Thesis no: BCS-2015-07

An Implementation and Performance Evaluation of a Peer-to-Peer Chat System

Simon Edänge

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author:
Simon Edänge
E-mail: sied10@student.bth.se
Tel: +46 730 66 43 44

University advisor:
Professor Kurt Tutschku
(DIKO) Department of Communication Systems

University examiner:
Prefect Veronica Sundstedt
(DIKR) Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Context: Chat applications have been around since the beginning of the modern internet. Today, there are many different chat systems with various communication solutions, but only a few utilize the fully decentralized Peer-to-Peer concept.

Objectives: In this report, we want to investigate to see if a fully decentralized P2P concept is a suitable choice for chat applications. In order to investigate, a P2P architecture was selected and a simulation was implemented in Java. The simulation was used to make a performance evaluation in order see if the P2P concept could meet the requirements of a chat application, and to identify problems and difficulties.

Methods: Two main methods were used in this thesis. First, a qualitative design method was used to identify and discuss different possibilities of designing a distributed chat application. Second, a performance evaluation was conducted to verify the selected and implemented mechanisms are able to obtain their general performance capabilities and to tune them towards anticipated performance.

Results: The simulation proved that a decentralized P2P system can scale and find resources in a network quite efficiently without the need of any centralized service. It also proved to be simpler for the user to use the P2P concept, as no special configurations are needed. However, the selected protocol (Chord) had problems with high rates of churn, which could cause problems in big chat environments. The P2P concept was also shown to be highly complex to implement.

Conclusion: P2P technology is a more complex technology, but it gives the host a lower cost in terms of hardware and maintenance. It also makes the system more robust and fault-tolerant. As we have seen in this report, P2P can scale and find other resources efficiently without the need of a centralized service. However, it will consume more power for each user, which makes mobile devices bad peers.

Keywords: P2P, Implementation, Chat System, DHT

Contents

1 Introduction

This report is focusing on the Peer-to-Peer technology within chat applications, and how such system could be implemented. The report will go through the Peer-to-Peer concept, comparing its advantages and disadvantages with the popular Client-Server concept. Three famous Peer-to-Peer protocols will be discussed and used as examples to illustrate different approaches of a network application (Chapter 4). After this Chapter, we will go through the requirements of a chat application (Chapter 5), the definition of Peer-to-Peer performance (Chapter 6) and then finally the actual implementation and performance evaluation (Chapter 7).

It has been an interesting journey to the completion of this project. The implementation of the Peer-to-Peer simulation application proved more complex to implement than excepted, but very informative and rewarding in the end. The Chord protocol, which was used in this implementation, is a fascinating protocol with self-organizing capabilities and efficient lookup functionality, which was also one the main reasons it was chosen. The chord protocol can be read more in depth in section 4.4.1.

1.1 Motivation

The idea behind this project was to implement a decentralized Peer-to-Peer application based on a promising protocol that fulfills the requirements of a chat application. The application's main purpose was to identify problems and difficulties with Peer-to-Peer systems and to evaluate and verify if the Peer-to-Peer concept is a suitable choice for a chat application.

There are many possible communication protocols to choose from. In this report we mention three different protocols in detail, which have contributed a lot to the research of the Peer-to-Peer technology. The Chord protocol was chosen for the implementation of this simulation application and is explained more in depth, see section 4.4.1.

We cannot justify the Chord protocol to be the best decentralized Peer-to-Peer solution for a chat application, but we can say that it could be a promising solution in terms of its efficiency, reliability, simplicity and the fast peer lookup functionality it offers.

1.2 Background

Many network applications today use the Client-Server based protocol and more developers choose to dismantle the Peer-to-Peer technology. There are various reasons for this and some are mentioned in this report. Microsoft decided to dismantle the Peer-to-Peer technology from Skype due to the increased use of mobile devices [1], [2]. They are generally slower than computers and may lose connection more often which could cripple the performance (section 5.2.3). Another reason could be that the cost of hardware has become cheaper over the years. Nevertheless, Peer-to-Peer technology is still used by many applications and in some cases is preferable: e.g. file sharing applications [3].

As for chat applications, there are not many in existences that use a decentralized Peer-to-Peer solution. The question is whether the decentralized Peer-to-Peer concept is an appropriate architecture for a chat application. This report is an attempt to address this question.

1.3 Problem definition

Today it is fairly easy to implement a simple Client-Server based chat application. The location of a client cannot be known for sure due to dynamic IP addresses. The server can store the location of every connected client, thus provide a lookup service database. The only thing a client has to do is to ask the server in order to find other clients and create a direct connection to them. This is very similar to the Napster protocol, see section 4.5.1.

The Client-Server example above is actually a mix of the Peer-to-Peer concept, as the clients have to establish direct connections with each other to communicate. However, to implement a simple chat application with the Peer-to-Peer concept, without any central server that provides a lookup service, is more complex. The fact is that the location of anyone is not known for sure; so to provide a lookup service in such system requires the peers to ask other connected peers. But which peer to ask, and which peer should be connected to which peer? Every peer in a Peer-to-Peer system has to organize themselves in order to find each other or maintain connectivity to the network, which could be a challenging task to implement. The lookup service for Peer-to-Peer systems is explained in section 4.2.

1.4 Research questions

As we want to see how a decentralized Peer-to-Peer system performs in chat applications, we have to investigate the concept. Below are research questions this report will focus on:

- How can one design and implement a chat application that does not rely on any centralized lookup service, and which class of lookup algorithms is capable to meet the key feature of decentralization?
- How can one analyze the performance of a decentralized chat system before it is deployed?
- How can one verify that the general search time of a DHT-based lookup algorithm is of order of $O(log\ N)$, with N is the number of peer/users?

Performance Evaluation: *This section defines the goals of our performance evaluation study (1). This refers to the methodology chosen for this thesis (cf. Performance Evaluation in section 3.2).*

2 Related work

When thinking of the Peer-to-Peer technology, we are entering a big world with numerous and different implementations and architectures. This report will introduce a few Peer-to-Peer systems in order to describe different Peer-to-Peer protocols and to get a better understanding what the technology really is. There are many systems today that utilize this technology, especially in file sharing applications: e.g. BitTorrent, as of today uses 40% of the world's internet traffic on a daily basis [3].

The BitTorrent protocol adopted the *Distributed Hash Table* (DHT) technology [4], which we used in our implementation. The Chord protocol we used in our implementation is one out of the four original DHT protocols: *Chord* [5]*, Tapestry* [6]*, Pastry* [7] *and CAN (Content Addressable Network)* [8], which made DHT a popular research topic back in 2001. These protocols have different implementations and algorithms, but they share the same concept. In terms of performance, they are very similar but Chord is one of the most researched DHT protocol.

Chord and the other three DHT protocols was an attempt to make decentralized P2P systems scalable and to provide a fast lookup functionality. They were originally motivated due to the lack of scalability and poor lookup service in earlier distributed P2P systems. The creators of Chord concluded that the DHT protocol was able to scale and had a lookup efficiency of $O(log\ N)$ based on their simulation tests [5].

In 2003, H. Zhang et al. [9] attempted to improve the lookup latency in DHT systems. They choose to slightly modify the Chord protocol to simulate their sampling technique called *lookup-parasitic random sampling* (LPRS). The technique basically tries to build a node's routing table, which contains nodes with a low unicast latency, thus reduce the latency when performing lookups. Their simulation revealed a qualitatively better latency scaling behavior than the original Chord Implementation.

Another paper by J. Li et al. [10], focused on comparing the lookup performance of four different DHT protocols (Chord, Tapestry, Kelips and Kademlia) under churn. They identified various parameters for each protocol that affects cost and performance, and conducted simulation tests for each protocol. Their conclusion was that the protocols can achieve similar performance, if the parameters are sufficiently well-tuned, but it is a delicate business. The same simulation were also used by D. Wu et al. [11] together with an analytic study on how to improve the DHT lookup performance under churn.

B. Leong et al. [12] proposed a method of achieving one-hop lookups in a DHT network, but in a more cost of bandwidth. They make use of a token-passing technique that efficiently broadcasts events in form of small messages to all nodes in the network. For example, when a node joins, it passes a join token that will go around the network. The token consist of information of the node, such as IP address that each node stores in a "B-Tree". A simulation was conducted to analyze the bandwidth consumption and comparing their results with a previous work by Gupta et al.'s one-hop routing scheme [13]. They concluded that the bandwidth consumption is rather moderate, but in a network with about a million nodes it is quite sizable.

The above work focuses on evaluating and improving the performance of DHT networks. Many of them choose to implement the system as a simulation to test their goals, often within the "lookup service" feature, which is related to our thesis. Our study is interested/focused in the "lookup service" function in the "performance term of hops/time for successful searches".

3 Methodologies

In order to find an appropriate design for a network and software architecture for a distributed chat application, we apply in this thesis two main design methods: first, a qualitative methodology for system design and mechanisms selection based on a separation of concerns and second, the use of performance evaluation to verify that the selected and implemented mechanisms are able to obtain their general performance capabilities and to tune them towards anticipated performance.

3.1 Qualitative Design of an Network and Software Architecture for a Distributed Chat Application

3.1.1 Overall Concept

This thesis applies first a qualitative design method for the definition of a network and software architecture for a chat application, which is based on the specification and separation of required functional features of the application's mechanisms and services.

In detail, the suggested method applies a separation concept for features, which is similar to the idea of "separation of concerns" in software design [14]. In software design, a "concern" is a set of information that affects the code of a computer program. Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface.

In networked applications, the above introduced notion of "code" is enhanced to the term "component". A component might comprise lines of code or/and a network structure or topology. The later mean, for example, that a component might describe the application layer network topology of the networked applications. The components are well-separated sections of a networked system that individually can be developed, implemented, updated or reused independently.

For the purpose of a networked application, at least three main areas of concern can be defined:

- Required services and functions: this area describe the required services and needed functions such that application logic is able to achieve the usage goals of the application.
- Application-layer topology and degree of centralization: this area specifies the topology of the application-layer relationship between the networked applications.
- Autonomy of mechanisms: this area specifies the dependency of the mechanisms of the application on information available of places in the network.

The adjective of "qualitative" decision in the architecture design, describes that the design decision is based on the availability (or none-availability) of this feature in the considered mechanism, rather than deciding whether this feature is fully, partly, efficiently, or for high-performance implemented. The later characterization is typically considered a quantitatively based decision.

3.1.2 Areas of Concern of a Distributed Chat Application

The above outlined areas of concern are rather general and need to be refined with respect to the considered distributed chat application. Next, we will discuss the interpretation of these area and refine them:

(a) Required services and functions:

A chat application need provide two major functions for the user. First, the application should enable a user to find the location of other users within a network. We call this function a "lookup service". Second, the application needs to be able to send a message in a point-to-point manner from the sender to the recipient and to display the message. We consider the later task a trivial and easily available in today's operating systems and network stack, e.g. a network socket for send a packet from a sender to a recipient or as an operating system call to display a message in a window system.

(b) Application-layer topology and degree of centralization:

Various kinds of network topologies can be chosen that might have a different degree in centralization. The Network topology together with the application-layer illustrates how data logically

flows within a network, usually arranged with various elements such as links (paths) and nodes (users). When a packet is sent from a node through a link to a destination, a hop occurs. For every node/entity this packet reach, the hop count is incremented. However, in the physical world the hop count is much larger due to physical devices in between, such as routers. Large number of hops implies lower performance.

(c) Autonomy of mechanisms:

In distributed systems, nodes usually wants to request a resource or information from other nodes/entities. When a node has requested a resource, it waits for a response. But considering anyone can leave the network at any given time, there will be no response and thus causing a deadlock. This can be prevented by using stateless protocols, which treats each request as an independent event that is unrelated to previous requests [15].

We will describe and discuss possibilities to implement these areas of concern in Chapter 4 in the thesis.

3.2 Performance Evaluation

The second method for application design applied in this thesis is performance evaluation and performance verification.

This method applies a well-established 10 step process for obtaining performance values, cf. Box 2.2 in [16].

1. State the goals of the study and define the system boundaries.
2. List system services and possible outcomes.
3. Select performance metrics.
4. List system and workload parameters.
5. Select factors and their values.
6. Select evaluation techniques.
7. Select the workload.
8. Design the experiments.
9. Analyze and interpret the data.
10. Present the results. Start over, if necessary.

We will describe and discuss how these process steps are carried out. Evaluation methods and parameters have been selected in section 4.2.

Different ways of studying a system is explained in the book “Simulation Modeling and Analysis” by A. M. Law and W. D. Kelton [17]. The book mention multiple ways of performing an experiment on a system once it has been defined, cf. Figure 1.

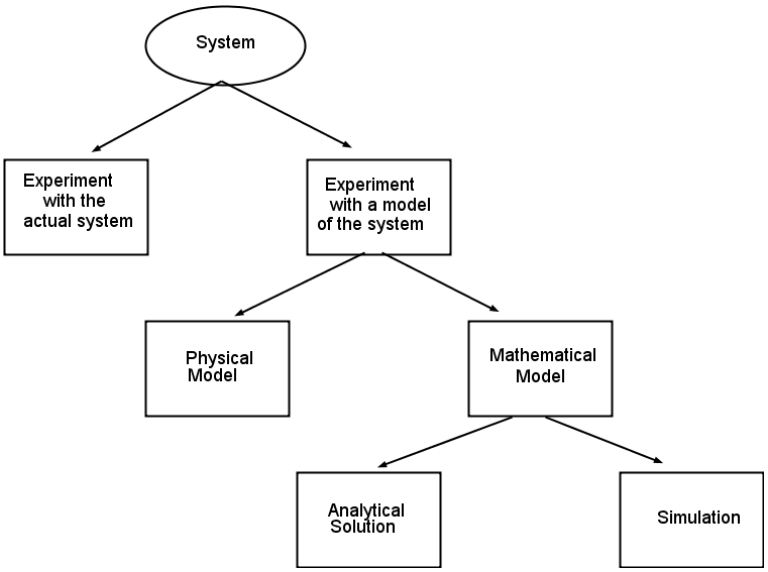

Figure 1: Ways to study a system **[17]** (Figure 1.1).

