The implementation and performance of Chord

Kasperi Kaija

Master's Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, October 1, 2017

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Kasperi Kaija

Työn nimi — Arbetets titel — Title

The implementation and performance of Chord

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	October 1, 2017	59

Tiivistelmä — Referat — Abstract

Chord is a distributed hash table solution that makes a set of assumptions about its performance and how that performance is affected when the size of the Chord network increases. This thesis studies those assumptions and the foundation they are based on. The main focus is to study how the Chord protocol performs in practice by utilizing a custom Chord protocol implementation written in Python.

The performance is tested by measuring the length of lookup queries over the network and the cost of maintaining the routing invariants. Additionally, the amount of data being exchanged when a new Chord node joins the network and how data has been distributed over network in general is also measured. The tests are repeated using various different networks sizes and states.

The measurements are used to formulate models and those models are then used to draw conclusions about the performance assumptions. Statistical measurements of quality are used to estimate the quality of the models.

The Ukko high performance cluster is used for running the Chord networks and to execute the tests.

Avainsanat — Nyckelord — Keywords
Chord, distributed hash table, P2P, distributed systems, networking, performance testing

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Chord is a distributed hash table solution, that provides a lookup service that maps an arbitrary key to a node in the network. It is a distributed algorithm, where the Chord nodes form a structured peer-to-peer network in a shape of a ring.

The Chord protocol makes assumptions about its performance. This thesis uses a Chord protocol implementation written in *Python* to test those assumptions and analyzes how well those assumptions hold in practice.

The Chord protocol implementation is built solely for the purpose of this thesis and the source code is stored in the *GitHub* web based service.

The performance is tested by measuring the number of times messages are being forwarded when executing lookup queries in a Chord network. The lookup queries are executed using different scenarios. In addition to a stable network state, scenarios where nodes are either missing part of their routing variables or have completely failed are used. Also the distribution of data items in the network as well as how the nodes behave when new nodes join the network is investigated.

Measurements are done by adding additional payload in the query messages to record data and by executing separate queries with the sole purpose to gather network statistics. In order to be able to create meaningful models, the measurements are repeated using several different network sizes.

Linear and logarithmic regression are used for model creation. The created models are compared to the expected behavior by utilizing statistical measurements of quality. Specifically the mean squared error and the coefficient of determination.

The measurement data is stored into files and then collected and processed into an applicable form and used for analysis. Python data processing libraries are used for preprocessing, model creation and analysis.

The chapters 2 and 3 provide the context and the background information to understand the purpose and the basic operation of the Chord protocol. The Chord implementation details are given in the chapter 5.

The research question and the approach is formulated in the chapter 4. The model formulation, analysis and results are explained in the chapter 6.

The test environment is the *Ukko* high performance cluster maintained by the Computer Science Department of the University of Helsinki.

The O-notation used in this thesis refers to the asymptotic upper bound of a function [CSRL01].

Unless otherwise specified, the logarithmic function $log\ x$ will always refer to $log_2 x$, the logarithm to base 2.

The intervals are specified according to [WS02]. The open interval $a < x < b$ is denoted with *(a,b)* and closed interval $a \leq x \leq y$ is denoted with *[a,b]*. The half open intervals $a \leq x < b$ and $a < x \leq b$ are denoted with *[a,b)* and *(a,b]* respectively. The first half open interval can also be called as right-open interval and the second one as left-open interval.

2 Distributed Hash Tables

A *peer-to-peer (P2P)* network is a network built using identical nodes, with each node providing service to one another [Tar10]. All nodes act as clients and servers to the other nodes.

A node refers to a process accessible in the network. A program running a service.

An overlay network is a logical network that runs on top of an existing physical network [Tar10]. The physical network can also be called as an underlay network.

In most cases the underlay is the *TCP/IP* network suite [Tar10]. Due to being connection oriented and reliable, TCP/IP or the Internet protocol suite *Transmission Control Protocol / Internet Protocol* [Pos81b, Pos81b] is well suited to serve as an underlay.

Internet routers, utilizing TCP/IP, are efficient in routing and forwarding messages between different network nodes. Routing is the process of building and maintaining the routing tables, that are used to identify where nodes lie in the network. Forwarding is the process of sending messages toward their destination in the network. [Tar10]

Overlay networks are useful in providing new functionality on top of an existing network, without the need to change the network infrastructure [Tar10]. This refers mainly to the need to change the routers responsible of the forwarding and routing functionality in the network. The new functionality could be e.g. an extension or a change in the default routing and forwarding behavior.

Overlay networks can increase the fault tolerance of routing and forwarding by providing several alternative paths between any two nodes in the network, and therefore providing added robustness. [Tar10]

The nodes of an overlay network are connected via logical links. These links do not need to be identical to the physical links that connect the physical network nodes together [Tar10]. Counted as hops, the physical distance between two nodes can be much longer than the logical distance. Logically two overlay nodes can be neighbors, but in the physical network those nodes can reside in computers located in the different sides of the world.

A hop refers to a message being forwarded between two neighboring nodes.

A *distributed hash table (DHT)* is a distributed algorithm providing a lookup service in a decentralized network. Nodes running the algorithm form a P2P network. The resulting DHT network can be structured, meaning that the logical topology of the network and the distribution of the data stored in the network, follow a strict set of rules. [Tar10]

A DHT network can also be unstructured, where the network topology is not controlled and forms a random graph [Tar10]. These kind of networks rely e.g. on flooding when sending messages between the nodes of the network. Unstructured networks are not covered in this thesis, because they are not relevant to the topic.

The data stored in a distributed hash table is processed as key-value pairs. The lookup service provides a function that finds the value for any given key by mapping the key to a node in the network. The data is distributed in the network in a manner that attempts to assure the efficiency of the lookups.

Keys are often represented as hashes of the stored values [Tar10]. A hash or a hash value is a fixed sized representation of an arbitrary sized input value [Sch07]. A function that takes an arbitrary sized input and provides a output value of fixed size representing that value, is called a hash function.

Each node of a DHT knows enough of the overall network topology, so that the lookup requests can always be forwarded to their destinations [Tar10]. The nodes are also able to adapt to constant change in the network, that is caused by nodes arbitrarily joining and leaving the network. This is achieved by having the nodes constantly update their routing tables.

It is efficient to implement a distributed hash table as an overlay, because the required high level routing and forwarding functionality is already provided by the TCP/IP protocol suite.

