash codes can be used as primary keys for the relational table; this should increase lookup speed in the table. Instead of having two different tables for different size of articles, a single table is used. When a *get*(*key*) is received and there is a row where hash is equal to *key* the size of the article is computed at runtime and the appropriate TCP or UDP connection will be established. A primary key for relational schema in Table 5 is created with the combination of all three attributes *hash*, *originalWord* and *articleTitle*:

Table 3: Benefits and drawbacks of File Systems and Database Management Systems

	Benefits	**Drawbacks**
FS	• Simplest approach: Java *Serialization* could be used. • Special file format could be used. • Can be compressed. (DB also)	• Java *Serialization* is slow and sensitive to versioning; size of serialized data usually larger then object's size in memory. • If special format used: parsing of file is necessary. • Read / write access must be managed and synchronized, as both can happen concurrently
DBMS	• Standardized Language (SQL). • Open source DBMS software available. • Complex queries possible • DBMS responsible for concurrent read/write access • Flexibility in accessing database table fields. • DBMS responsible for data integrity	• CPU overhead from DBMS threads • Memory overhead from DBMS threads

Table 4: Relational schema for storing articles

ARTICLES	hash	title	abstract

Table 5: Relational schema for storing index data

INDEX	hash	originalWord	articleTitle

they are all needed to uniquely represent a given deletion neighbor, of a given keyword, in a given article.

3.3.3 Implementation

A decision was made, to create two new classes implementing the *Map* interface, *DBArticleMap* for article storage and *DBIndexMap* for index data storage. These classes implements the fundamental *Map* methods, *get*() and *put*() and they use JDBC API to interact with the HSQLDB database system trough SQL queries. This approach was preferred because it keeps a good modular and reusable design. Instances of these classes replace the default *ConcurrentHashMap*-objects which were used to save information in memory. Since methods in classes *StoreMultiMapMemory* (trough which

articles' map is accessed) and *StoreMemoryCumulative* (which manages the access to index data map), only interacted with a generic *Map* object, there was no need in modifying them.

While the memory-based implementation used a concurrent hash map to support concurrency of retrievals and for updates, for the disk-based implementation it is assumed that the DBMS takes care of concurrency, so that there is no need to manually synchronize the access to the database tables.

3.4 Keyword Relevance

3.4.1 Motivation

To test the correct running of the P2PFastSS system and to allow a simple view of the search results, a simple web interface is used; it allows the user to select XML files for indexing, to perform similarity search and to display the results. However, the results weren't sorted: they were displayed in the order the various nodes answered to the search queries. This caused situations in which results with an exact match (with edit distance 0) were displayed after some results with larger edit distance. In order to sort results by relevance, a numerical value representing the relevance of a keyword in an article has to be calculated.

3.4.2 Design

A "relevance function" was formulated, which assigns, at indexing time, a numerical value to each deletion neighbor calculated. The factors that influence the relevance score are edit distance, whether the keyword appeared in the articles'title. The relevance score is saved with each generated deletion neighbor and is used as the base for sorting the search results before showing them on the web interface.

The list below enumerates some important properties that have impact on a keyword's relevance:

- Edit distance (the lower, the greater relevance),
- number of occurences of the keyword in the article,
- length of the keyword (in characters),
- total words contained in an article,
- occurence in article's title (if the keyword appears in the title the score should be increased).

The simple function which combines all those factors is the following:

$$r(k, a) = \frac{C_1}{(ed + 1)} + C_2 \cdot T + C_3 \cdot \left(\frac{o}{w}\right) \cdot L^{C_4}$$

where $r(k, a)$ is the relevance score of keyword k in article a, C_i positive real constants, ed the edit distance ($ed \geq 0$ and integer), T number of keyword's occurences in title ($T \geq 0$ and integer), o number of keyword's occurences in text ($o \geq 0$ and integer), w number of all words in the article ($w \geq 0$ and integer) and L the length of the keyword k ($L \geq 1$ and integer).

By analyzing the function the following properties can be observed:

- The smaller the edit distance, the bigger the score, because $\frac{C_1}{(ed + 1)}$ is maximal when $ed = 0$ and decreases with growing ed.
- If the keyword appears in the title, the score will increase.
- The more often the keyword occurs in the article's text, the larger the score, because *o/w* grows with encreasing *o*; *o/w* also implies that more words an article has, the smaller the score's growth.
- The L^{C_4} factor was added to decrease the score of short words which can easily have many occurences in an article, such as the keyword "the". From two keywords with the same number of occurences, the longer one receives a bigger increase in its score.