One could experiment with the actual system to test the performance and identify weaknesses. It would give accurate data, but this solution is often impractical due to the high cost of time and resources. In addition, if the system does not yet exist, it may not even be possible. I these cases, one could develop a smaller partition of the system as a physical model, if it is feasible or sufficient to do so. If not, a mathematical model may be created.

The mathematical model has two ways, either an analytical or a simulation model. The analytical model provide more exact solutions than a simulation and is more of a theoretical solution. However, real world systems are usually too complex to use realistic models and evaluate them analytically. In this case, a simulation would be more preferable. Simulation is a replication of the computer model's behavior of a real system, and is a feasible solution when the application model is too complex.

For this thesis, a simulation is a more feasible evaluation technique, due to the limited time and the high complexity of P2P systems. This will be used in this thesis as an evaluation technique.

Performance Evaluation: *Simulation was chosen as evaluation technique (6).*

4 Peer-to-Peer Technology

The Peer-to-Peer (i.e. P2P) technology has been around for some time and a lot of research has been done in the area [18]. P2P is a network application architecture, mostly used for file sharing between computers or mobile devices. In P2P, the work is distributed amongst the peers, which means there is no server doing all the heavy work like in the Client-Server architecture. The P2P technology can offer more robustness, fault-tolerance and could be an inexpensive choice (overall cost of maintenance) compared to the more common Client-Server architecture.

4.1 Client-Server vs. Peer-to-Peer

Many network applications today are based on the Client-Server model. In this model, the server works as a centralized resource service, providing information to the connected clients. The clients typically do not have any direct connection with each other, but instead communicate with the server to get information about other clients and resources (cf. Figure 2). The Client-Server model gives more control to the service provider because the server knows who is connected to the network and what information are stored in the system. But there are some drawbacks. In a system with many clients, the Client-Server is an expensive choice. To guarantee good performance, the server needs to have fast hardware to be able to maintain a stable throughput with thousands or perhaps even millions of users [18, p. 9]. In some cases, one server is not good enough. Big systems like *Facebook* have thousands of servers to serve millions of people around the globe. Using this model to serve a large quantity of users requires better and faster hardware, which will result in a higher cost. Another drawback is that the clients are dependent upon the server. If it goes down, the system will not function properly, thus the connected clients will be affected by it.

As mention before, P2P systems and applications are distributed in a decentralized way [18, p. 57], which means there is no server that does the heavy work. Information about resources and other entities are distributed amongst the peers. To find another user or a resource, every peer maintains a connection with x other peers (*neighbors*). These neighbors are used as routing paths to provide *lookup* functionality [18, p. 269]. Because there is no centralized service, the P2P concept is more fault-tolerant and also more inexpensive due to the distributed work. However, routing messages through peers with a slow internet connection could cripple the system, and a neighbor peer is not guaranteed to be online. The peer could have left the network without notifying, or the IP address may have been changed due to dynamic IP address.

Autonomy and autonomous behavior, e.g. stateless protocols or self-organization, are key characteristics of P2P systems (cf. concern (c) "*Autonomy of mechanisms*", section 3.1.2). The P2P concept is fulfilling the "stateless et al." requirement and therefore we decided to consider a P2P concept.

A pure P2P concept, like DHT (*Distributed Hash-tables*), but unlike Napster/eDonkey/BitTorrent, is the foundation for any service with absolutely no infrastructure (cf. concern (b) *"Application-layer topology and degree of centralization"*, section 3.1.2). Pure P2P concepts fulfills the "infrastructure-less" requirement, thus a pure P2P concept needs to be considered.

In Figure 3 we can see a comparison of P2P systems by its architectural characteristics to get a better understanding on how the resources are maintained in different systems and how different they are from each other.

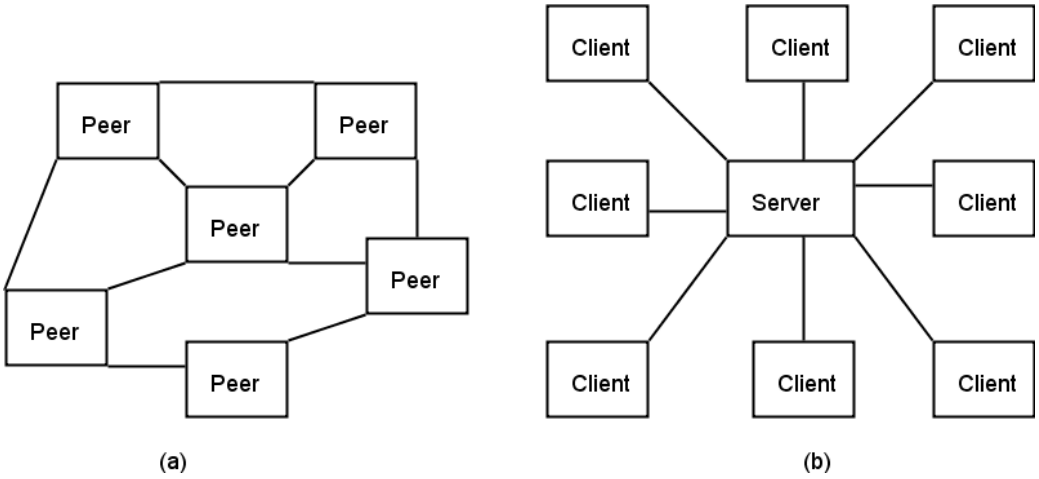

Figure 2: (a) An illustration of a P2P network. (b) An illustration of a Client-Server network.

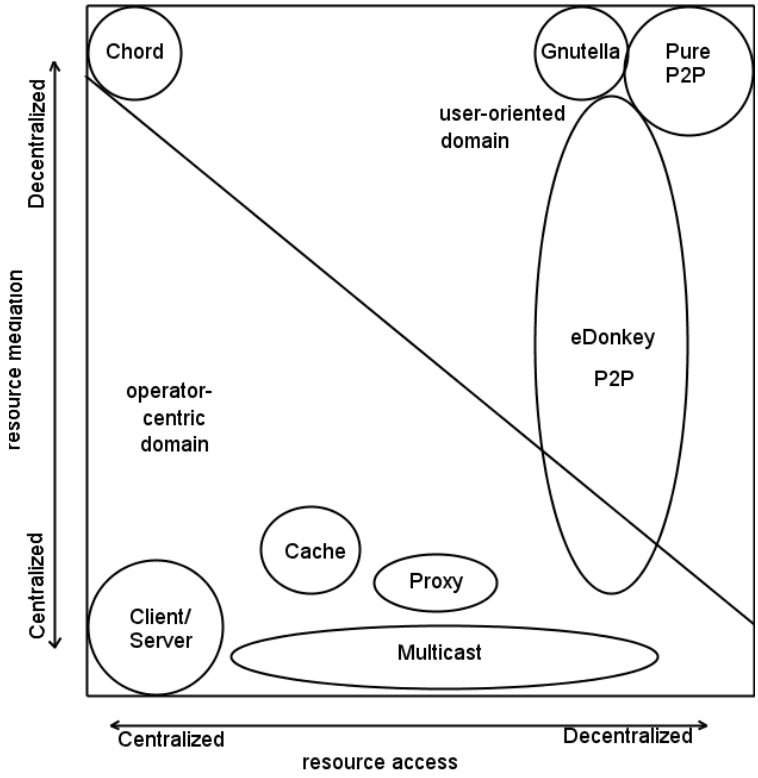

Figure 3: Shows a two dimensional cartography of P2P applications and content distribution architectures. This graph can be seen in **[18]** (Figure 23.1).

4.2 The 'Lookup Service'

Today, users lay strong emphasis on applications to be "mobile". That means that the users want to use their devices and applications while they are moving or after a relocation of the device to another network (the later feature is also known as "*roaming*"). In addition, the frequency of a user of being mobile or roaming around has tremendously increased. Hence, it must be assumed that a direct communication between users, which relies on static IP addresses (i.e. addresses that are simultaneously identifier and locator), is not possible anymore. Hence, the IP address of a user has to be looked-up every time before a message is sent to a user. Moreover, users nowadays prefer nicknames, i.e. self-selected names, since they are much easier to remember than IP addresses. In addition, the nickname might change overtime and should be easily changeable by the user itself.

As a result, for a network or an application to consider mobility they require a service function, which relates the current location and IP address of a user (aka "*locator*") to its nickname ("*identifier*"). We call this function in this thesis "*Lookup Service"*. This service will provide a locator for a given identifier. Unfortunately, IP networks typically do not provide such a lookup function. Hence, every application that would like to support mobility has to implement such a function on its own. Of course, this function has to be tailored to the specific needs of the application, e.g. it needs to be distributed if a central server is not available.

Mobility management and the split between identifiers and locators are important problems in today's network and distributed systems design, with a number of solutions currently being discussed [19]. In this work we will consider, discuss, analyze and implement a distributed P2P based solution for such a Lookup Service. In general, architectures for lookup services can be classified (cf. Figure 4) either into centralized (e.g. Cloud technology) or decentralized implementations (e.g. P2P-based solutions).

The lookup service is an important key function for distributed networks to find and locate other entities/resources (cf. concern (a) *"Required services and functions"*, section 3.1.2). Locating users within chat applications is also an important requirement (cf. section 5.1).

Performance Evaluation: *This section characterize the boundaries of the system (1) and the service that will be focused (2).*

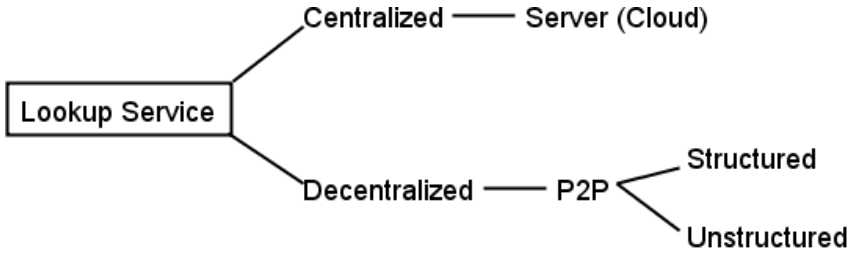

Figure 4: Choices for the architecture of a "Lookup Service" **[19]**.

4.3 Unstructured Networks

In unstructured P2P networks, the peers joining the network are placed randomly in the systems topology overlay [19]. This makes the system more dynamic when entities join and leave the network and is less of a burden for the programmer. But because of lack of structure, if a peer wants to search for data in a decentralized network, an unstructured network has its limitations. The peer has no clue where the data might be stored. To find it, the peer sends search queries to all its neighbors and its neighbors forward it to their neighbors and so on. This creates a lot of traffic and there is no guarantee the queries will be resolved.

4.3.1 Gnutella

Gnutella is a good example of a decentralized, unstructured P2P system [20]. No central element keeps track of the data like in Napster [21] (section 4.5.1). Gnutella is a large pool of nodes communicating with neighbors (cf. Figure 5). At least one other node has to be known to be able to be part of a network. The term node is commonly used to describe a peer when discussing the overlay topology of a P2P protocol. The good thing about Gnutella, it is very dynamic and robust. Peers joining and leaving is not a big issue and is easy to maintain. The biggest issue is when a peer is searching for e.g. a file in a network. It uses an algorithm called *flooding*. With flooding, it sends queries to neighbors, which then routes the message to their neighbors and so on. This process continues until the data has been found or if the *TTL* (*Time to Live*) is over. This creates a large amount of messages in the network and the searching could take some time if there are many nodes. If nodes are too far away, TTL will end before it reaches the target node, which means all peers cannot be reached. When the file sharing giant Napster was shut down in 2001, due to being accused for spreading pirate copied data [22], the swarm of Napster users moved to Gnutella instead. The result was a total system collapse and proved that Gnutella system has scalability problems [23].

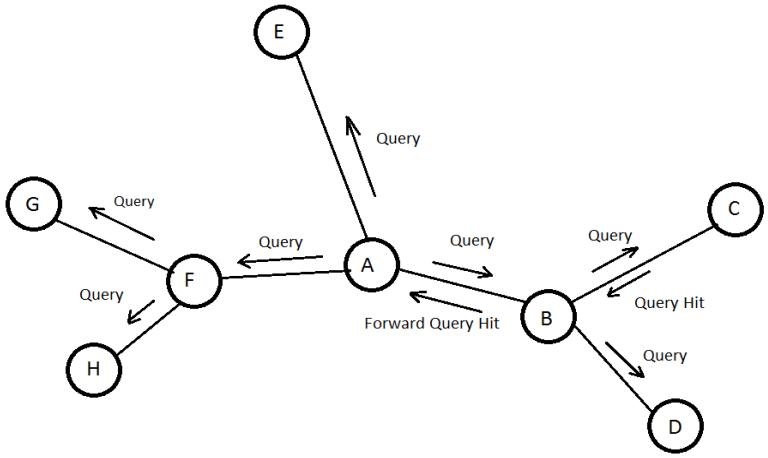

Figure 5: Gnutella overview. Node $\boldsymbol{A}$ sends a query to its neighbors ($\boldsymbol{B}, \boldsymbol{E}, \boldsymbol{F}$), who forwards the query to their neighbors. Node $\boldsymbol{C}$ has a matching object for node $\boldsymbol{A's}$ query, and so returns a query hit message to node $\boldsymbol{B}$, who then forwards the results back to node $\boldsymbol{A}$.