Due to the structured nature of the network, all lookups are guaranteed to find their destinations in a bounded number of hops in the logical network [Tar10].

In a hash table data structure, that maps keys to values and where an index in the data structure represents the key, a change in the size of the hash table requires that all stored data items need to be rehashed and restored [Tar10]. An index in the data structure can be thought as a bucket where the data item can be placed.

Consistent hashing is a technique providing hash functionality in a manner that does not require rehashing when the number of buckets change [SMK+01]. In the context of a DHT, when new nodes join or leave the network.

In a large network, it is unfeasible for every node to be aware of all the other nodes. A node only has a view of the network state that consists of a

subset of the nodes in the network. No node has the complete routing table that would contain the full state of the network, but the overall picture is distributed among the nodes.

Consistent hashing allows a node to map a key into one of the nodes in its view. This mechanisms allows the lookup requests to be forwarded to a known node, that is the closest possible node of the target in the current view. If the receiving node is not the final destination, it is at least closer to the target than the previous node. [KLL+97]

When nodes join or leave the network, the other nodes' view can change and they might need to adapt their routing invariants. Since a change in the network topology affects a partial view that is visible only to some of the nodes, only that subset of nodes need to be updated. Additionally, as the affect of the change is local, only a subset of the stored key-value pairs might be required to be moved to a new location.

Consistent hashing requires, that only K/N keys are remapped when a node joins or leaves the network. Here K is the total number of key-value pairs stored in the network and N is the total number of nodes in the network [KLL+97].

Chord is a distributed hash table that utilizes consistent hashing to assure load balancing by spreading keys evenly across the network [SMK+01]. Because all Chord nodes are equal, it is decentralized. It scales well, because the lookup length of finding a node in the network grows logarithmically to the number of nodes in the network.

Chord targets to simplify the design of P2P systems by addressing complex designs issues. These include how to balance load evenly across the system and decentralize responsibilities to avoid any single point of failure, how to scale well without losing too much performance and make sure that the service is available by constantly updating to adjust to nodes joining and leaving the network, and how to be application agnostic by placing no constrains on the structure of the keys. [SMK+01]

In a Chord network, the nodes form a logical ring, with each node having a specific and singular position on that ring [SMK+01]. Each node has a unique identifier of the length l, thus making the maximum size of the ring to 2^l nodes.

The Chord nodes update their routing invariants when nodes join or leave the network to assure that each node can be reached. The keys do not need to follow any specific structure, thus proving freedom on how data is being mapped to the nodes.

Chord uses the *SHA-1* hash function to assign identifier keys to the data items as well as to the Chord nodes themselves [SMLN+03]. A key of a data item is based on the content of the data. A key of a Chord node is based on

its network address.

The *Secure Hash Algorithm 1 (SHA-1)* is cryptographic hash algorithm [oST15], which means that it is designed to be a one-way function. For a one-way hash function, calculating the hash value from an input should be easy, but it should be computationally unfeasible to determine the input from a hash value.

Computationally unfeasible can be defined to mean, that the only way to find the answer is to use brute force and trying every possible variation to find the answer [Sch07]. An approach that can take millions of years to complete, even if one would utilize the entire computing power in the world.

One-way hash functions should also be collision resistant [Sch07]. This means that it should be practically impossible to find two distinct inputs that would produce the same hash value [RS04].

The SHA-1 uses a 40 digit long hexadecimal number as the hash value [oST15]. This equals to 160 bits and means that there are 2^{160} possible different hash values. In terms of Chord, this means that the maximum size of the logical Chord ring is 2^{160} nodes.

In principle, Chord could use any other hash function in addition to SHA-1 to provide identifiers to its nodes and data items, but in the context of this thesis only the use of SHA-1 is considered.

Additionally, the SHA-1 is no longer considered to be collision resistant. A successful collision attack has been performed against SHA-1, that required less computational operations than a brute force attack. [WYY05]

Even though the collision resistance of SHA-1 has been questioned, it is considered to have no impact on the study of performance of Chord in the context of this thesis. This is because the thesis does not concentrate on the security of Chord or the effectiveness of SHA-1.

Chord is but one distributed hash table solution. Others include *Pastry*, *Tapestry* and *Kademlia* to name a few [Tar10]. Each DHT solution has its own distinct features. The description of other DHT solutions with their similarities and differences to Chord is not in the scope of this thesis.

Whereas Kademlia is used in the *BitTorrent* P2P file-sharing protocol, the application of Chord has been mainly academic [Tar10].

3 The Chord protocol

The *Chord* distributed hash table solution is a distributed lookup protocol [SMK+01]. Its goal is to efficiently locate data items that have been stored in a decentralized network. The protocol tolerates constant joining and leaving

of nodes to and from the network.

Chord operates by mapping keys to nodes. Consistent hashing is used to assign identifiers to the data items. The same identifier space is used to give identifiers to the Chord nodes. Each Chord node, that has joined the network, has a unique identifier. Using *SHA-1* hash function makes it highly improbable that two distinct nodes or data items would have the same identifier [oST15].

A data item is mapped to the closest Chord node whose identifier is greater or equal to the identifier of the data item [SMK+01]. The Chord node can store the data item that has been mapped to it, thus providing a direct mapping between an identifier key and the data.

On average, a Chord node maintains information about $logN$ other nodes, where N is the total number of nodes in the Chord network [SMK+01]. This information is stored in specific routing invariants. The routing invariants are pointers to a node's predecessor and successor on the logical Chord ring as well as pointers to specific nodes further on the ring. These latter pointers are called *fingers*.

A lookup is resolved by finding the closest known node that precedes the target from the routing invariants and forwarding the lookup request to that node. If the node receiving the lookup request knows the target, then the lookup is directly forwarded to the target node. If not, then the request is once again sent to closest known preceding node and so on.

The cost of a lookup is measured in how many times the request has been forwarded to other nodes in the network. The expected lookup cost in hops is $O(logN)$ messages, where N is the number of nodes in the network. This is because the distance between the node currently forwarding the request and the target node halves every time the request has been forwarded [SMK+01]. This is behavior results from the structure of the finger table, that will be covered later.

A Chord node is in a stable state if all its routing invariants are correct according to the current network topology, or state, that is visible to that node. The Chord network is in a stable state if all Chord nodes are in a stable state.