3.4.3 Implementation

Since the application must store, with each deletion neighbor of an indexed word, its relevance score in an article, the *KeyAndKeyWord* class is modified to carry the additional relevance variable. The relational schema for the index data is also updated, with the *relevance* attribute, as Table 6 shows.

Table 6: Updated relational schema with relevance attribute

INDEX	hash	originalWord	articleTitle	relevance

3.5 Paging of Search Results

3.5.1 Motivation

In the previous version of the web interface, all the results were displayed in a single page. Whenever a search query produced many search results, this page would result higher than the user's screen, so he or she would have to scroll the page to check all the results. If the results had to be verified one at a time, then the user had to visually keep track of the last verified result, which could be tedious and frustrating for the user. Another issue was the results page loading time, where a page displaying many results would take more time to be visualized.

For this reason a paging feature was implemented, so that the search results would have ben separated into multiple sets.

3.5.2 Design

Only one set of results would be displayed on the results page at a given time, and the user can decide to navigate back and forth trough the sets. The sets would have to be sorted by relevance so that those displayed first would contain the results with higher relevance score then those displayed later. The user browses the search results by selecting the *previous* or *next* link. Navigation trough

the results shouldn't imply new search queries. The first page of search results doesn't show the *previous* link, while the last page doesn't show the *next* link. A default number of results per page is predefined and if the result set is smaller or equal to it, both links are not displayed.

3.5.3 Implementation

To obtain the sorting of search results, they need to have an ordering relation. The most natural ordering relation is their keyword relevance score. By the time this particular implementation has begun, the keyword relevance function was already implemented and available. But the *KeyAndKeyWord*-objects still couldn't be compared between each other.

To make them comparable, it was decided to modify *KeyAndKeyWord* class so that it would implement the *Comparable* interface. Consequently, the method *compareTo*() is implemented, making *KeyAndKeyWord*-objects comparable by relevance. A new collection type is used for containing the results of search queries to avoid manual sorting. The Java class *TreeSet* is chosen for this task because it automatically sorts the elements it contains by using their *compareTo*() method, and updates the order if needed when elements are added or removed.

Finally, the code for submitting queries was logically separated from the code responsible for displaying the results. The results of a search query are temporarily saved in a *TreeSet* object and this object is then iterated to obtain the results for the displayed page. Iteration of the *TreeSet* doesn't produce new search queries or additional network traffic.

3.6 Preview of Search Results

3.6.1 Motivation

The visual output for the search results displayed in the web interface consisted only of the article's title and a numerical hash value of it. If the title wasn't informative enough for the user, he had to open several articles, until the searched article was found. If the articles weren't on the node that user was using, then the node had to fetch many of them from the DHT, thus generating network traffic. It is assumed that, if the user was given more information about the search results, he or she would identify the searched article (or other type of resource) faster and it would generate less network traffic.

3.6.2 Design

To present better contextual information to the users, each result must include a segment containing the keyword in the article. It was decided to extend the software so that when a keyword is indexed, its position in the article is also to identified and a short segment of text containing the keyword, defined as the *preview string*, is saved together with the index results.

3.6.3 Implementation

The *KeyAndKeyWord* class is furthermore extended to allow saving of the preview string inside of it. At index time, for each generated deletion neighbor, a portion of the article's text, containing the

original keyword and a given number of surrounding words from the left and from the right side, is saved as the preview text. To allow the preview string to be saved on disk, the database table for index data is extended. Table 7 shows the final state of the relational schema for the index data.

Table 7: Final relational schema for index data

INDEX	hash	originalWord	articleTitle	relevance	preview

The web interface is modified to show the preview text of the article, in addition to the article's title. Before actual displaying, the search results are processed, the keyword is identified in the preview string and HTML bold tags are applied to it, as Figure 4 shows, to make the keyword even more visible to the user.