4.4 Structured Networks

Structured P2P concepts, like DHT are fast (in terms of searching) by using a well-ordered application-layer overlay topology, which defines the forwarding of the search requests (cf. concern (b) *"Application-layer topology and degree of centralization"*, section 3.1.2). The DHT concept reveals an ordered overlay topology which is (relatively) easy to maintain.

In a structured P2P network, the peers joining are arranged in a restrictive structure. Peers joining are being placed in an organized way in the overlay topology [19]. This makes it less dynamic but makes searching more efficient. DHT is a good and promising solution that utilizes this technique. The DHT has many different implementations, such as: *Pastry* [7], *Tapestry* [6], *Chord* [5] and *CAN* (Content Addressable Network) [8]. They all share the same fundamental functionality but differ in other solutions such as how to locate other peers and how they are managed. We will in this thesis focus more on Chord due to its flexibility and simplicity, but also because Chord is one of the most researched DHT protocol.

4.4.1 Chord (DHT)

The core feature in most P2P systems is to search and find resources and other entities efficiently. Chord is a solution that strives to make the system scalable with its efficient lookup algorithm, but also robust due to its self-organizing capabilities [5].

You can think of a Chord network topology as a ring of node slots (cf. Figure 6). Every node slot is equivalent to a number between 0 and $2^m - 1$ (e.g. if $m = 4$, we have a ring of slots from 0 to 15). Every node slot that is not already occupied by a peer is a free slot for a new joining peer. Which slot the peer should occupy, is determined by using the *consistent hashing* function [24] on its IP-address. When the position to occupy is known, it will use the same function on its unique identifier (e.g. email address or a unique nickname) to get another number between 0 and $2^m - 1$. In this thesis we will use the term "key" for both the consistent hashed unique identifier and its original value. This key represents another node slot and contains information about the key's owner. The node who occupies this slot is responsible for storing this key, and is called the successor node of the key k. If the responsible node slot is not occupied by another node, the next occupied slot that follows will maintain the key instead.

When node r wants to find node n (aka. Lookup), node r will use its routing table (aka. *finger table*) to localize the successor of the target node's key k. The routing table contains x maintained nodes, which are used as routing paths to communicate with other nodes in the network. The table is frequently updated and maintained. When the successor of key k has been found, the node r can extract information about n's IP-address or its position on the ring.

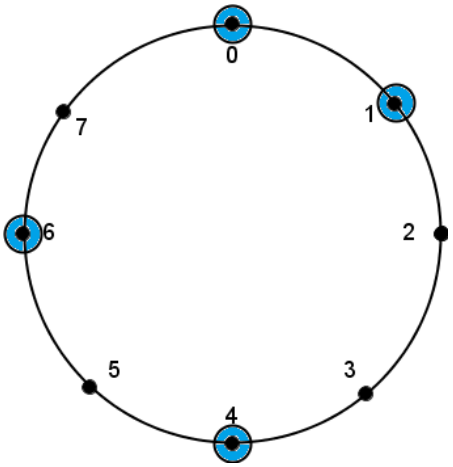

Figure 6: An example of a Chord ring with $m = 3$. Of the 8 possible node slots ($2^3 - 1$), four is occupied by a peer node (shown in blue).

4.4.2 Consistent Hashing

The consistent hashing function gives each peer and its key an m-bit identifier location on the Chord ring, by using a basic hash algorithm, such as *sha1*. Input data for a sha1 hash function will always give the same 160-bit output. The output is typically a 40-digits long hexadecimal number. The outputted hexadecimal number will then be calculated with modulo 2^m to get a number between 0 and $2^m - 1$ (cf. Figure 7). The amount of m-bits to use depends on how many peers we allow into the network. However, the m-bit identifier has to be large enough to make the probability of the nodes or keys to receive the same identifier negligible.

IP: 193.11.185.1 → sha1_hash(IP) → 63aeea5c6d6f86ee497556865802e26157024774 → consistent_hash(sha1_ip) = 3

Key: @eclipse → sha1_hash(Key) → 0ecb9702b7fe231cde95575d1f7a66efa15dbb5e → consistent_hash(sha1_key) = 6

Figure 7: This example shows us how the sequence of consistent hashing works. A node's IP and key are hashed by sha1, later it is calculated by modulo $\mathbf{2^3}$. The result is an identifier number 3 and 6, which represent a node spot on the ring (between 0 and 7).

When the successor node of key k leaves the network, the responsibility of key k is moved to the next successor clockwise in the circular Chord ring. In Figure 8, we can see a chord ring with 8 node spots ($m = 3$). The circle has 3 nodes (1, 2 and 5) and their respective keys (1, 2 and 3). The successor of key 1 is located at its real successor node 1. Similarly, key 2 is maintained by node 2. However, the successor of key 3 does not exist in this example, so the next node followed by 3 which is node 5, will maintain key 3 instead. A similar scenario happens when a new node joins the network; if a key matches its identifier on the circle, the key will get transferred to the new node.

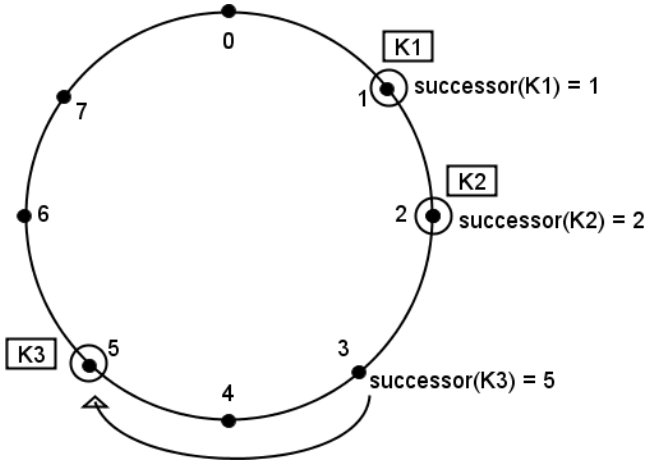

Figure 8: An example of a 3-bit chord ring with 3 nodes: 1, 2 and 5. The keys $\boldsymbol{K1}$ and $\boldsymbol{K2}$ in this example are maintained by their immediate successor, node 1 and 2. The immediate successor of $\boldsymbol{K3}$ (node 3) does not exist, thus the node 5 will maintain $\boldsymbol{K3}$.

4.4.3 The Finger Table

To be able to communicate, each node has to be aware of its successor node on the circle, which is the closest available node in a clockwise order. For example, if node n with the position 15 joins a 4-bit Chord network, the immediate successor of node n would be the node on position 0. However, if the node on position 0 does not exist, the next available node in a clockwise order will be its successor instead. Queries with a given identifier can be passed around the circle using the successor nodes. The query will stop when it first encounters a node that succeeds the given identifier. It is a simple solution but not very efficient. This linear solution is not scalable because in worst case scenarios, the query may have to traverse all nodes in the network. Imagine a network with a million nodes. This is not how Chord actually works, but it utilizes the successor method by maintaining additional routing information.

Each node in an m-bit Chord circle maintains a routing table called *finger table* with (at most) m entries. Every table entry i has a predefined *start*, which is defined after the node has retrieved its position on the identifier circle. The start variable for every entry i in the node $n's$ finger table, describes the identity of the first node k that succeeds n by at least 2^{i-1} on the identifier circle. Thus the i^{th} entry in the finger table of n, maintains the successor $k = \left(n + 2^{i-1}\right)$, where $1 \leq i \leq m$ (cf. Figure 9). However, the start describes the i^{th} finger's true successor, but it does not mean there is an existing node on that position in the identifier circle. If that is the case, the next available node followed by start will be maintained instead. Note that the successor maintained by the first finger in the finger table, should always be the immediate successor of node n. The start position could be denoted as $n.finger[i].start$ and the actual node it maintains is denoted as $n.finger[i].node$.

The Figure 10 is a good illustration on how the finger table may look like. In this example, we have a 3-bit ($m = 3$) identifier circle with the nodes: 0, 1 and 4. The finger table of node 0 has the start pointers: $(0 + 2^0)\ mod\ 2^3 = 1, (0 + 2^1)\ mod\ 2^3 = 2\ and\ (0 + 2^2)\ mod\ 2^3 = 4$. As we can see, the node 2 does not exist in the identifier circle. Instead the next following node 4 has been chosen.

The finger table contains a small number of nodes, yet it is very efficient. What the table does is cut the ring in half. Not many queries have to be sent in order to find another node. For example, if we look again at Figure 10, we can add an additional node on position 6 on the identifier circle. The node 0 wants to find node 6 and the finger closest to node 6 is the last finger that maintains node 4. Because we added a new node, node 4 has now the successors: 6, 6 and 0. As we can see, only 2 queries are required in order to find node 6; node 0 sends a query to node 4 and node 4 forwards it to node 6.

To make it easier to handle arrival and departure of nodes, the finger table also maintains a *predecessor* pointer that points to the closest preceding node. More about this will be discussed below. The definition of the finger table variables can be seen in Table 1.

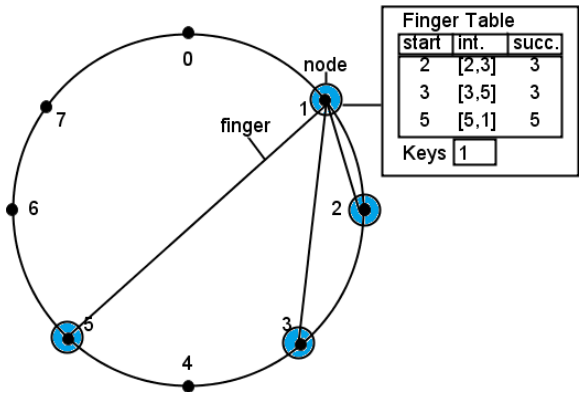

Figure 9: This figure illustrates a 3-bit Chord ring with 4 nodes showing a complete finger table of node 1. The node 1 has the successors 3, 3 and 5 shown in the table. As we can see, the successors are 1, 2, 4 positions away from node 1.

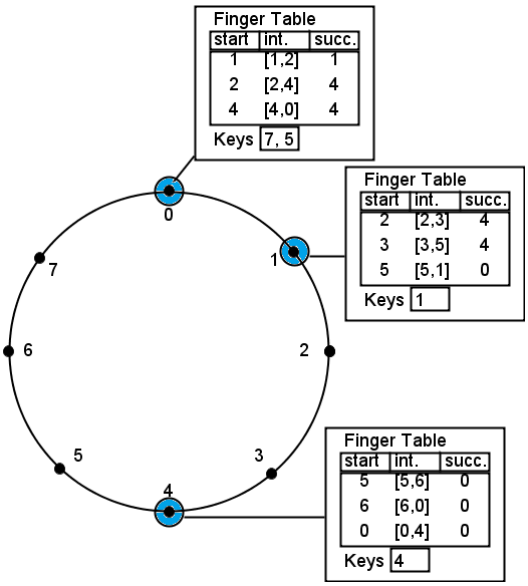

Figure 10: Compared to Figure 9, this figure is displaying all finger tables. The start pointer in this example is not always equivalent to its successor due to missing nodes. Hence the next following node in a clockwise order is a successor.

Notation	Definition
$finger[i].start$	$(n + 2^{i-1}) \ mod \ 2^m, 1 \leq i \leq m$
$.interval$	$(finger[i].start, finger[i+1].start)$
$.node$	$maintained \ node \geq finger[i].start$
$successor$	$the \ immediate \ node \ from \ n;$ $n.finger[1].node$
$predecessor$	$previous \ immediate \ node \ from \ n$

Table 1: Finger table variables definitions of node n are shown here. The table can be seen in **[5, p. 4]**.

4.4.4 Node Arrivals

When a peer joins the network, the first thing to do is to determine its location and its key successor on the identifier circle, by using the consistent hashing function we explained before. Next, the peer has to contact an arbitrary node in the network to establish a junction with the network itself. The arbitrary node will help the peer to find its immediate successor. Let's say node k is an arbitrary node in the network. The joining node n asks k to find the next available successor from its location on the identifier circle. The node k will then use the function $find\_successor(id)$, where id is the identification location of n on the identifier circle, to find the immediate successor of n. When found, the successor will be returned to n. In the current state, the node n knows its successor node r, but n is not part of the network yet.

Figure 11 illustrates the current joining state of node n. As we can see, the other nodes are not aware of n yet. The predecessor of node r (node a) should be $n's$ predecessor and n should be the new immediate successor of node a. Thus the next step is to notify r that it got a new potential predecessor n. The notify function being called will check if the node n is a closer, immediate predecessor of r and if that is the case, r will change from its current predecessor to node n.

Now the new node n is part of the network, but the system's current state is not at its best, because n only have one successor in its finger table, and the immediate successor of node a is not pointing to the closest successor. In addition, n is not visible to other nodes except r, because their finger table's has not yet been updated.