In order for the routing to work, a Chord node only needs to have one correct routing invariant [SMK+01]. However, in such case the performance will degrade heavily. For the routing to always work, the *successor* pointer needs to be correct. It is possible for the routing to work even if the *successor* pointer would not be correct, if the *finger* pointer used to find the target of the next hop would still be valid. However, in such situation the correct routing behavior would be based only on chance, because the forwarded query would miss the broken pointer only based on the target of the query

and the state of the routing invariants.

A Chord node might be in a state were most of its routing invariants are incorrect, if a large number of different nodes are joining and leaving the network in its neighborhood and the node has not had the time to update.

When nodes join or leave the network, the Chord nodes are notified about the change and it is possible that there is a change to what data items are mapped to the existing nodes. In this case, the affected data items are moved to their new homes.

The use of a hash function that hashes its input uniformly across the output range is essential in keeping the keys mapped equally between the nodes in the network. If the keys are mapped uniformly, then the nodes in the network are all roughly responsible of $O(K/N)$ stored data items, where K is the total number of data items stored in the network and N is the total number of nodes in the network [SMK+01]. Additionally, when nodes join and leave the network, then roughly $O(K/N)$ data items will be moved from one node to another.

Chord is decentralized by having all the nodes being equal in functionality and therefore not having any single point of failure. Chord also scales well. The communication cost and state maintenance increases logarithmically with the number of nodes joining the network [SMK+01].

The logical network forms a ring of identifiers. If the length of the identifier is m bits, then the identifiers are placed on a ring from *0* to $2^m - 1$. In case of the 160-bit hash generated by *SHA-1*, the identifier starts from 0 and ends to $2^{160} - 1$. For simplicity's sake, it is assumed in this thesis that the identifiers always increase while traveling clockwise on the identifier ring.

The identifiers are ordered in *modulo* 2^m, meaning that successor node of the node with the identifier $2^m - 1$ is the first node on its clockwise side with an identifier ≥ 0.

The *successor* of a node is the closest node on the clockwise side and the *predecessor* the first one on the counterclockwise side.

When a new node joins, then the successor of that node will always transfer the responsibility of a set of stored data items to the new node. This is because a key is always mapped to a node that succeeds it on the identifier ring and when a new joins, it always comes between two existing nodes and takes the responsibility of the keys that fall in the interval between it and its predecessor. In a similar fashion, when a node leaves, then the responsibility of the keys mapped to it will be transferred to its successor.

All arithmetic operations, that are executed when determining the successors, predecessors and intervals, are executed in *modulo* 2^m.

An example Chord ring is shown in the figure 1. In the example, the key

length is set to 4 bits. This means that the possible identifiers range from *0* to *15*. The smaller black dots represent locations on the identifier circle and the bigger grey dots represent the two Chord nodes that have joined the Chord network. The boxes next to the grey dots represent the *finger tables* of the nodes.

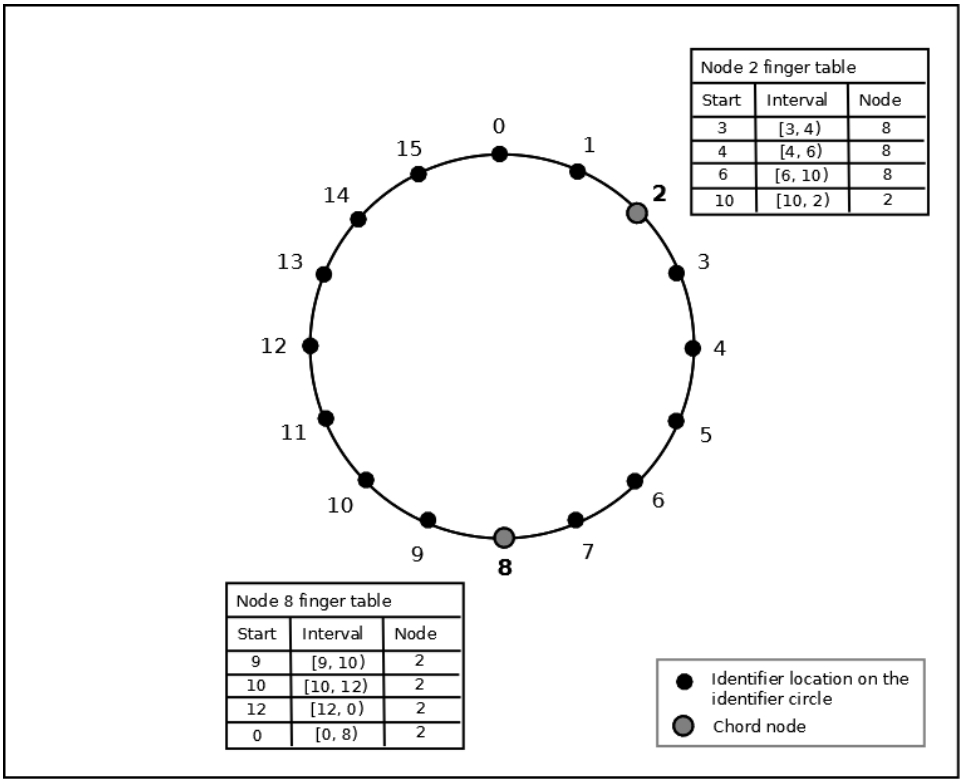

Figure 1: An example Chord network with two nodes and with the identifier length of 4 bits.

The *finger table* is a routing structure, or a routing invariant, that allows the lookups to traverse long distances on the identifier ring with one hop and without the need to go through all the nodes in between [SMK+01]. This is possible, because the finger table keeps pointers to nodes further than the immediate successor.

The finger table contains at maximum the same number of pointers as there are bits on the identifier. In case of SHA-1 *160* fingers and in the example in figure 1 *4* fingers.

Each finger table entry has three values. These are the *start*, *interval* and *node*. The *node* entry contains a pointer to the first node, that has an

identifier that is greater or equal to the *start* value of the entry.

The *start* value equals to $(n + 2^{k-1}) mod 2^m$, where n is the identifier of the node whose finger table is under investigation, k is the index of the finger table entry from $1 \leq k \leq m$, and m is the identifier length in bits. This way, each finger table entry points to a node that is at least 2^{k-1} positions further away on the identifier circle.