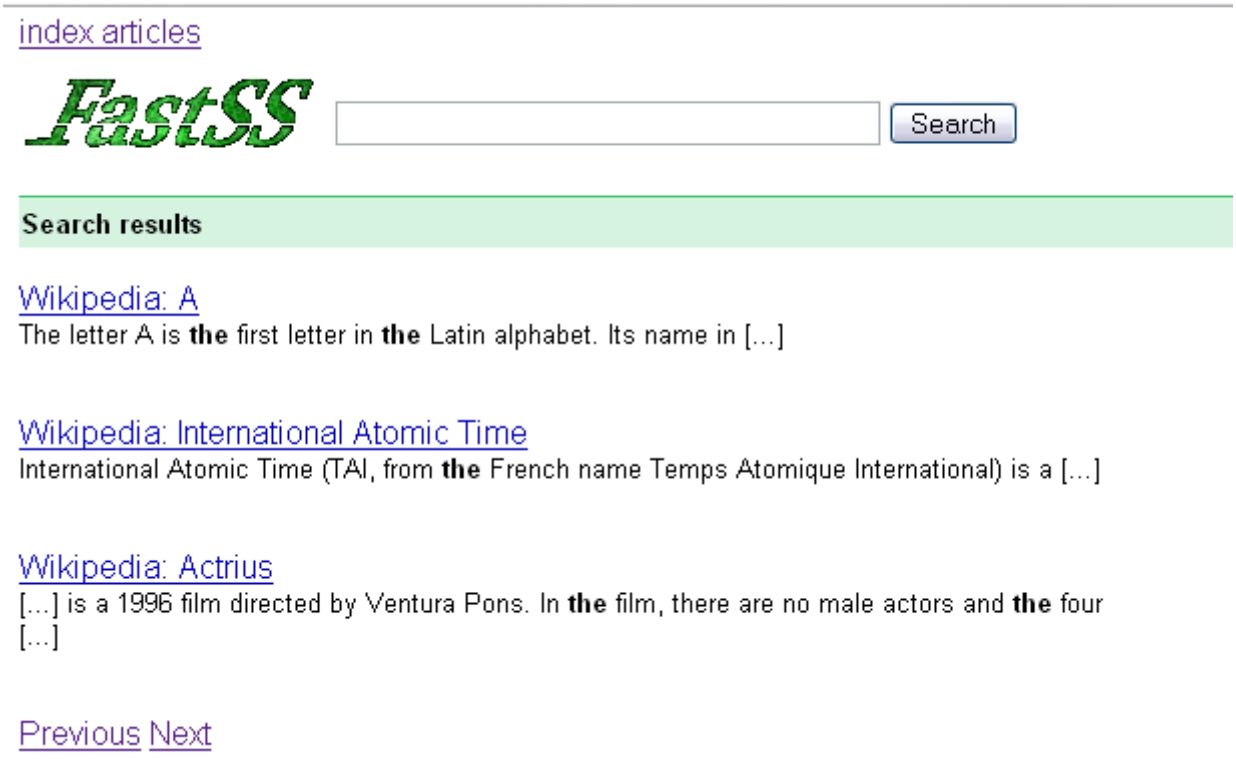

Figure 4: P2PFastSS web interface displaying some search results

4 Conclusion and Future Work

This chapter concludes this thesis with a short summary of this assignment work. It also gives some final statements about the results and achievements of objectives, pointing out possible weaknesses. Ultimately, it indicate possible research areas for future work.

4.1 Summary

First chapter of this thesis motivated the work by describing the problem and defining the assignment goals. Second chapter gave an overview of related work in research areas of P2P networks and similarity search. Third chapter started with a detailled description of P2PFastSS implementation, providing necessary knowledge to understand design choices and implementation details of new features, such as distributed indexing, disk-based storage, keyword relevance and improvements to the web interface, which followed.

4.2 Conclusion

Distributed indexing for P2PFastSS was implemented and tested. In the implementation, one node parses the XML file and sends the extracted documents to responsible nodes via DHT-*put* operation. When documents are saved on those nodes, a special thread is dedicated for indexing them and for storing the index data. Testing shows that document storing and indexing tasks are done in parallel, and not sequentially as before the assignment.

The implemention presented in this thesis uses database tables to store original and indexed data to disk, instead of in memory. It provides the P2P system even more fault tolerance, by allowing the system to resume from a given time point after a shutdown, or a crash, without having to re-index the data or resend the documents. This is particularly useful for testing new features, because it eliminates the need to re-process the data before being able to try a new feature. One current issue is that the tables with data are saved to disk using the node ID as file identifier. This means that a failed node has to re-obtain its old node ID, in order to acquire access on the data tables. For this reason, a non-random node ID allocation is advised when using P2PFastSS with disk-based storage.