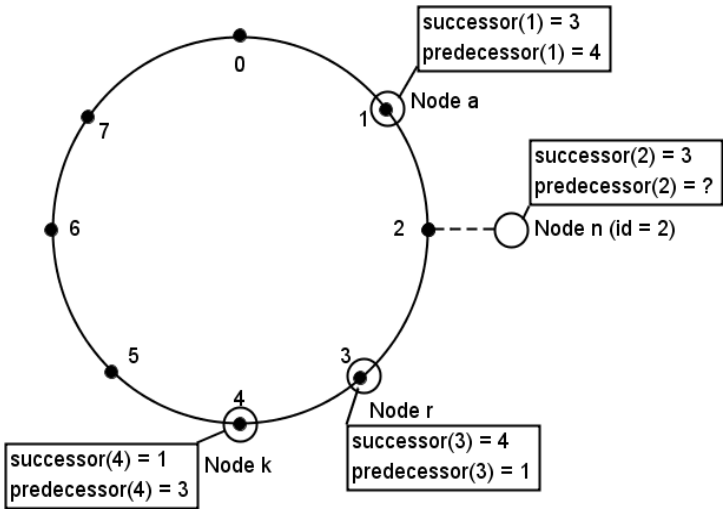

Figure 11: An illustration of an early stage of the joining sequence of node n. Node n is aware of its successor r. However, node r and node a are not aware of n yet at this point.

To cope with these problems, Chord will stabilize itself and solve these problems overtime. It is a more efficient way than trying to solve all problems at the same time to stabilize the system. For example, take into account that there could be more than just one peer trying to join at the same time. Fixing all the problems that node arrivals generate at the same time is costly. Therefore, every node in Chord has two functions that are being called regularly every time t (in milliseconds), $stabilize()$ and $fix\_fingers()$. The $stabilize()$ function verifies every time t if a node $n's$ current immediate successor is the closest successor of n, and updates its successor if a closer node has been found and notifies it of its presence as a potential predecessor. The $fix\_fingers()$ function updates its fingers by finding new closer successors to the start variable. This function only updates one finger per call in an incremented order. Because outdated fingers have a low impact on performance, Chord can update the fingers lazily. So, if we go back to the example from Figure 11, the node a will eventually change its successor to node n and notify it of its presence and the finger tables will be updated (cf. Figure 12). More about these important functions will be explained later in this report.

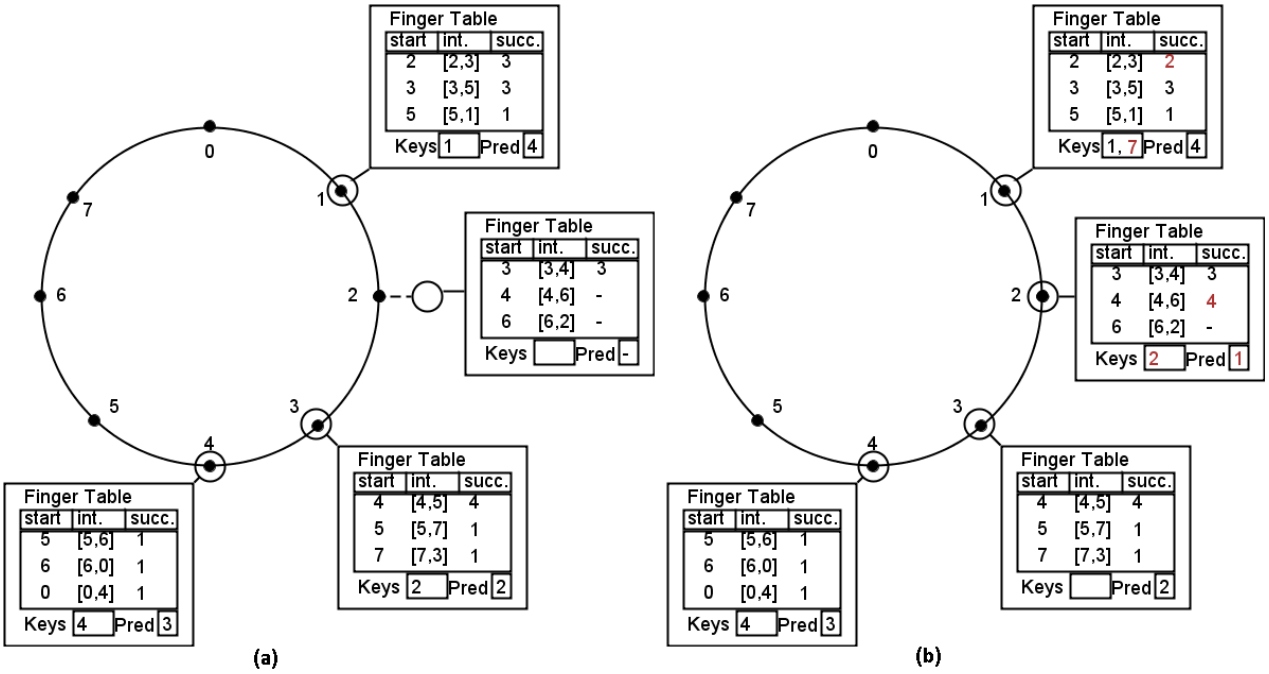

Figure 12: (a) Illustrates the finger tables when node 2 is in an early stage of joining. The node 2 has notified node 3 to be its potential predecessor. Note that the node 1 has not yet called the ***stabilize***() function as well as node 2 has not called ***fix_fingers***(). (b) In this example, both of these functions have been called after time t. Note that node 2 still has one missing finger yet to be updated. The changes from example (a) are shown in red.

4.4.5 Node Departures

When a node voluntarily leaves the network, the most important thing is to transfer its maintained keys to its successor. However, a node could depart from the network without notifying due to loss of connection or by simply closing the application in an incorrect way. If this happens, the node is incapable of communicating with other nodes and is therefore unable to transfer any keys. A solution to this problem is to replicate the keys to successor nodes. This will be discussed more in the Node Failure section 4.4.7.

Node departures have a similar sequence to node arrivals. If the departure is voluntary, the keys will first be transferred from node n to its successor k to secure the maintained keys stays in the system. Second stage, if it is voluntary, node n notifies its predecessor p of the departure and gives p a pointer to its successor k. The predecessor p can use the successor k and notify it as its potential predecessor. However, the system should focus more on involuntary departures as it is probably going to happen more often and is more critical to the system. If node n involuntarily depart due to connection problems, or other various reasons, its successor k, its predecessor p and other nodes with n in their finger table, will eventually notice the departure of node n and thus must find a new node to replace it. Any node can leave at any given time and it is one of the most complicated problems in Chord. When a node fails to respond to a query, the node is considered to be dead and a *node failure* occurs. The node with a failed node has to find a replacement node quickly to maintain a stable system. Chord has a solution to this problem and will be explained more in depth in section 4.4.7.

4.4.6 Stabilize Protocol and Self-Organizing Operations

The stabilize protocol makes Chord a self-organizing system. The protocol's objective is to validate and update immediate successor pointers as new nodes joins and leaves the network. The protocol is executed every time t (in milliseconds) to regularly check if there is a new closer node than the current successor, or if the current successor is a failed node. The fact is the nodes are responsible for identifying their own immediate successors and to notify them to change their predecessor pointers to them if the successor agrees to this change, which in most cases they will. The $stabilize()$ function will do the following when called in node n:

1. First, the node n will query its immediate successor node s, to fetch the predecessor p of node s.
2. If s fails to reply the query, replace s with the next available finger in the finger table and try again.
3. Node n checks if the predecessor p is in between n and s and if the predecessor p is not empty.
4. If true, n replaces the current successor with p and notifies it. If false (or empty), n notifies his current successor as its potential predecessor.
5. Stabilize has finished and will be called again after time t.

We will go through a scenario to explain how the stabilize function works in Chord if a node involuntarily departs from the network. Think of a 3-bit Chord network with 4 nodes: 0, 2, 4 and 6. Every time t, the nodes are calling the stabilize function to see if a closer successor exists, but because no changes have been made in the system, the current successor remains. After a while, node 6 involuntarily leaves the system and node 4 notices node 6 does not reply its queries when it runs the stabilize function again. Node 4 has to replace its failed successor by finding a new one. It will use the next available finger node 0 in its finger table to query it instead. Node 4 asks node 0 to get its current predecessor and because the node 0's predecessor is the failed node 6; the node 0 will become node 4's new successor. Node 4 then notifies node 0 as its potential predecessor, and node 0 changes the predecessor pointer to node 4.

As we can see, node 4 organized itself to find a new successor in a simple and efficient way. However, there are still a few small problems lingering after node 6 left the network and the stabilization of node 4. The nodes 4 and 2 had node 6 in their finger tables; hence they have outdated finger tables that need to be updated. The function $fix\_fingers()$ will solve this problem by regularly update only one finger at a time, every time t. This lazy update is possible because having outdated fingers is a small performance issue and therefore a cheap update is more efficient than updating all the fingers at once. The function will do the following when called by node n:

1. Increment an index by one to determine the next finger entry, denoted as $next = next + 1$.
2. Check if $next$ is greater than the maximum number of fingers. If true, reset $next$ to its initial value 1, denoted as $next = 1$.
3. Find the successor of the $next^{th}$ finger using its start value, denoted as $s = find\_successor(finger[next].start)$.
4. Replace the current $next^{th}$ finger with the new node s, denoted as $finger[next].node = s$.
5. Fix fingers has finished and will be called again after time t.

Each node also periodically runs another function denoted as $check\_predecessor()$. It is used to determine if the current predecessor has involuntarily left the system. The function will clear the inactive predecessor in order to accept new predecessor nodes in the $notify()$ function. This check is essential because nodes rarely communicate with their own predecessors.

4.4.7 Node Failures and Replication

A node failure occurs when a node is not responding after a certain time. The user might leave at any given time without notifying due to connection problems or other various reasons, which cannot be controlled by the P2P system. Every peer in a P2P network relies on each other to be available to forwarding packets or answering queries, therefore a node failure is a more critical problem than in a Client-Server network. There is no good solution to check if a node is alive or dead, the only way to know is if the node fails to reply in a certain amount of time (a.k.a. *time out)*.

When a node is considered to be dead, its keys have to be moved to the successor node and the dead node has to be quickly replaced by an active node to ensure efficiency. However, the node is incapable of communicating and therefore cannot transfer its unknown keys to the successor node. Node failure is a big challenge for P2P systems. Every node has to accept the fact that a connection to a node could fail in any given time. If a node failure occurs in one of the fingers, why not use the next proceeding finger? It is possible to utilize another finger if one fails, but it is important to maintain the accuracy of the successor fingers as the precision of lookups depends on it. Imagine this scenario where the first 3 fingers out of 4 fails and the node n's only option is to send the lookup query to the 4th finger. By sending it a lookup query of a certain key k that should be located somewhere behind the 4th finger's query range, but the node n assumes the key k is located at the 4th finger. This will result in n sending incorrect query replies for the key k.

Chord copes with the node failure problem by introducing a list of maintained successor nodes. The successor-list contains a node's r nearest successors on the Chord-ring, which are used to temporarily replace dead nodes, until new nodes have been found. A modified version of the stabilize function explained earlier maintains the successor-list by refreshing it. If a node detects a node failure on one of its fingers, the node will utilize the successor-list and temporarily replace the dead finger node with the first live entry from the list and then re-run the operation with the new node. The result is that, even if a node fails, lookup queries are able to proceed by using alternative routes by using the dead node's successor. The Chord system is only affected if all the successor nodes fail simultaneously.

The successor-list deals with the node failure problem but it does not solve the key problem. The dead node is unable to transfer its maintained key's to the new successor. This problem is solved in Chord by replicating the keys to other nodes. There are different ways to implement the replication algorithm. A good idea is to take advantage of the successor-list, because the list contains a node's closest successors. The replication of a node's keys is done every time a node receives a new key or transfers a key to a successor.

4.4.8 Chord Functions and Definitions

In this section we will go through the important main functions in Chord to get a better understanding how the system works and how they are written in pseudo code examples. Note that a function can be called locally or remotely. Locally means the function is being called by its owner node n (this) while remote calls means the function is called in another node n'.

4.4.9 *Find Successor Function*

This function asks node n to find the successor of the id provided in the function parameter. If the successor of node n is not the successor of the provided id, node n will recursively forward the query to the closest preceding node n' in its finger table.

The following pseudo code describes this function:

```
// ask node n to find the successor of id
n.find_succesor(id)
    if(id ∈ (n, successor))
        return successor;
    else
        n' = closest_preceding_node(id);
        return n'.find_succesor(id):
```

4.4.10 Closest Preceding Node

This function asks a node n to find the node that is closest to the id provided in the function parameter. This function is used to search efficiently by always fetching the best node for the query, which exists in between n and id. If no node exist in the interval between n and id, the function will return its local node.

It is possible to modify this function so it also searches for nodes in other lists, such as the successor-list to increase accuracy. The following pseudo code describes this function:

```
//ask node n to find the closest preceding node of id
n.closest_preceding_node(id)
    for i = m downto 1
        if (finger[i].node ∈ (n,id))
            return finger[i].node;
   return n;
```

4.4.11 Join Function

The join function starts the join sequence that allows a node to join the Chord ring. The function basically asks an arbitrary node from the network, provided in the function parameter, to find its successor. When the joining node has received its immediate successor, the periodically called functions stabilize and fix-fingers will take care of the rest. Note that this function can only be called locally; it cannot be called remotely by another node.