The *interval* value defines an interval that starts from the *start* value of a finger table entry and ends with the *start* value of the next finger table entry. More specifically, the interval that is *finger[k].start* $\leq x <$ *finger[k+1].start*, where k is the index on the finger table and the notation *finger[k].start* refers to the start value of the k^{th} finger table entry.

The *inteval* value is useful when looking for a successor node of an arbitrary identifier key.

The *node* value on the finger table contains the identifier of the node it points to and its address, that most likely is a combination of an *IP address* and a *port number*.

An example of the routing variables and their values are listed in the table 1. It contains the routing invariants of the Chord node *8* taken from the state shown in the figure 1.

The first entry on the finger table always points to the same node as the *successor* pointer.

Routing variable	*Value*		
Successor	2		
Predecessor	2		
Fingers	*Start*	*Interval*	*Node*
	9	$[9, 10)$	2
	10	$[10, 12)$	2
	12	$[12, 0)$	2
	0	$[0, 8)$	2

Table 1: Routing table of the Chord node with id 8.

The responsibility of a data item with an identifier k lies within the closest node whose identifier is smaller or equal to k. In another terms, the data item with the identifier k has a logical position in the identifier circle and the node that needs to be located is the next one on the clockwise side.

When lookups are executed, an arbitrary node sends the request to identify the target node. If the target node is the successor of the node executing the lookup, then the request can be trivially forwarded to the target. If not, the lookup query is sent to the node that is known to be the closest known

node that precedes the target. The expectation is that, the next node has more knowledge about the the target than the current node.

The node that is expected to have the best possible knowledge about the target, is the node in the finger table that immediately precedes the target. The closest known node on the counterclockwise side of the target.

The starting point of the lookup algorithm is the *find_successor* method shown in the listing 1 [SMK+01]. The method takes a target identifier as a parameter and returns information about the node where the target identifier is mapped to. It does this by searching for the immediate predecessor of the node it is looking for and returning the successor of that node.

The code shown in the listing 1 as well as in the other code snippets in this thesis are pseudo code based on the Python programming language [Fou17]. The Python syntax has been sacrificed in order to increase readability. The implementation details, that make the Chord implementation to work, and that are described in the paragraph 5, have been purposefully removed for the same reason.

The *self* in the code listings refers to node that is executing the method.

```
def find_successor(target):

  if (self.predecessor < target <= self.identifier):
    return self

  predecessor = find_predecessor(target)

  return predecessor.successor
```

Listing 1: The method for finding the successor.

The *find_successor* method uses the *find_predecessor* method to locate the immediate predecessor of the node it is looking for [SMK+01]. The *find_predecessor* method is shown in the listing 2. The method moves forward in the identifier circle, looking for a node where the target would lie between it and its successor.

The *find_predecessor* method requires to execute a *remote procedure call (RPC)* in order to query the closest preceding finger from another node further down on the identifier ring.

A remote procedure call is a request sent to another node in the network that triggers a part of the Chord algorithm to be executed on that node. In case of the *find_predecessor* method, it is required to access the full routing invariants of another node. Information that is accessible only by making a query over the network.

```
def find_predecessor(target):
  predecessor = self

  while not (predecessor < target <= predecessor.successor):
    predecessor
      = RPC(predecessor,
            closest_preceeding_finger(predecessor, target))

  return predecessor
```

Listing 2: The method for finding the predecessor.

Finding the predecessor depends on the *closest_preceding_finger* method shown the listing 3 [SMK+01]. The method checks the finger table entries starting from the last entry down and tries to identify an entry that would be the closest to the target but would still lie between the node executing the method and the target.

The remote procedure call is marked as *RPC* in the code listing 2. It takes two parameters. The first one is the node where the RPC should be executed and the second one is the method name that should be run on that remote node, the *closest_preceding_finger*. The return value of the RPC is the return value of the *closest_preceding_finger* method.

Each finger table entry doubles the distance from the node it belongs to. This way the theoretical distance to the target is halved with each iteration in the *find_predecessor* method. During the first iteration the maximum distance to a target is 2^m and within m iterations the distance has come down to 1 [SMK+01]. Each iteration either finds the predecessor it is looking for or asks from a new node further down the identifier circle of what it knows.

```
def closest_preceding_finger(target):

  for finger in reversed(self.fingers):
    if (self.identifier < finger.node < target):
      return finger.node

  return self
```

Listing 3: The method for finding the closest preceeding finger.

When a new node joins the network, the routing invariants of the affected nodes are updated. The affected nodes are those whose view of the network has been changed by the new node. The routing invariants need to be updated in order to make sure that each node in the network is reachable.

Each node that joins the network needs a known entry point to the network [SMK+01]. The entry point is another Chord node that has already joined and is able to retrieve information about where the new node will lie in the network.

The detailed algorithm of handling a join is described in [SMK+01], but the general idea is as follows.

The new node needs to retrieve information about its predecessor and successor nodes, as well as the nodes it needs to point in its finger table. Then it requires to update the routing invariants of the other affected nodes by informing them of the change. Lastly, the data items the new node is now responsible of needs to be transferred to it.

Updating the routing invariants of the new node, is done by utilizing an exiting and known node, that can be used to query the required information. This is done by repeatedly calling the *find_successor* method on the known node and storing the results.

Then the new node needs to inform other nodes of its existence, so that the finger tables of the existing nodes can be updated. This is achieved by sending a separate update message for each entry in the finger table. All those nodes who could theoretically have outdated information in their finger tables, need to receive the update message.

For each finger table entry, an update request is sent to the first node that precedes the new node by at least 2^{i-1} and from there it will travel counterclockwise in the identifier circle. The i is the index of the finger table entry from *1* to m, where m is the size of the finger table [SMK+01].

If the finger table entry on the node s receiving an update message points to a node e that succeeds the new node n, then n will replace the old finger table entry e. This is because a finger table entry entry needs to be an immediate successor for the *start* value of the finger in question.

If the finger table entry was updated, then the update request travels further back in the identifier circle. The update request stops traveling, when for the first time it was not necessary to update a finger table entry. This happens when a finger table entry precedes the new node n instead of succeeding it.

The predecessor pointer is used to travel the identifier circle in the counterclockwise direction [SMK+01].

The network state shown in the figure 1 has two nodes *2* and *8* forming a Chord network. A third node, node *12*, joins the network. Now the successor of *8* is *12* and the three first fingers of *8* all point to the node *12*. This is because the node *12* is the first node on the clockwise side of the start values of the finger table entries on the identifier circle. The new network state with new three nodes is shown in the figure 2 and the updated routing

invariants of the node *8* are shown in the table 2.