Furher improvements to the system have been made, such as defining a keyword relevance function which numerically characterizes a keyword's importance in the document that contains it, and can be used to sort the search results.

Finally, a more user-friendly displaying of search results has been implemented in the existing web interface, by allowing the paging of the results and by showing a preview of the document containing the match.

4.3 Future Work

Altough the current P2PFastSS system now uses nodes in parallel for data indexing, simulations and tests need to be carried out. Future work should address this question by measuring in both versions, for example, the time needed to process a fixed quantity of articles. But given the number of changes the system underwent and the new features that were added, it is probable that the two versions are now too dissimilar to be comared. It could be difficult to determine the right causes of eventual performance gain or loss.

A possibility for integrating algorithms dealing with key relevance problem could be researched. In particular the Information Retrieval scientific literature could be consulted, finding the best ways of estimating the relevance and evaluating their integration in Peer-to-Peer Fast Similarity Search

References

[1] D. Eichhorn: *A Peer-to-Peer Network Framework with Network Address Translation Traversal*. http://www.csg.uzh.ch/staff/waldburger/extern/theses/DA_Dani_Eichhorn.pdf; Departement of Informatics, University of Zurich, May 2006.

[2] T. Bocek, E. Hunt, B. Stiller: *Fast Similarity Search in P2P Networks*. http://fastss.csg.uzh.ch/PID557494.pdf; Dipartement of Informatics, University of Zurich, April 2008.

[3] G. Navarro: *A guided tour to approximate string matching*. ACM Computing Surveys (CSUR), v.33 n.1, p.31-88, March 2001.

[4] T. Bocek, E. Hunt, B. Stiller: *Fast Similarity Search in Large Dictionaries*. http://fastss.csg.uzh.ch/ifi-2007.02.pdf; Departements of Informatics, University of Zurich, April 2007.

[5] T. Bocek, E. Hunt, B. Stiller, F. Hecht: *FastSS - Fast Similarity Search in Large Dictionaries*. http://fastss.csg.uzh.ch/; Last visited February 2008.

[6] J. D. Gradecki: *Mastering JXTA - Building Java Peer-to-Peer Applications*.Wiley Publishing, Inc., Indianapolis, September 2002.

[7] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim: *A Survey and comparison of Peer-to-Peer Overlay Network Schemes*. Communication Surveyes & Tutorials, IEEE, p.72-93, 2005.

[8] R. Steinmetz, K. Wehrle: *Peer-to-Peer Systems and Applications*. Lecture notes in Computer Science. Springer, October 2005

[9] A. W. Loo: *Peer-to-Peer Computing - Building Supercomputers with Web Technologies*. Springer, 2007.

[10] B. Stiller, J. Gerke, T. Bocek, D. Hausheer: *To Peer or not to Pear?*. Research retreat, Sonthofen, Germany, March 2006.

[11] P. Backx, T. Wauters, B. Dhoedt and P. Demeester: *A comparison of peer-to-peer architectures*. Eurescom Summit, 2002.

[12] C. H. Ding, S. Nutanong and R. Buyya: *P2P Networks for Content Sharing*. Department of Computer Science and Software Engineering, The University of Melbourne, 2004.

[13] C. Wang and B. Li: *Peer-to-Peer Overlay Networks: A Survey*. Departement of Computer Science, The Hong Kong University of Science and Technology. April, 2003.

[14] D. Andersen, H. Balakrishnan, F. Kaashoek and R. Morris: *Resilient Overlay Networks*. MIT Laboratory for Computer Science. 2001.

[15] J. Janotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek and J. W. O'Toole, Jr.: *Overcast: Reliable Multicasting with an Overlay Network*. Cisco Systems, 2000.

[16] D. S. Milojcic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins and Z. Xu. *Peer-to-Peer Computing*. HP Laboratories Palo Alto. March, 2002.

[17] S. Staab and H. Stuckenschmidt (Eds.): *Semantic Web and Peer-to-Peer*. Springer, 2006

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker: *A Scalable Content Addressable Network*. ACM SIGCOMM 2001, San Diego, California, August 2001.

[19] A. Rowstron and P. Druschel: *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp 329-350, November 2001.

[20] P. Maymounkov and D. Mazieres: *Kademlia: A peer-to-peer information system based on the XOR metric*. In Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, USA, March 2002.