The following pseudo code describes this function:

```
//join a Chord ring with the help from an already joined node n'
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);
```

4.4.12 Stabilize Function

This function periodically validates and updates the current immediate successor of n every time t. If a closer successor is found, the function will notify the new successor of its presence. If the successor accepts the new node, it will set it as its predecessor. This function may also maintain the successor-list by refreshing it. Note that this function can only be called locally; it cannot be called remotely by another node.

The following pseudo code describes this function:

```
//periodically validates the successor of n and notifies it of its presence
n.stabilize()
    x = successor.predecessor;
    if(x is not nil and x ∈ (n,successor))
        successor = x;
    successor.notify(n);
    list = successor.successor_list;
    // an add function that adds the new successors to the list
    //sorted list, if full: replace old with new, else add to list
    successor_list.add_new_successor(list);
```

4.4.13 Notify Function

This function is remotely called in node n', when node n thinks it is the immediate predecessor of node n'. The node n calls this function in the stabilize function. The function checks if the node n really is the immediate predecessor of n'. If true, node n' sets node n as its new predecessor, or else it will do nothing. The reason why it checks is due to security reasons. There could be a rogue node trying to harm the system by pretending to be the predecessor.

The following pseudo code describes this function:

```
//node n thinks it is the predecessor of node n'
n.notify(n')
    if(predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';
```

4.4.14 Fix Fingers Function

This function is periodically updating a finger table entry, every time t. The function will increment an index variable by one every time it is called and therefore update each entry linearly. The function will perform a successor lookup on the chosen finger's start value to see if it can find a more accurate successor node than the current one. Note that this function is only called locally; it cannot be called remotely from another node.

The following pseudo code describes this function:

```
//Periodically update finger table entries
//the variable m is the number of fingers and
//the next variable stores the next index to fix.
n.fix_fingers()
    next = next + 1
    if(next > m)
        next = 1;
    finger[next].node = find_successor(finger[next].start);
```

4.4.15 Check Predecessor Function

This function simply checks periodically if the node n's current predeccessor has failed. If it has failed, the current predecessor is set to nil. This allows the node to accept new predecessors in $notify()$. The reason why it requires a periodically called function is because the nodes rarely communicate with its predecessor and thus has to determine if it is still active or a dead. Note that this function is only called locally; it cannot be called remotely from another node.

The following pseudo code describes this function:

```
// periodically checks whether the predecessor has failed
n.check_predecessor()
    if(predecessor has failed)
        predecessor = nil;
```

4.5 Hybrid

A hybrid is a mix of a decentralized and centralized system. For example, a system could use the Client-Server architecture and utilize some parts from the P2P architecture. When a system is a hybrid, it is not defined as a pure P2P system because it is not fully decentralized.

4.5.1 Napster

Napster uses a cluster of centralized servers to maintain index of files that are shared in a network (cf. Figure 13). A peer in Napster obtains list of files from a central server and initiate a file exchange directly to that peer who is currently sharing the file. Napster [21], compared to Gnutella [20], is a hybrid P2P system. This is because it uses central servers to store information about files in a network. If a server goes down, it will affect the whole system.

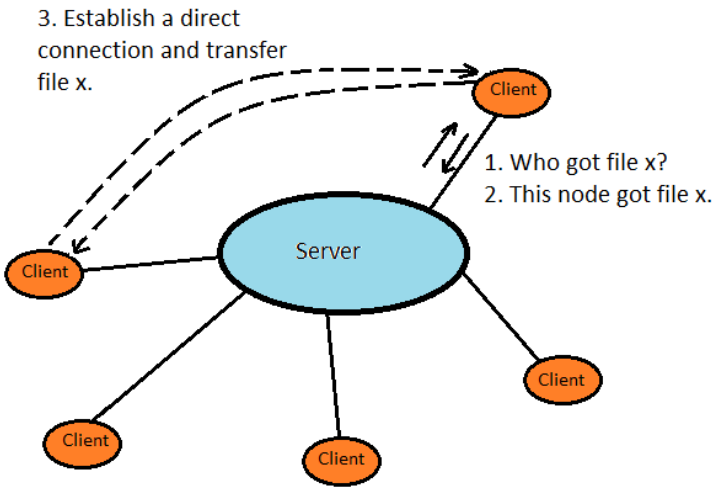

Figure 13: Napster overview. A client asks the server about a certain file. The server answers the client with the IP-address to the client who shares the file. The client then establishes a direct connection to the file owner.

5 Online Chat Application

Now let's turn to an application that may take advantage of the P2P technology. Online chat applications have been around for some time and are used by millions of people every day to communicate quickly and easily with each other around the globe. A chat application is a network communication between computers where text messages are transmitted in real-time. The messages are generally short in order for people to respond quickly, making them more similar to oral conversations, as opposed to email or Internet forums, for example.

5.1 Requirements

There are many different kinds of requirements for an application. We will focus on requirements that are essential for this thesis. For example, a typical requirement could be the user interface design. This is obviously not essential for this thesis as it describes the usability and graphical elements which do not contribute to the achievement for our goals. We will focus more on the performance requirements and other aspects which are essential for P2P protocols.

5.1.1 Scalability

To be able to handle thousands, or perhaps even millions of users connecting and communicating with each other, the system has to be able to scale without degradation in performance. This is an important requirement for all big network systems. If the network is only aiming to serve a few users, scalability is not that important but it should always be kept in mind so the same problem/issues do not occur as they did with the Gnutella protocol (section 4.3.1).

5.1.2 Robust and Fault-Tolerant

The network has to be able to withstand errors and should not go down easily. Users rely on the system to be online at all times and if the system cannot fulfill this requirement, users may experience difficulties with joining the network and communicating with other users.

5.1.3 Guaranteed Message Delivery

Users rely on the system to deliver their message to the recipient. Even if the recipient is offline, the message has to be delivered when the recipient comes online. In addition, a message has to reach the recipient as fast as possible. The messaging is happening in real time, but a small delay is not a big issue. A few seconds delay is acceptable but more than that, minutes and hours, are unacceptable for a real time chat service.

5.1.4 Fast and Efficient "lookups"

The true power lies on how fast you can obtain your friends new IP address or find new friends in the network or other resources. Every time a user comes online, the chat application has to somehow identify which of its friends are online. A fast and efficient lookup algorithm will solve this problem, even if the user has many friends. The definition of a fast lookup is the time it takes to find a target. An efficient lookup is defined by the amount of packets which are required to be sent in order to identify a target. A high amount of packets being sent may stress the system and cripple the performance.

5.2 Existing P2P-Applications

There are not many Chat applications in existence that use the P2P technology. Most applications that utilize P2P are file sharing programs such as µTorrent [25]. One famous chat application that used the P2P technology was Skype [26]. Unfortunately, Skype recently changed its protocol to a more Client-Server based architecture. Below we will discuss a few P2P chat applications and also the problems and difficulties with such systems.

5.2.1 Skype

Skype is a famous online text-messaging and voice chat application that is used by millions of people around the globe. Today the Skype application is no longer using the P2P technology. In 2011, Microsoft acquired the Skype Company and recently Microsoft decided to reconstruct the protocol, from its famous P2P architecture to Microsoft Cloud [2], which is a more Client-Server based architecture. The reason for the reconstruction is believed to be because of the increased use of mobile devices. Nevertheless, the elder P2P Skype protocol is worth mentioning here because the reason Microsoft decided to change the protocol is an important detail (section 5.2.3). The original Skype protocol is more of a hybrid rather than a pure P2P system and little is known about the protocol because it is not open to the public. However, based on a reverse engineering of the Skype application in 2006 [27], Skype uses special nodes called *super nodes*. Peers with the best bandwidth, CPU and memory are chosen to act as super nodes. An ordinary node must connect to a super node in order to be a part of the Skype network. The node must also register itself with the Skype's login server to identify the user.

5.2.2 µChat

µChat [28] is a relatively new chat application created to be used in µTorrent. The main idea behind µChat is to make it a more pure P2P application without the need of a central server. Although the application is not famous, the idea of creating a chat application without any need of servers is exemplary.

5.2.3 Problems and Difficulties

We mentioned above that Microsoft changed the famous P2P protocol of Skype to the Cloud, which is a more Client-Server based protocol. The reason is believed to be because of the increased uses of mobile devices [2]. But why is that a problem? As we already know, Skype selects users to serve as super nodes. The responsibility of a super node is higher than an ordinary user and therefore requires more resources and power. This is why the users with best CPU and memory are chosen to act as super nodes. The problem is the mobile devices act as bad and slow super nodes [1], due to battery depletion and power consumption. Another problem with mobile devices is that the IP address is likely to change at any given time due to relocation of the device to another network when moving around (also known as "roaming"). Because it is mobile, the device can freely move anywhere, including areas where the connection may be lost (e.g. a tunnel when riding the bus). Losing the connection, even for a short period of time, could hurt the performance, especially if it concerns a node with more responsibility where other nodes rely on it to be alive. This could be a problem for other P2P protocols as well, because peers are working as paths to communicate with other peers in the network. If peers' paths are slow or are simultaneously losing connection, it could cause problems with lookups.

6 What is P2P Performance?

Describing what performance is for P2P systems is not an easy task. The P2P area has a huge variety of different architectures. Using only one metric for all of them is close to impossible. P2P architectures and algorithms have to be measured corresponding to what they are meant for. But some functionality in P2P systems shares the same fundamental principle. The concept of P2P performance can be found in [18, pp. 383-396].

6.1 Performance of Information Lookup

In a distributed P2P system, peers use other peers in the system to receive information about other entities in a network. To do this, a peer n sends a lookup query to a known peer r in the network. If the peer r knows the requested entity, then r will send a reply to peer n, containing the information about the entity. If the entity is not known, peer r will then forward the query to another known peer. During this lookup process the peer n is waiting for a response.

In a central server system, the information is managed by the server. In theory, a lookup query in such a system gives us the complexity of $O(1)$. The requester only needs to know about the central server. Thus, it makes the performance faster compared to a P2P system, where the data is distributed amongst other peers. However, managing all the resources has some drawbacks. The server requires more potent hardware such as CPU and memory but also network availability. In terms of memory consumption, it gives us the complexity of $O(n)$.

In distributed P2P systems, the search performance is essential. Finding resources or other peers in a network efficiently is fundamental. Other essential aspects are arrivals and departures of peers. Peers should be able to join and leave the network dynamically and efficiently. Few nodes should be affected by it, no matter how big the system is (scalability). *Churn rate,* is a common measuring concept that is used to describe the frequency of a node that is joining and leaving a P2P network. For example, a high churn rate is when a large amount of peers joining and leaving a network in a specific period of time. We will use this concept when we refer to this case.

6.2 Network vs. Overlay Performance

Because P2P is a networked application architecture, the measuring has to be done in the network layer. Measuring performance of the lookup functionality could be done by counting the number of hops between peers that the query encounters before it reaches the target. Every time the query reaches a new peer, the hop counter is incremented by one. The result is later returned to the peer who initiated this query. Many hops mean greater delay. This can be used to test how efficient a system is when searching for a resource within a network. The test could also be combined with latency metric to calculate how long time it takes in milliseconds for a peer to find a resource.

Unfortunately, it is more complicated in the real world. When a peer joins the network, it will maintain a connection with other peers. It will use these peers as paths to be able to communicate with others and access resources from within the network. But It does not necessarily mean the maintained peers are close in a geographically point of view. The peer n has a direct connection with peer r. The peer r is located in China and n in England. The peer n sends a lookup query to find k which is located in Denmark. If r does not know k, then it will continue routing the query to another peer. This will affect performance and latency measurements. This is a good illustration of why the number of hops in a search query is essential.

Performance Evaluation: *This section, hops has been chosen as the performance metric (3).*

6.3 P2P Performance Facts

What kinds of mechanisms are influencing the performance in a P2P system? The main reasons can be divided into four categories: network performance feature, overlay structure, hardware and P2P-behavior.

Performance Evaluation: *This section, the four categories that affects performance is mentioned. They are described below. (4).*

6.4 Network Performance Feature

There are many factors in a network that could impact the performance. Many of them will be ignored during measuring, simply because it is difficult to take them all into account. One packet that is being sent over the network can encounter many fractious problems on the way to its destination, or sometimes not arrive at all. The following are few factors that are often considered important: throughput (bandwidth), latency and packet loss.

6.4.1 Throughput

Throughput (bandwidth) is defined by how many bits per time unit that can be pushed through the network. For example, you can think of it like the number of lanes on a freeway. The more lanes, the more traffic the freeway can provide. The bandwidth has a great impact on the performance, especially for P2P systems, where the peer contributes to the system and works like a router. If a peer has a small throughput, it could affect not only itself, but also others.

6.4.2 Latency

Latency is a metric that describes the delay between the source and destination. When a packet is being sent, it has to go through various amounts of physical routers that will guide the packet to its destination. The time it takes depends on many different variables: how far away the destination is, how many routers it encounters and how fast it is being transported (throughput), for example. The latency can either be measured *one-way* (the time from source to destination) or *round-trip (*RTT - *Round Trip Time)* which is the time from source to destination and back.

6.4.3 Packet loss

A packet may be lost in transmission due to bit errors (packet being discarded) or congestion (buffer overflow). Bit errors occur when received bits of data have been altered due to interference, noise, distortion or synchronization errors. The congestion on the other hand is entirely different. Sometimes routers or switches might have to dispose packets intentionally in order to reduce the traffic and maintain a certain service level. Hence, a packet loss could indicate a network overload and cause TCP to reduce the throughput for the TCP connection. The transport protocol TCP (*Transmission Control Protocol*) guarantees reliable deliveries of packets due to its error checking capabilities, but the TCP causes performance issues due to packet retransmission and byte order correction. When a packet loss occurs, TCP will retransmit the package, therefore causing a delay.