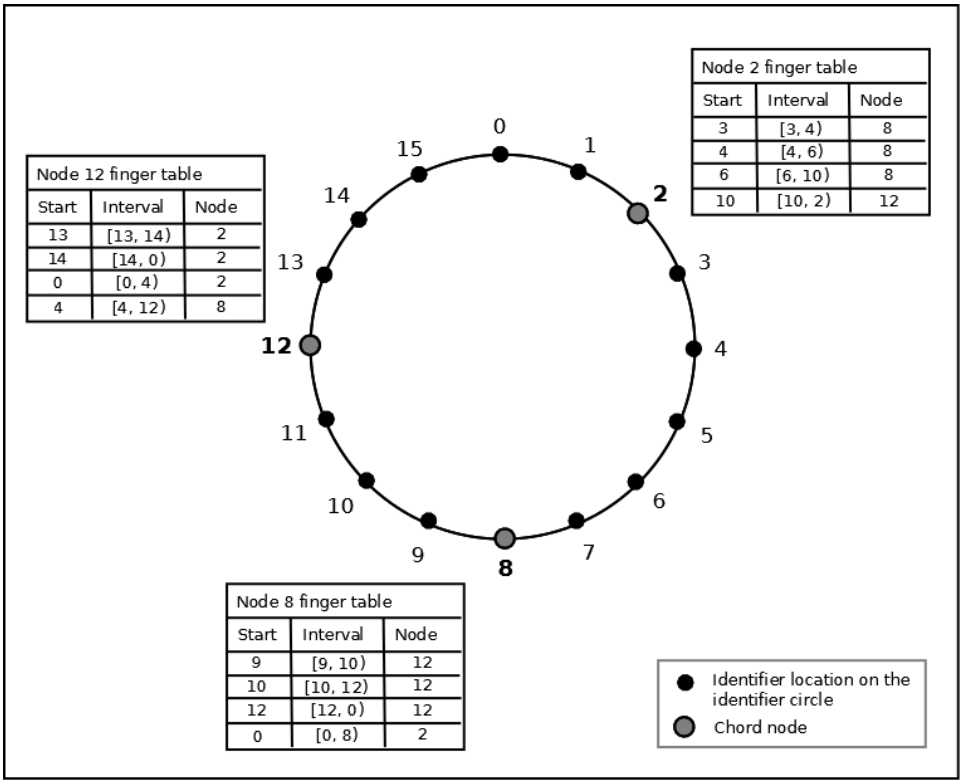

Figure 2: The example Chord network after node 12 has joined.

When nodes join or leave the network concurrently, it might be that not all routing invariant are correct. The concurrent joining and leaving refers to a change, that takes place so fast that there has not been enough time to update the routing invariant due to one change before another one occurs.

Concurrent changes are handled by periodically executing a stabilization routine on every Chord node in the network [SMK+01]. The stabilization routine running on the node n makes a query that checks what is its *successor's predecessor* p. This node p should be the same as the node n. However, p might be a new node that is now the *successor* of n, and n the predecessor of p. In a similar fashion, the finger tables entries need to be periodically refreshed.

Since concurrent performance is not in the scope of this thesis, the details of the stabilization routine will not be covered. The details are described in [SMK+01].

A Chord node fails, when it drops out of the network due to unexpected

Routing variable	*Value*		
Successor	12		
Predecessor	2		
Fingers	*Start*	*Interval*	*Node*
	9	$[9, 10)$	12
	10	$[10, 12)$	12
	12	$[12, 0)$	12
	0	$[0, 8)$	2

Table 2: Routing table of the Chord node 8 after node 12 has joined.

error either in the node itself or in the underlay network. If this happens, then it is important to be able to find the successor of the failed node in the network. This is mandatory for the routing to work.

One way to assure this, is for every node to keep track of more than one of their immediate successors on the identifier circle [SMK+01]. This way, if one of the nodes fail, then its predecessor already has the knowledge of the new successor it should point to. By default, the first finger table entry always points to the successor node, but there is no guarantee that the any of other fingers would point to the successor's successor. Therefore a separate list would need to be maintained.

To make sure that data items are not lost due to a node failure, data items could be replicated to several different nodes in the network.

If a lookup query can not be forwarded to its destination due to a failure, then the application using Chord could choose to wait for a while and retry [SMK+01]. It is expected, that with time the stabilization would correct the routing invariants and all nodes in the network would once again be reachable.

4 Testing the Performance

The Chord protocol [SMK+01] makes the following statements about its performance:

1. *"Each node in a Chord network is responsible for K/N keys, where K is the total number of keys stored in the network and N is the the total number of nodes in the network."*
2. *"When the (N+1)st node joins the network, the joining node receives $O(K/N)$ keys from its successor."*
3. *"Finding a successor of an arbitrary key in a N-node network requires*

the query to be routed through $O(logN)$ nodes."

4. *"When a node joins the network, $O((logN)^2)$ messages are needed to update the routing invariants of the affected nodes."*

5. *"When N new nodes, each without finger pointers, join an existing stable network of N nodes, the successor lookup of a key will still take $O(logN)$ messages."*

6. *"When half of nodes of a stable N node network fail and are disconnected from the network, the successor lookup of a key will still take $O(logN)$ messages."*

All the statements above are expected to be true with high probability [SMK+01]. This means that the statements can be expected to hold with probability at least $1 - 1/n^c$, for any $c \geq 1$ [Zü17] where n refers to the number of nodes in a network. So when the number of nodes n is high enough, then the probability will go to *1*.

A stable Chord network is a network where all the nodes have their routing invariants, including the finger tables, in a correct state.

The goal of the performance testing of the Chord protocol is to validate the above statements. This section defines the used test setup.

Certain aspects of the Chord protocol are out of scope for the testing. These are the normal leaving of a node from the network and the stabilization of the network after a node has left the network due to a failure. Also the concurrent joining and leaving of nodes is out of scope.

The normal leaving of nodes is expected to perform similarly to joining and it will be left out to simplify the testing. The stabilization, as well as concurrent joining and leaving, is left out to simplify the implementation of the Chord protocol and the test driver.