[21] I. Stoica, R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan: *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications*. ACM SIGCOMM 2001, pp 149-160, San Diego, California, August 2001.

[22] H. Feistel: *Cryptography and Computer Privacy*. Scientific American, Vol. 228, No. 5, 1973.

[23] V. Levenshtein: *Binary codes capable of correcting spurious insertions and deletions of ones*. Probl. Inf. Transmission 1, 8–17. 1965.

[24] D. Sankoff and J. Kruskal (Eds).: *TimeWarps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA. 1983.

[25] G. Das, R. Fleisher, L. Gasieniek, D. Gunopulos and J. Kärkäinen: *Episode matching*. In Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM '97). LNCS, vol. 1264, Springer, 12–27. 1997.

[26] A. Apostolico, and C. Guerra: *The Longest Common Subsequence problem revisited*. Algorithmica 2, 315–336. 1987.

[27] V. I. Levenstein: *Binary codes capable of correcting insertions and reversals*. Sov. Phys. Dokl., 10:707–10, 1966.

[28] G. Navarro: NR-grep: *A Fast and Flexible Pattern Matching Tool*, Technical Report TR/DCC-2000-3. Technical report, University of Chile, Departmento de Ciencias de la Computacion, Santiago, 2000. http://www.dcc.uchile.cl/~gnavarro.

[29] D. Gusfield: *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[30] E. Myers: *A sublinear algorithm for approximate key word searching*. Algorithmica, 12(4/5):345–374, 1994.

[31] A. Califano and I. Rigoutsos: *FLASH: A Fast Look-up Algorithm for String Homology*. In ISMB, 1993.

[32] R. Huebsch, B. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica and A. R. Yumerefendi: *The Architecture of PIER: An Internet-Scale Query Processor*. The Second Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, January 2005.

[33] G. Skobeltsyn, T. Luu, I. P. Zarko, M. Rajman and K. Aberer: *Web text retrieval with a P2P query-driven index*. SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, Amsterdam, The Netherlands, pp 679-686, 2007.

[34] M. Bawa, T. Condie and P. Ganesan: *LSH forest: self-tuning indexes for similarity search*. WWW '05: Proceedings of the 14th international conference on World Wide Web, Chiba, Japan, pp 651-660, May 2005.

[35] I. Aekaterinidis and P. Triantafillou: *Substring Matching in P2P Publish/Subscribe Data Management Networks*. IEEE 23rd International Conference on Data Engineering (ICDE), pp 1390-1394, April 2007.

[36] M. Karnstedt, K. Sattler, M. Hauswirth and R. Schmidt: *Similarity Queries on Structured Data in Structured Overlays*. ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops, April 2006.

[37] W. Litwin, R. Mokadem, P. Rigaux and T. Schwarz: *Fast nGram-Based String Search Over Data Encoded Using Algebraic Signatures*. Very Large Data Bases (VLDB) 2007, Vienna, Austria, September 2007.

[38] R. Ahmed and R. Boutaba: *Distributed Pattern Matching: A Key to Flexible and Efficient P2P Search*. IEEE Journal on Selected Areas in Communications, Volume 25, Issue 1, pp 73-83, January 2007.

[39] B. H. Bloom: *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM 13 (7): 422–426, 1970

[40] JUnit, Testing Resources for Extreme Programming, URL: http://www.junit.org/index.htm, last visited: February 2008.

[41] GNU General Public Licence, URL: http://www.gnu.org/copyleft/gpl.html, last visited: February 2008.

[42] HSQL Database Engine, URL: http://hsqldb.org/, last visited February 2008.

List of Figures

Distributed P2P Application running on top of a DHT 6

P2PFastSS web interface indexing page 13

P2PFastSS web interface search page 13

P2PFastSS web interface displaying some search results 20

List of Tables

P2P and C/S - Benefits and Drawbacks 4

Example of the avalanche effect 6

Relational schema for storing articles 16

Relational schema for storing index data 16

Benefits and drawbacks of File Systems and Database Management Systems 16

Updated relational schema with relevance attribute 18

Final relational schema for index data 20

Appendix A

Contents of the CD

- Collection of related work papers in PDF
- Source code of the new version of P2PFastSS
- This assignment thesis as PDF and PS files
- FrameMaker source of this thesis
- Figures in source files