6.5 Overlay Structure

Choosing one overlay topology has a great impact on the performance. The topology describes how the peers are managed in the system and how the communication works between them. In a virtual point of view, the peers are commonly called *nodes*, which are connected to other nodes with virtual or logical links which corresponds to a communication path.

Which structure to choose from depends on what the program is trying to achieve. Big systems with many peers require scalability and fast search. There are two kinds of decentralized overlay topologies: *unstructured* and *structured*.

6.5.1 Unstructured Overlay Topology

In an unstructured overlay topology (section 4.3), there are no particular rules how nodes are managed. Their connection links to other nodes are somewhat randomly chosen. This makes the system more robust when facing high rates of churn. It also unlocks the possibility of localized optimization, putting nodes connection links close to each other in the physical overlay point of view. But in big systems with many nodes, searching has a big drawback in performance. Because there is no structure, a node must flood the network in order to find a resource, which causes high traffic and more CPU and memory usage. In addition, the resource is not guaranteed to be found because of the *TTL* (*Time to Live*). To make sure a query never bounces around the network forever, every query has a limited number of hops. When the limited number is reached, the query is discarded.

6.5.2 Structured Overlay Topology

Structured overlay topologies are the opposite (section 4.4). The nodes are organized, which makes search more efficient. One common and promising implementation of a structured decentralized distributed system is *distributed hash tables* (DHT). With DHT, one node knows where to send the query to find a specific resource. A structured system is scalable but the drawback is departure and arrival of nodes. In DHT, every node must maintain a routing list of nodes and if a node leaves or joins, the list has to be updated. To update the list, the node has to search for replacement nodes in the network. This could generate performance problems if the system experiences high rates of churn.

6.6 Hardware

Because every peer in the network contributes to the system, more stress is put on the user's hardware. Users have more responsibility and, particularly, a slow computer could hurt the performance. Today, as mobile devices are getting more popular, the use of network applications on mobile devices is increasing. This could generate performance issues in P2P network applications. Slow devices could cripple the performance and, being a mobile entity, the IP address would most likely change more frequently. In addition the mobile device could lose the internet access more often. For example, if you are riding a bus and the bus goes into a tunnel, the mobile device could lose its connection for a short period of time. This is one of the main reasons why Microsoft decided to dismantle the P2P functionality in the Skype [26] application and replace it with Cloud [2], which is a Client-Server based system.

6.7 Peer-to-Peer Behavior

As mentioned before, there is a huge amount of various types of P2P architectures. A decentralized architecture can either use a structured or unstructured approach, but their algorithms and behavior may differ. The DHT protocol for example has many different implementations and behaviors, such as: *Pastry* [7], *Chord* [5], *Tapestry* [6] and *CAN* (Content Addressable Network) [8]. These implementations use different algorithms for how to manage the nodes and for how to search for a resource. Some implementations use similar solutions, but all use the same concept.

One of the most challenging things with DHT systems is to make them self-organizing. The protocol strives to become fully functional and usable with thousands, or perhaps even millions, of users. One implementation could have slightly different versions. For example, Chord may use additional or slightly different algorithms for how to stabilize the system if node failures occur. A node failure is when a node fails to reply a certain query in a certain amount of time. If it occurs, nodes being affected by the failed node have to update their routing tables. Simultaneously failing nodes may affect performance with an unsuitable algorithm.

6.8 Comparison

If we want to compare two systems, how can we compare the performance? Simply comparing the theoretical complexity (*big O notation*) is not good enough, because the real world is more complex. P2P is more complicated to measure and compare because the work is distributed with many different workers with different hardware and internet bandwidth around the world; whereas in a Client-Server system, we know exactly what we are working with. However, the big O notation could give us an idea how the system may perform in best case and worst case scenarios. If the complexity is far away from each other e.g. $O(log\ n)$ and $O(n^2)$, we could easily draw a conclusion. A better way to compare the performance is to evaluate them with practical tests. However, practical tests could be a challenging task. To get accurate data, many computers around the world must participate in this test. In most cases, this is not possible, especially for large systems. An easier way would be to run a simulation on a single computer with the use of threads to simulate peers. The data would not be as accurate, but the results can be used for evaluation to see how the system performs [17].

7 Implementation and Performance Evaluation

For this thesis, a Chord simulation application has been implemented in order to perform practical lookup tests. The implementation is based on the paper “Chord: a scalable P2P lookup protocol for internet applications”, by I. Stoica et al. [5]. The pseudo code for the most relevant Chord functions which this implementation has adopted can be found in section 4.4.8.

The Java language [29] was chosen for this implementation due to its simplicity and its ability to run on different operating systems. The Java language has a wide variety of APIs which saves a lot of time and effort for the programmer. However, there is a disadvantage by using Java as the implementation language for this Chord application in terms of performance. Java uses more memory and is generally slower compared to e.g. C++. Therefore the maximum number of nodes that can be created and simulated on a single computer will become more limited.

7.1 Chord Application

The application simulates a Chord protocol system on a single computer, were nodes can be added and removed from the Chord ring. The application also provides a lookup test functionality where the user can perform lookup tests on random selected nodes. The test results are saved in a file with a custom made format which can be used to make statistics.

Because the program is a simulation and is only run on a single machine, there is no need for the nodes to communicate using network protocols. This means the nodes do not send any network packets to each other over the internet, but rather simulate packet exchanges using event messaging within the program. This solution does not affect how Chord works or its general performance. The only difference is the communication latency between the nodes. The time between each message is significantly reduced compared with a real Chord system that sends packets over the internet. For this thesis, the time data is not very relevant but is included in the statistics. The main purpose of this application is to test the lookup functionality to see if it is able to scale and discover problems and difficulties.

Performance Evaluation: *This section explains the workload, e.g. how the lookup tests are being logged and tested. Below is a more detailed overview of the applications features (7).*

7.1.1 Application Features

When the application is started, the user can choose from how many bits the system should utilize and how many maximum fingers each node should have in their finger table (cf. Figure 14). The most optimal setting is when the bit value is equal to the max fingers value. If the max fingers value is less than the bit value, the last finger will not cut the ring in half, which means a lookup operation could result in more hops and a decrease in performance.

When the Chord network is started, the user can go into the console tab (cf. Figure 15). In this tab, the user can create nodes and perform lookup tests. This tab also provides buttons which give the user information about the Chord ring such as successors, finger tables and which keys each node is responsible for and also which keys they have as replicas. There is also a functionality that enables the user to kill a specific node to see if nodes can automatically organize themselves and stabilize the system into a more stable state.

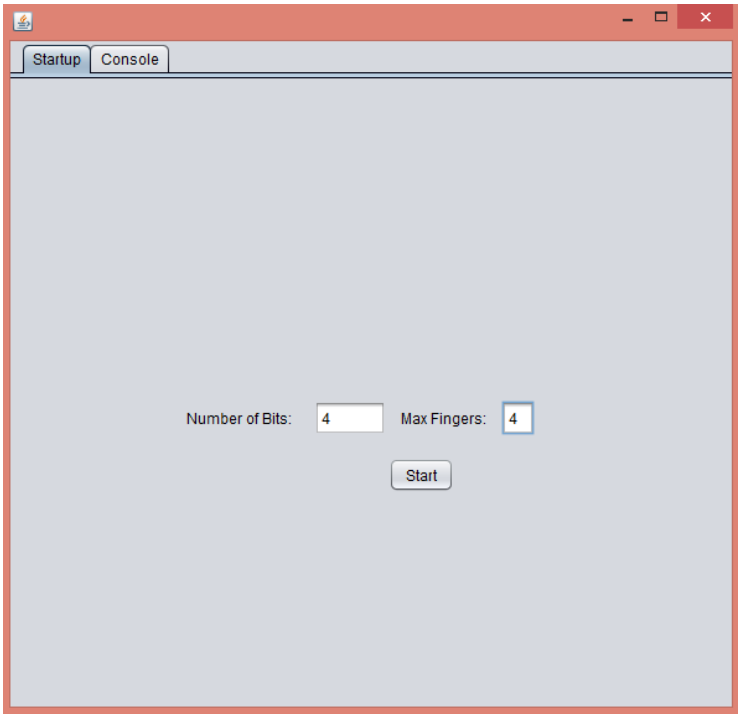

Figure 14: The start window of the Chord application. Here the user can choose from how many nodes the system can create (2^{bits}) and how many fingers each node should have ($1 \leq fingers \leq bits$).

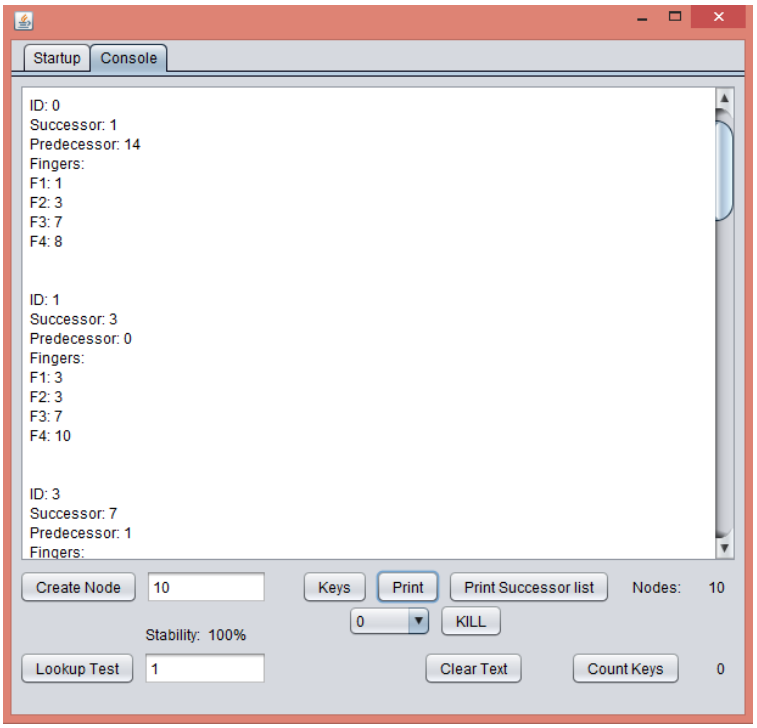

Figure 15: The console window of the Chord application. Here the user can create nodes, perform tests and get information from the Chord ring. In this example, 10 nodes has been created In a 4-bit Chord ring. The print button has been clicked in order to see the current finger table of each node.

7.1.2 Performance Evaluation

We used this application to test the scalability of the Chord protocol by simulating lookup queries between random nodes. The application gives the user the ability to choose how many lookup tests should be performed in a test sequence. When the arbitrary amount of lookup test has been chosen, the user starts the test by clicking the button “Lookup Test”.

The lookup simulation will randomly select one target node and one lookup issuer node. The issuer node will perform a lookup on the target’s key. The lookup function will perform recursive iteration between different nodes by forwarding the issuer’s lookup query to the node closest to the target key. When the query reaches a node that is responsible for, or has the key as a replica, that node will notify itself to the issuer and thereby terminate the lookup. The issuer can then extract key information such as the IP address of the target node. When the node has completed its lookup, the next lookup test will be prepared by randomizing a new issuer and a new target. This scenario will repeat until all tests have been performed. Note that tests are not done concurrently, but only one at a time.

In this thesis we performed 1,000 lookup tests with different quantities of active nodes in a Chord ring. Two demonstrative tests have been chosen to illustrate the scalability of the Chord protocol in form of graphs. The most important element in the graphs is not the numbers, but the shape. The shape demonstrates Chord’s logarithmic behavior when searching for other nodes in the network.

The first test was made in a ring with 1,000 nodes (cf. Figure 16). In this test, 7 hops had the highest probability of 26.5% of the lookup tests. And not far away, 6 hops had its share of 25.7% as well as 8 hops with 19.3%. The highest hop result this test shows us was 11 hops, with the probability of 0.1%, which is a very small chance. The test also shows the time taken per each hop on average, which was proved to be very similar. This means, every peer in the system is treated equally, because it takes almost the same amount of time to perform the same query on each peer.