4.1 The test environment

The tests will be executed in the *Ukko* high performance cluster [UoH15] from the University of Helsinki's Computer Science Department. The cluster consists of 240 nodes. The Ukko node addresses range from *ukko001.hpc.cs.helsinki.fi* to *ukko240.hpc.cs.helsinki.fi*.

1 to 30 of the Ukko nodes will be used in the Chord performance testing. Each Ukko node running 10 to 500 Chord nodes.

The Ukko nodes can be accessed remotely through *melkinkari.cs.helsinki.fi* and *melkki.cs.helsinki.fi* interactive Linux servers of the Computer Science Department. Inside the Ukko nodes, the test processes are executed in

Screen window managers [Fou], that allows the processes to run even if the connection to a node would be lost and the session would be detached.

The IPv4 [(IE81] address of the used Ukko nodes and the available ports need to be known to configure the Chord nodes. This is achieved by using the utilities *ifconfig* [EKB+13] and *netstat* [EMH+13].

The Chord nodes and test scenario are controlled by using a set of computer programs that are executed in parallel in different Ukko nodes. Python version 3 [Fou17] will be used to execute the programs. The same Python version that was used to implement the Chord protocol and the test programs.

Git version control system [Git13] is used to maintain the test software in the Ukko cluster as well as for storing the results. The *GitHub* web-based version control repository [Git17] will act as the remote repository where the software and results will be stored. The url of the remote repository is *https://github.com/kasperi/DHT.git*.

The data items that are used in the testing are generated key-value pairs. The keys are used in the successor lookups. The keys and the data items are to be stored in the Chord network in stages. The data items have been generated by dividing novels in the Edgar Rice Burroughs' *Barsoom*, *Tarzan*, *Pellucidar* and *Caspak* series into full sentences and taking the SHA-1 hash values [oST15] of those sentences. The hash value is the key and the sentence is the value.

As an example, the first full story sentence of the book *A Princess of Mars* [Bur12] is *"I am a very old man; how old I do not know."* Its SHA-1 hash value in hexadecimal is *f4bbf309de29c0581727ed6b644e22cad35880df* and *139718515907647019090607546488578281868766219491l* as an integer. That means, that the first data item is the key-value pair (139718515907647019090607546488578281868766219491l, "I am a very old man; how old I do not know.").

The books were selected because they are in a digital format and in public domain. The books were retrieved from the *Project Gutenberg* web-based service [Gut71].

The books that were used are the *A Princess of Mars*, *The Gods of Mars*, *Warlord of Mars*, *Thuvia, Maid of Mars*, *The Chessmen of Mars*, *Tarzan of the Apes*, *The Return of Tarzan*, *The Beasts of Tarzan*, *The Son of Tarzan*, *Tarzan and the Jewels of Opar*, *Jungle Tales of Tarzan*, *Tarzan the Untamed*, *Tarzan the Terrible*, *At the Earth's Core*, *Pellucidar*, *The Land That Time Forgot*, *The People that Time Forgot* and *Out of Time's Abyss*.

It was possible to obtain 50000 unique data items from the books.

4.2 The test scenario

The test scenario starts by running 10 Chord nodes in 1 Ukko node and the state of the Chord network is saved in a configuration file. The configuration is used to save a stable state that can be easily restored. When the Chord network is stable, then 100 data items are stored into the network. After this the state is saved again using a different configuration name to separate the two saved network states.

All new Chord nodes join the network in a sequential order.

After the targeted stable network state with data has been reached, a measurement will be taken to determine the distribution of data in the network. Specifically, the number of data items stored in each node will be recorded.

Next, 100 new nodes will be added to the network. While the new nodes are joining, they are keeping track of the number of network messages that are needed to get all the routing invariants into a correct state. The joining nodes also keep track of the number of data items that they receive from their successors. After a successful join, the measurements will be stored in a results file.

After the test to obtain the cost of joining has been executed, the stored stable network state that includes the data items will be restored from saved configuration.

Next, 100 random successor lookups will be executed through a known node. The key list used for the lookups is the same one that contains the 50000 key value pairs that are used populate the network with data. The results, the number of hops each lookup made, is stored in a result file. The number of hops means the number of times the lookup query was passed from one node to another.

The network does not need to be populated with data in order to execute the successor lookups, because the data itself is not a prerequisite to find the node that is responsible of a certain identifier key value in a network of arbitrary Chord nodes.

The next step is to bring 10 more Chord nodes to the network, but this time they will not resolve their finger tables. These new nodes only know about their successors and predecessors. After this, the successor lookup is executed again and the results are stored in a separate result file from the previous successor lookup. When this is done, the network is shut down and restored from configuration to get rid of the nodes without the correct routing invariants.

After the network has returned to a stable state, 50% of the nodes are randomly shut down. The successor lookup with 100 random keys is executed

again and the results are stored in a separate result file. Lastly the whole network is shut down and restored from configuration.

This ends the complete test set for a Chord network of a certain size. The next step is to redo all steps with a bigger network. The amount of nodes and data items to add, as well as the number of successor lookups, depends on the target network size. In general, when the stable Chord network size is increased to N nodes, then the amount of data items is increased to *10*N* nodes and 100 successor lookup will be executed against three different variations of the Chord network.

The size of the Chord network will increase from 10 to 100 in 10 node intervals. From 100 the size of the network will be increased to 1000 in 100 node intervals. Starting from 1000 nodes to 6000 nodes, the network size will be increased in 1000 node intervals. From there on, two sets of 2000 nodes will increase the total size of the network to 10000 nodes.

The target is to reach a network that contains 10000 Chord nodes, that should be sufficient to identify those aspects of the Chord protocol performance that are logarithmic by nature.

Since the maximum number of available data items is 50000, the measurements to determine the distribution of data in the network will only be done with networks containing 5000 or less Chord nodes.

50000 is the number of generated data items that was possible to be retrieved from the source material. It would be possible to use randomly generated data items to supplement the existing ones, but this will not be necessary until the network size reaches 5000 Chord nodes.

Since the data items are used to measure the average load on one Chord node and the average moved data items when a new node joins, it is expected that using 5000 nodes is enough to find meaningful results. The expectation is, that the results should not fluctuate after the network size increases from 5000 nodes, since the ratio between the number of nodes and the number of data items is kept constant.

However, the data is expected to be more evenly distributed in the Chord network when the network size increases, due to the larger distribution of the nodes in the identifier key space.

The stable Chord network of 5000 nodes will also be the the maximum stable network, that will be used to add the new nodes without the finger tables. Adding the nodes without resolving the finger tables will double the network size and as based on the same reasoning used previously, a Chord network of 10000 nodes is expected to be big enough retrieve meaningful measurements.

The complete set of Chord nodes will be added to total of 29 Ukko nodes. 1 additional node will be used to execute the test that measures the cost of

adding 100 new Chord nodes to a stable network. The number of Chord nodes that will be run on the Ukko nodes will vary. The first 15 Ukko nodes will each run the maximum of 200 Chord nodes. The next 14 Ukko nodes will each run the maximum of 500 Chord nodes.

The number of Chord nodes that can be run in a stable manner in each Ukko node, is limited by the maximum number of open file descriptors a user can have. All Chord nodes will be run using the author's user account.

To summarize, the test set used with all network sizes is:

1. From a stable state, start new Chord nodes one at a time. First start all new nodes from one Ukko node before moving to the next.
2. Save the network state.
3. Add new data items to the network.
4. Save the network state.
5. Execute a query to get the number of data items stored in each Chord node.
6. Add 100 new nodes to the network to measure the cost of joining.
7. Shut down the Chord network and restore it from the configuration.
8. Execute the successor lookups for random keys.
9. Add new nodes without finger table entries.
10. Execute the successor lookups for the second time.
11. Shut down the Chord network and restore it from the configuration.
12. Shut down 50% of the Chord nodes randomly, but excluding the Chord node used as the entry point to the network.
13. Execute the successor lookup for the third time.
14. Shut down the Chord network and restore it from the configuration.

The number of nodes, data items and lookups that will be used in different stages of the testing is shown in the table 3.

Stage	Chord nodes	Ukko nodes	Data items	Lookups	Nodes without finger table	Failed nodes
1	10	1	100	100	10	5
2	20	1	200	100	20	10
3	30	1	300	100	30	15
4	40	1	400	100	40	20
5	50	1	500	100	50	25
6	60	1	600	100	60	30
7	70	1	700	100	70	35
8	80	1	800	100	80	40
9	90	1	900	100	90	45
10	100	1	1000	100	100	50
11	200	1	2000	100	200	100
12	300	2	3000	100	300	150
13	400	2	4000	100	400	200
14	500	3	5000	100	500	250
15	600	3	6000	100	600	300
16	700	4	7000	100	700	350
17	800	4	8000	100	800	400
18	900	5	9000	100	900	450
19	1000	5	10000	100	1000	500
20	1500	8	15000	100	1500	750
21	2000	10	20000	100	2000	1000
22	2500	13	25000	100	2500	1250
23	3000	15	30000	100	3000	1500
24	3500	16	35000	100	3500	1750
25	4000	17	40000	100	4000	2000
26	4500	18	45000	100	4500	2250
27	5000	19	50000	100	5000	2500
28	6000	21	n/a	100	n/a	3000
29	8000	25	n/a	100	n/a	4000
30	10000	29	n/a	100	n/a	5000

Table 3: The number of nodes, data items and lookups in different stages.

4.3 Collecting the results

The results are written in the directory *./dht/results* relative to the main program directory. There are four result subdirectories, one that contains all the results from the joining of new nodes, one for the lookup results, one for the statistics stating how many data objects are stored in each node of the network and one for tracking the key identifiers of the nodes in each saved configuration as well as the nodes having been shutdown.

The result files are written automatically by the programs running the tests. The structure of the results files are such that they can be programmatically combined and converted into a *comma-separated values* (*CSV*) [Sha05] file. In practice this means that all results are stored in files in *JavaScript Object Notation* (*JSON*) [Bra14] form.

Later, during the analysis of the results, the CVS formatted files are manipulated using standard Python libraries and the *NumPy* library [dev17]. The NumPy library provides a set of mathematical functions to help with the preprocessing of the data for analysis.

The results from the join operation contain the ordinal number of the node, the number of messages that was required to setup the correct routing invariants and number of data items that were moved to the new node. There will be one result file for each node that joined the network in a normal fashion. No result files are written if the nodes were started from a stored configuration or to a state that was not valid. The file names are based on the Chord node id and the test stage the result belongs to.

The results from the successor lookups contain the amount of nodes the lookup passed through when looking for the successor of a random key stored in the Chord network. One file contain all the results for all lookups done during one test stage. The test stage, referring to the the current network size, is identifiable from the name of the file.

The results from the statistics query contains information about the state of each node in the network. This includes the information on how many data items has been stored to the node and how many unique Chord nodes are referenced from its finger table. There will be one file for each stage of the test.

The result files are added and stored in the same *Git* repository where the source code is stored and pushed to the remote repository for safekeeping.

4.4 Analyzing the results

The numerical measurement results stored in the CVS files are loaded and manipulated using the NumPy library when applicable. When the results

contain more than just numerical values, then the results need to be processed using standard Python libraries first. Since the NumPy library provides powerful mathematical tools for data processing, its use is preferred to the the standard Python libraries.

The *Scikit-learn* machine learning library [hll17] is used to find the regression models from the preprocessed data.

The Scikit-learn and the Numpy libraries are obtained using the *Anaconda* Python distribution [Ana17]. The Anaconda distribution provides the common machine learning, mathematical, data analysis, data manipulation and plotting libraries in a single readily configured package.

The *matplotlib* library [dt17] is used to visualize the results data and models.

4.5 Interpreting the results

The target of the performance testing is to validate the performance statements given in the beginning of the section 4. This section defines how the test results are to be interpreted.

On average, the number of keys each node is responsible for will remain constant relative to the network size because the ratio of nodes to data items is kept the same. More interesting question is how equally the data items are distributed throughout the network. It can be assumed that in bigger networks the data items will be distributed more equally than in smaller networks. This is because it is less probable in bigger networks to have the Chord node key identifiers to be clustered in one particular area of the identifier space. This assumption is based on the assumption that the SHA-1 hash function is capable of mapping identifiers uniformly over its output range.

The number of keys a node receives from its successor on average, should also remain constant relative to the network size. Also in this case as above, it is assumed that joins in smaller networks will differ more than in bigger networks, because the data items are expected to be more evenly distributed in bigger networks.