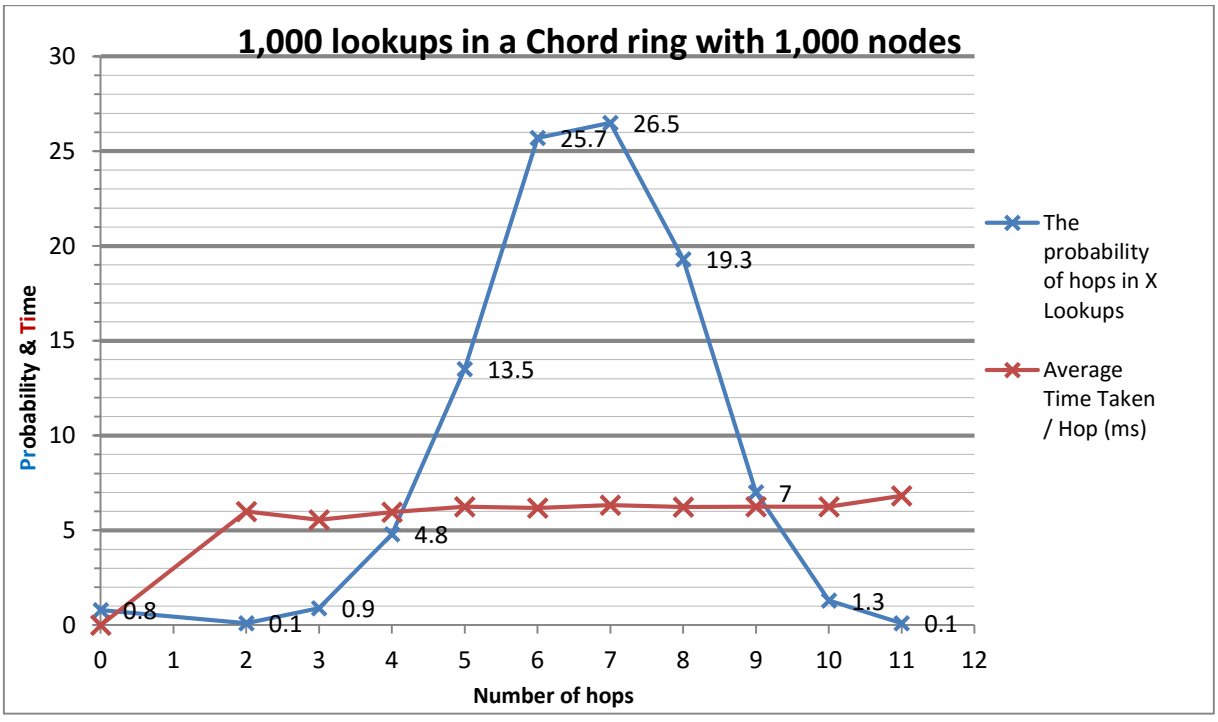

Figure 16: This is a result from 1,000 lookup tests in a ring with 1,000 nodes. As we can see, it does not require many hops to find the target node. The statistics shows us that 7 hops has the highest probability to occur (26.5%), followed by 6 hops (25.7%) and 8 hops (19.3%). The shape of the blue line tells us the search is done logarithmically.

In the second test we increased the number of nodes by 4 times from the previous test (cf. Figure 17). As we can see from this test, 8 hops have the highest probability of 24.4%, followed by 7 and 9 hops (23% and 17.2%). Clearly a small change if we considering the big difference.

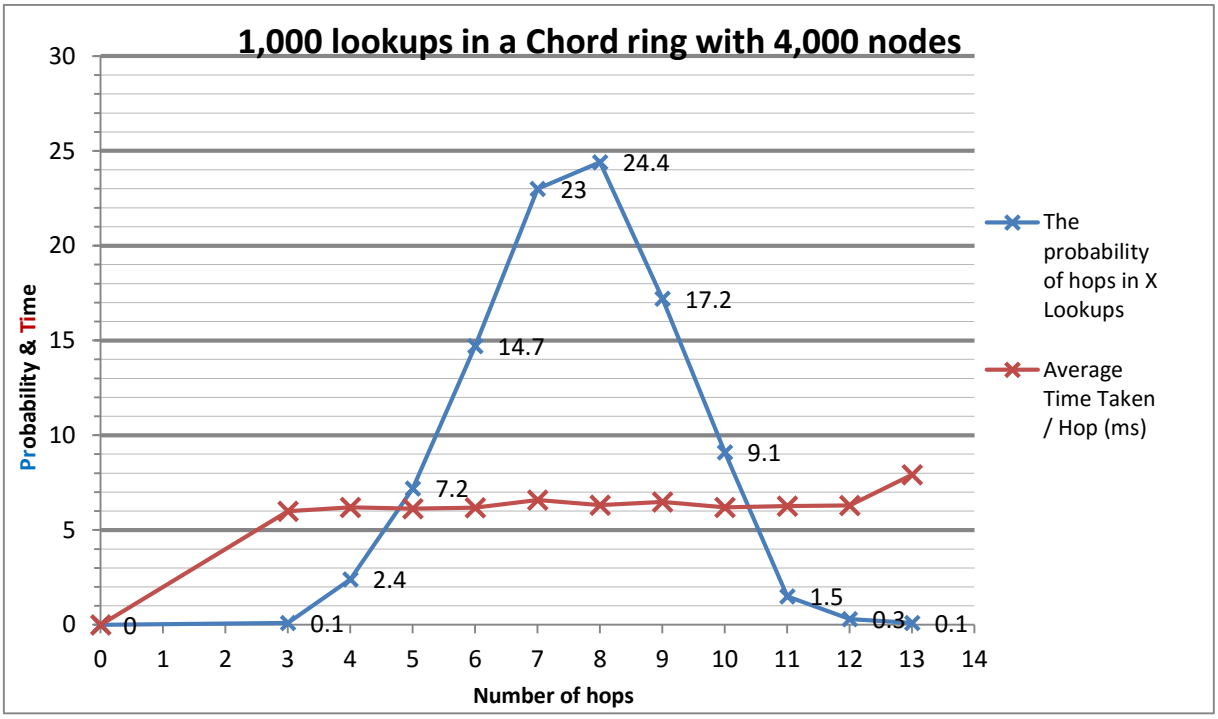

Figure 17: This is a result from a 1,000 lookup tests in a ring with 4,000 nodes. Compared with Figure 16, this test has 4 times more nodes. In this test, 8 has the highest probability of 24.4% followed by 7 (23%) and 9 (17.2%). The shape of the blue line tells us the search is done logarithmically.

To summarize, these graphs above proves the scalability of the Chord protocol. The logarithmic shapes and the small difference in hops between Figure 16 and Figure 17 show us the protocol is scalable in terms of searching and can be used in large systems.

Performance Evaluation: *This section we can see an Experimental design (explaining how the experiment is performed and how many, e.g. the number of nodes and lookup tests), Data Analysis (explaining the data result from the graphs, e.g. the shape has a logarithmic characteristics), Data Presentation (presenting the results in a graphical form) (8), (9), (10).*

7.1.3 Problems and Difficulties

Before we performed the lookup tests, we wanted to see how many nodes the simulation could handle on a single computer. We mentioned before in the beginning of this Chapter, the Java language could limit the number of nodes we could create due to its memory usage and performance. We were only able to reach 4,000 nodes with this application before it became unstable. If we used a faster language e.g. C++, the application would probably be able to handle more nodes, but at the cost of development time. Even so, if we wanted to reach a million nodes or perhaps just ten thousand, we would have to use faster hardware. However, 4,000 nodes are enough to justify that Chord can scale.

A more Chord related problem we discovered during our tests that is worth mentioning is that the protocol has problems with very high rates of churn. For example, if many nodes join simultaneously when the ring is empty, it takes a long time before the system becomes 100% stable. However, let us say that we already have x number of stable nodes in the ring and then another x joins the ring simultaneously. The time before all nodes become 100% stable is a lot faster. But if the new nodes were instead twice as big as the stable nodes in the ring, it would take a long time to stabilize the system. This is a known problem with Chord and is also mentioned in the paper this simulation is based on [5]. If a large quantity of nodes simultaneously join a small ring with stable nodes, many of the newly joined nodes will most likely have the wrong or the same immediate successor in the ring. This is

because one of the new nodes is probably the real successor of another new node, but has not yet been discovered. For a node to be used as a router during a lookup or to be discovered, it has to exist in other nodes finger tables. But a new node will not exist in any finger table until its immediate predecessor discovers it during the stabilize operation. The predecessor can only discover the new node if it exists as a predecessor of its successor. However, if many newly joined nodes have wrong successors; they will be declined as predecessors and therefore cannot be discovered until they have found their true successors. To find a new successor, the new node will set its current successor's predecessor to its new immediate successor during the stabilize operation (section 4.4.6) and notify it. This will repeat every stabilization call until the true successor has been found; but it could take a long time because the search is done in a linear behavior.

A challenging problem we came across when implementing this application was the key replication and the key management, after nodes joins and departs from the network. The paper mentions that the key replication can be done by utilizing the already existing successor-list, but the paper does not describe how it can be implemented. In the book "Peer-to-Peer Systems and Applications", by R. Steinmetz and K. Wehrle [18], different approaches are mention on how to use the successor-list for replication, but not in much depth. This made us implement our own solution by adding additional functionality.

7.2 Additional Functionality

As mentioned before, the application is based on an already existing Chord implementation that is presented in the paper [5]. However, as we acknowledged in the "Problems and Difficulties" section; the paper did not give us a detailed solution to the replication procedure, nor how to manage the keys when nodes joins and leaves the system. This forced us to implement our own solution based on the information that is presented in the paper.

The application utilizes the successor-list to perform replication of keys. Each node that is responsible for any key will replicate its keys to each node in that list, but never replicate its replicas. The replication happens every time a node has acquired a new key or when a new successor is added to the list. The key managing nodes will always keep replicates on its successors as long as they are responsible for those keys. If a key has been removed from a node due to a release of responsibility, that node will remove that key replica from each successor in the list. A similar scenario occurs if a node removes a successor: all its key replicas from that node will be removed.

We will present additional functionality which we implemented to solve these problems, but note that there are many ways to perform replication and key managing. This is not necessarily the most optimal solution.

Performance Evaluation**:** *This section defines and describes sub-mechanisms of the search mechanisms that affect the performance, but which are expected not to be major factors (5).*

7.2.1 Insert Key Function

When a node is joining the network, the node has to transfer its generated key to the node that is responsible for it, so that it can be located during a lookup. This is done by introducing a local function denoted as $n.insert(key)$. The function will find the successor in the ring that is responsible for maintaining this key and send it. The function is called after the node has received a successor in the join function (section 4.4.11).

```
//find the responsible node and send the key
n.insert(key)
    responsibleNode = find_successor(key);
    // Check which node is closest to the key
    responsibleNode = key.whichIsClosestToKey(n, responsibleNode);
    return responsibleNode.putKey(key);
```

7.2.2 Put Key Function

This function is called remotely and is denoted as $n.putKey(key)$. This function will first remove any replica that referrers to the key provided from this node, and add responsibility of the provided key. When the key has been added, the function will replicate this key to all its successors in the successor-list.

```
//sends a key to a node
n.putKey(key)
    successor_list.removeReplica(key);
    result = successor_list.insertKey(key);
    if(result is false)
        return false;
    //replicate the key on all successors
    successor_list.invokeReplicateAll(key);
    return true;
```

7.2.3 Put Replicas Function

A remotely called function denoted as $n.putReplicas(r[\,])$. The function is used to replicate keys to other nodes in the successor-list. The node will simply just add the replicas to its list. But if the node is already responsible for a key referring to the same replica, the replica will not be added.

```
//puts replicas into the list
//returns true if added
n.putReplicas(replicas[])
    return successor_list.insertReplica(replicas);
```

7.2.4 Modified Stabilize Function

The original stabilize function can be seen in the Chord section 4.4.12. This modified version of the stabilize function will now also manage a node's keys and replicas. Here the node will check and manage if it has any replicas that it should take responsibility for, or if it has keys it should release responsibility to, and it will also check if there are any keys that the node's successors is maintaining but should instead be maintained by the current node. We can categorize the key management into two parts presented in section 7.2.5 and 7.2.6 below.

7.2.5 Check Replication Responsibility

Early in the stabilize function, before the node manages its successor, the node will check if it has any replicas it should take responsibility for. The node does not know if the previous responsible node has involuntarily left the system, so it has to check if any of its replicas belong to it. To do that, the node will compare its ID with its immediate predecessor, to see which one is closer to each replica. If the node does not have any predecessor, it will check its successor instead. These replicas will be removed and added as keys instead, and the node will then replicate all its keys to the successor list.

7.2.6 Check Key Responsibility

At the end of the stabilize function, the node will check with the successors if it has any keys to release responsibility to or if the successors have any keys to take responsibility for. So basically, the node exchanges keys between its successors.

The transfer of keys must be done safely between the node and its successors. Both must agree on which keys to add or remove, and thus synchronize the exchange. The previously mentioned put key function is not safe enough; therefore we introduced another function that will be described in section 7.2.7 below.

7.2.7 Transfer Keys Function

This function is called remotely and is denoted as $n.transferKeys(keys\_add[\,], keys\_remove[\,])$. The function has two parameters $keys\_add$ and $keys\_remove$. Depending on the transfer, if it is a remove or an add transfer call, only one parameter is used and the other parameter is set to nil.

If it is a remove call, the function will remove all the keys provided in the parameter. Those keys that were successfully removed will instead be added as replicas. Since the responsibility of the removed keys now has been released from the node, all replicas referring to the removed keys will be removed from the successors. The function ends by returning the removed keys to the caller node. Similar scenario with the add call, but in an opposite manner.

```
//safely remove or add keys to a node.
//returns removed or added keys
n.transferKeys(key_add[], key_remove[])
    if(key_remove is not nil)
        removedKeys = successor_list.removeKey(key_remove);
        successor_list.insertReplica(removedKeys);
        //remove replicates from all successors
        successor_list.invokeRemoveReplicateAll(key_remove);
        return removedKeys;
    else if(key_add is not nil)
        successor_list.removeReplica(key_add);
        addedKeys = successor_list.insertKey(key_add);
        //replicate keys on all succesors
        successor_list.InvokeReplicateAll(addedKeys);
        return addedKeys;
    return nil;
```

7.2.8 Remove Replicas Function

This function removes replicas on a remotely called node and is denoted as $removeReplicas(r[\,])$. The function is called when exchanging key responsibilities or if a successor node from the successor-list has been removed.

```
//removes replicas provided in the parameter.
n.removeReplicas(replicasToRemove[])
   return successor_list.removeReplica(replicasToRemove);
```

8 Conclusion

The P2P technology offers many interesting features as we have discussed in this report. Chord in particular has great scalability and self-organizing features; not to mention the fast lookup functionality. It can find other peers quite fast in a large network and repair the system when nodes come and go. Because there is no central element that the peers depend on, it offers robustness and reliability as the system will not go down easily.

Because every peer in Chord is equal and none has any special privileges, the protocol makes it easier for the user to use the application. No special configuration is needed to create a network compared to the Client-Server protocol; where the server differs from the clients and has to be configured to serve the clients special needs. On the other hand, this makes it a lot more complex and harder to implement such system. Because every peer is independent, it is difficult to get an overview of the system to see how it performs and how data is being managed. This will make debugging, testing and performance evaluation of the system a lot harder.

During our implementation, we rediscovered Chord's Achilles' heel. As we mention in section 7.1.3, Chord has problems with high rates of churn. For example, if we had a thousand stable nodes inside the ring and if we simultaneously added another thousand nodes, the system would stabilize itself in a matter of seconds. The new nodes would find their true successors and therefore be found by other nodes relatively quickly considering the high quantity. But if we instead doubled the amount of new nodes, it would take the system a long time for all new nodes to find their true successors.

Evaluation

The aim of this thesis was to implement an application using a feasible decentralized P2P architecture that can be used in chat applications, - and to evaluate some of its performance. To be able to identify a suitable architecture, we had to go deeply into the P2P technology to discuss different architectures in existence and compare the capabilities of their algorithms with the requirements of a chat application. The purpose of this implementation was to get a better understanding of the technology and make it easier to identify problems and difficulties. These results would be used to help evaluate and see if the technology is a suitable choice for chat applications.

Based on the information and experience of this report, a decentralized P2P chat application could be a suitable choice for chat applications, but in different situations. The technology clearly consumes more power and memory for each user. Today, this could be a problem for mobile devices due to the battery depletion, as well as internet bandwidth consumption in large chat systems. In Client-Server based networks, the client usually works more like a dummy, which means it lets the server do the heavy work by processing queries, storing data etc. It is possible to put most of the work on the server, - and instead of communicating directly with the contacts (e.g. the Napster protocol), the client could send and receive messages directly from the server, thereby greatly reducing power consumption and internet bandwidth usage for the user. But choosing this approach in large systems would require more powerful hardware and internet bandwidth for the server, thus increasing the cost for the service provider.

In other situations, P2P could actually be an appropriate approach. For example, if the host does not have access to high-end hardware, it is an inexpensive choice for hosting a chat network with many users.

Research Questions

The research questions from section 1.4 will be reconnected in this area and briefly answered.

- How can one design and implement a chat application that does not rely on any centralized lookup service, and which class of lookup algorithms is capable to meet the key feature of decentralization?

In the Methodologies Chapter, we describe three concerns that defines a network application. We use these concerns to investigate requirements of a networked chat application and study different methods to achieve decentralization. The key feature of a decentralized application is the network is more resilient, due to the lack of dependency of other entities. This will make the network more robust and fault-tolerant. P2P based architectures can offer decentralization and some of them are discussed in Chapter 4.

An implementation can be done by selecting a P2P based architecture and select one out of three implementation methods: an actual system, a physical part or a simulation [17]. We choose to implement the system as a simulation, due to the time limit of this thesis. As we stated in concern (a), some functions and services of a chat application are irrelevant to implement, as they are regarded as trivial for the goal of this thesis.

- How can one analyze the performance of a decentralized chat system before it is deployed?

In the qualitative design method, in the first concern, we stated that chat applications should enable users to find the location of other users in a network (section 3.1.2). In this thesis we choose to call it "lookup service". The same service is used in networked applications to find resources or other entities. Searching within distributed P2P networks is an important performance factor and is usually analyzed when evaluating such systems (section 6.1).

To analyze the lookup service's efficiency, there were some choices of evaluation techniques [17] (the 7th step of a performance evaluation). The simulation method was chosen and an implementation was constructed. The simulation contains the necessary functionality in order to test the lookup service and analyze the data, to verify if the protocol can achieve scalability and search efficiently.

- How can one verify that the general search time of a DHT-based lookup algorithm is of order of $O(log\ N)$, with N is the number of peer/users?

Simulation data was extracted from several lookup tests with different amounts of nodes, described in section 7.1.2. The collected data was displayed in forms of graphs, showing the average probability of hops in x lookups. These graphs verifies the search is done logarithmically due to their shapes.

Future Work

The P2P technology is not so focused anymore, but it has contributed a lot to the research of distributed systems. The technology is often associated with illegal file sharing which has given it a bad name over the years. The technology will be more focused again in the future within telecommunications, but with a different name.

The security within P2P has not been discussed in this report and is not provided in the current application. A malicious peer could disrupt and damage the system by altering data and sending fake queries around the network.

Another possibility for future work is improving the replication and key managing mechanism in the application. The method we used to replicate and manage keys on other nodes has not been thoroughly analyzed and discussed. Because keys are important for the lookup functionality, they have to be quickly and efficiently managed when nodes join and leave the network. Currently, a node's key will never be removed when it has left the network, resulting in unnecessary data being maintained.

To test the scalability of the search algorithm in Chord, a lookup simulation was implemented. The simulation only performs one lookup at a time in a fully stable system. Additional simulation tests could be implemented in order to evaluate how the protocol performs during stabilization, key management or arrival and departures of nodes.

Appendix A

Application Contents

The application contents can be found at https://github.com/Edaenge/Chord_P2P_Simulation, or at the university (*Blekinge Institute of Technology*). Below is a tree structure in ASCII of the most relevant files of this application (e.g. the source code). The application also contains a "Microsoft Office Excel" file with a simple Visual Basic script, which was used to import simulation data.

```
+---ChordSimulation
|   |   Chord_Import.xlsm
|   |   Readme.txt
|   |
|   \---src
|       |
|       +---Chord
|       |   |   ChordId.java
|       |   |   ChordKey.java
|       |   |   ChordNode.java
|       |   |   ChordRemote.java
|       |   |   Entries.java
|       |   |   FakeRMICommunication.java
|       |   |   IDGenerator.java
|       |   |   KeyContainer.java
|       |   |
|       |   +---FakeRMIEvents
|       |   |   |   ClosestPrecedingFingerEvent.java
|       |   |   |   Event.java
|       |   |   |   FindSuccessorEvent.java
|       |   |   |   GetPredecessorEvent.java
|       |   |   |   GetSuccessorEvent.java
|       |   |   |   LookupEvent.java
|       |   |   |   NotifyEvent.java
|       |   |   |   PingEvent.java
|       |   |   |   PutKeyEvent.java
|       |   |   |   PutReplicasEvent.java
|       |   |   |   RemoveReplicasEvent.java
|       |   |   |   RetrieveKeysEvent.java
|       |   |   |   TransferKeysEvent.java
|       |   |   |
|       |   |   \---ReturnRMIevents
|       |   |           ClosestPrecedingFingerEventRE.java
|       |   |           FindSuccessorEventRE.java
|       |   |           GetPredecessorEventRE.java
|       |   |           GetSuccessorEventRE.java
|       |   |           LookupEventRE.java
|       |   |           NotifyEventRE.java
|       |   |           PingEventRE.java
|       |   |           PutKeyEventRE.java
|       |   |           PutReplicasEventRE.java
|       |   |           RemoveReplicasEventRE.java
|       |   |           RetrieveKeysEventRE.java
|       |   |           ReturnEvent.java
|       |   |           TransferKeysEventRE.java
|       |   |
|       |   +---FingerTable
|       |   |       Finger.java
|       |   |       FingerTable.java
|       |   |       SuccessorList.java
|       |   |
```

```
|       |   +---Manager
|       |   |       ChordLookupSimulation.java
|       |   |       ChordManager.java
|       |   |       CircleManager.java
|       |   |       InChordManager.java
|       |   |
|       |   \---Tasks
|       |       |   CheckPredecessorTask.java
|       |       |   FixFingersTask.java
|       |       |   InsertTask.java
|       |       |   JoinTask.java
|       |       |   StabilizeTask.java
|       |       |
|       |       \---Event
|       |               ClosestPrecedingFingerTask.java
|       |               EventTask.java
|       |               FindSucessorTask.java
|       |               LookupTask.java
|       |               NotifyTask.java
|       |               PutKeyTask.java
|       |
|       +---Crypt
|       |       Hashfunction.java
|       |
|       +---GUI
|       |       GUI.form
|       |       GUI.java
|       |       JTextAreaOutputStream.java
|       |       Quicksort.java
|       |       StabilityCheck.java
|       |
|       +---Process
|       |       ProcessEvent.java
|       |       ProcessX.java
|       |
|       \---Statistics
|               ChordLog.java
|               HopChordLog.java
|               HopData.java
|               HopLookupMeasure.java
```

References

[1] C. Huitema, "RIP P2P - Spotify moves to the cloud," 17 April 2014. [Online]. Available: https://www.mail-archive.com/p2p-hackers@lists.zooko.com/msg03496.html. [Accessed 4 October 2014].

[2] Skype, "About Cloud," 21 September 2014. [Online]. Available: https://support.skype.com/en/faq/FA12381/what-is-the-cloud. [Accessed 24 September 2014].

[3] Bittorrent, "About BitTorrent," BitTorrent Inc., [Online]. Available: http://www.bittorrent.com/company/about. [Accessed 20 April 2015].

[4] A. Loewenstern and A. Norberg, "DHT Protocol," 31 January 2008. [Online]. Available: http://www.bittorrent.org/beps/bep_0005.html. [Accessed 19 April 2015].

[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking,* vol. 11, no. 1, pp. 17-32, February 2003.

[6] B. Y. Zhao, H. Ling, J. Stribling, S. C. Rhea, A. D. Joseph and J. D. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications,* vol. 22, no. 1, pp. 41-53, 2004.

[7] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg,* pp. 329-350, 2001.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01 Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001.

[9] H. Zhang, A. Goel and R. Govindan, "Incrementally improving lookup latency in distributed hash table systems," *ACM SIGMETRICS Performance Evaluation Review,* vol. 31, no. 1, pp. 114-125, June 2003.

[10] J. Li, J. Stribling, T. M. Gil, R. Morris and M. F. Kaashoek, "Comparing the Performance of Distributed Hash Tables Under Churn," in *Peer-to-Peer Systems III*, Springer, 2005, pp. 87-99.

[11] D. Wu, Y. Tian and K.-W. Ng, "Analytical Study on Improving DHT Lookup Performance under Churn," in *Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference*, Cambridge, 2006.

[12] B. Leong and J. Li, "Achieving one-hop DHT lookup and strong stabilization by passing tokens," in *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference*, 2004.

[13] A. Gupta and B. Liskov, "Efficient Routing for Peer-to-Peer Overlays," *in Proceedings of the 1st Symposium on Networked,* pp. 113-126, March 2004.

[14] "Separation of concerns," [Online]. Available: https://en.wikipedia.org/wiki/Separation_of_concerns. [Accessed 28 08 2015].

[15] "Stateless Protocol," [Online]. Available: https://en.wikipedia.org/wiki/Stateless_protocol. [Accessed 02 09 2015].

[16] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, New York: Wiley, 1991.

[17] A. M. Law and W. D. Kelton, Simulation Modeling and Analysis, 3, Ed., Boston: McGraw-Hill, 2000.

[18] R. Steinmetz and K. Wehrle, Peer-to-Peer Systems and Applications, Berlin: Springer, 2005.

[19] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach, vol. Sixth Edition, Boston: Pearson, 2013, p. 581.

[20] Clip2, "The Gnutella Protocol Specification v0.4," 2001. [Online]. Available: http://www.content-networking.com/papers/gnutella-protocol-04.pdf. [Accessed 11 November 2014].

[21] "Napster Network," 1999. [Online]. Available: http://en.wikipedia.org/wiki/Napster. [Accessed 28 November 2014].

[22] "Napster shutdown extended," BBC News, 12 July 2001. [Online]. Available: http://news.bbc.co.uk/2/hi/entertainment/1435182.stm. [Accessed 28 November 2014].

[23] J. Ritter, "Why Gnutella Can't Scale. No, Really.," February 2001. [Online]. Available: http://www.cs.rice.edu/~alc/old/comp520/papers/ritter01gnutella-cant-scale.pdf. [Accessed 28 November 2014].

[24] D. Karger, E. Lehman, T. Leighton, R. Paingraphy and M. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," *STOC '97 Proceedings of the twenty-ninth annual ACM symposium on Theory of computing,* pp. 654-663, 1997.

[25] "Webiste of µTorrent," [Online]. Available: http://www.utorrent.com/. [Accessed 6 October 2014].

[26] Skype, "About Skype," [Online]. Available: https://support.skype.com/en/faq/FA10983/what-are-p2p-communications. [Accessed 21 August 2014].

[27] S. A. Baset and H. G. Schulzrinne, "An Analysis of the Skype Peer-to-Peer Internet," in *INFOCOM 2006. Proceedings*, Barcelona, 2006.

[28] P. Williams, "µChat: We Just Need Each Other," 30 June 2011. [Online]. Available: http://blog.bittorrent.com/2011/06/30/uchat-we-just-need-each-other/. [Accessed 23 May 2014].

[29] "Java Homepage," Oracle Corporation, [Online]. Available: https://www.java.com. [Accessed 14 12 2014].