The identifier lookups are expected to take on average $O(logN)$ messages, where N is the number of nodes in the network. The results are to be plotted for visual inspection to try to verify this. In addition, logarithmic regression and linear regression will be used to model the relationship of the number of messages and the network size. It is assumed that the logarithmic regression will provide a better fit, when comparing the two model with each other. The models will be assessed by comparing the *mean squared error (MSE)* [JWHT14] and the *coefficient of determination (R^2)* [Woo15].

Logarithmic regression models are created by applying logarithmic transfor-

mation of the independent variable x and then applying linear regression to the result. Where the normal linear regression model is of the form $y = b * x + a$, the logarithmic regression model will be of the form $y = b * log(x) + a$. The logarithmic function *log* will always refer to the logarithm of base 2, $log_2 x$, unless stated otherwise.

The mean squared error is an often used measure to assess the quality of fit of regression models [JWHT14]. The MSE gives a measure how close the predictions the model gives are to the actual measurements. The closer to zero an MSE value is, the better the fit.

The coefficient of determination is also a commonly used quality measurement of regression models [Woo15]. The R^2 gives a measure how well the model explains the measurements based on variance. In general, the R^2 value lies between 0 and 1. The closer to 1 the value is, the better the fit.

The number of network messages needed to establish stable routing invariants is expected to be $O((logN)^2)$, where N is the number of nodes in the network. Similarly to above, logarithmic and linear regression will be used be used to model the relationship between the number of messages and the network size. However, since the relationship model between the number of messages and the network size is expected to be a polylogarithmic function, more complex non-linear regression or preprocessing might be necessary to accurately fit the model.

4.6 Test programs

Several different helper programs are needed for starting new Chord nodes, starting them from an existing configuration, uploading data items to the network, saving the network state, retrieving the node statistics and executing the identifier key successor lookups.

The program that starts new Chord nodes is *start_servers* and its usage is:

```
python start_servers.py [-h] [-b BASE] [-k KNOWN]
[-n NUMBER] [-f] [-c COUNT] [-s SLEEP]
```

The optional arguments are:

```
-h              show help message and exit
-b BASE         base ip address with port
-k KNOWN        known ip address with port
-n NUMBER       number of servers to be started
-f              init the first known server
-c COUNT        count of previously started servers
-s SLEEP        polling interval in seconds when
                waiting servers to join
```

If optional arguments are not given, then a default configuration will be used. The default configuration are stored as constant values as a part of the program.

The program keeps track of the number of network messages that are used to establish the routing invariants when a node joins a Chord network. The results are stored in a file in *./dht/results/join* relative to the main program directory.

In example, to start *10* Chord node servers from a Ukko node with IPv4 address *86.50.20.7* starting from port *50000* with a known Chord node server running at *86.50.20.50* in port *51001*, the correct command line parameters would be:

```
python start_servers.py -b 86.50.20.7:50000
-k 86.50.20.50:51001 -n 10
```

All programs are required to be given a known entry point to the Chord network with the known option *-k*.

Each program provides more help on its usage when given the help argument *-h*.

The program to start Chord nodes from a configuration is
start_servers_from_conf. The configuration argument *-c* is used to define the scenario name identifying the configuration or state to be loaded. The configuration name is defined when saving the the state of a Chord network. Each Chord node will load its configuration from an individual file located in *./dht/conf/* relative to the main program directory. The program usage is:

```
python start_servers_from_conf.py [-h] [-b BASE]
[-k KNOWN] [-n NUMBER] [-c CONF]
```

When running, the *start_servers_from_conf* can be given commands from the command line. It can be given a command to shutdown either all the nodes it started and are running or only half of them.

The program to start Chord nodes servers without fully resolving the finger table is
start_servers_without_fingers and its usage is:

```
python start_servers_without_fingers.py [-h] [-b BASE]
[-k KNOWN] [-n NUMBER] [-c COUNT] [-s SLEEP]
```

The above programs are meant to be used in cooperation. First a stable network state is to be loaded with *start_servers_from_conf* and then the network size can be increased by using *start_servers*. Alternatively, *start_servers_without_fingers* can be used on top a stable network to create an unstable network for performance testing purposes.

The program *upload_data* is used to upload data items to an existing and stable Chord network. The program randomly selects data items to be uploaded from the master data item file located in *./dht/data* relative to the main program directory. It keeps track of the data items that have already been uploaded, to make sure that it does not attempt to upload the same data item twice. The program usage is:

```
python upload_data.py [-h] [-k KNOWN] [-n NUMBER]
```

The program used to retrieve the information about how many data items are stored in each Chord node is called *get_node_statistics*. The program stores the results in a file in *./dht/results/statistics* relative to the main program directory. The program usage is:

```
python get_node_statistics.py [-h] [-k KNOWN]
[-s SCENARIO]
```

The *save_network_state* program is used to save a stable Chord network state into a set configuration files. One file for each node in the network. Its usage is:

```
python save_network_state.py [-h] [-k KNOWN]
[-s SCENARIO]
```

The *find_successor_lookup* program executes a number of key lookup queries to a Chord network and measures the number of Chord nodes it had to traverse through to find the correct node. The results are stored in a file in *./dht/results/lookup* relative to the main program directore. The program usage is:

```
python find_successor_lookup.py [-h] [-k KNOWN]
[-n NUMBER] [-s SCENARIO] [-r RUN]
```

5 The Chord protocol implementation

The architecture of the Chord implementation aims to separate the concerns of the applications specific business logic, the application independent business logic and the interactions with the outside world.

The application specific business logic defines the core behavior of the system. The application independent business logic refers to all reusable code that is not specific the system that is being constructed. The outside world refers to objects that the system needs to interact with, like the user, the *Ukko* cluster and the Internet.

In its simplicity, the software is divided into control, entity and interface objects [Jac92].

The control objects contain the Chord specific business logic, including the interaction of several entity objects. The entity objects are domain objects that are independent of the application business logic. They hold information that is not volatile and the behavior coupled with that information. The interface objects are used by control objects to interact with the outside world.

The control objects depend on the entity and interface objects. In principle, the interface objects should be replaceable without affecting the behavior of the control objects.

Having a good separation of concerns allows the different objects to be testable independently from the overall system. Utilizing good design principles, such as the Dependency-Inversion Principle [Mar03], that states that objects should depend on abstractions, allows the use of mock objects as dependencies during testing.

Mock objects are test specific objects that are used to verify the behavior of the target object independently from the overall system [Mes07].

The figure 3 visualizes the high level architecture of the implementation. It is in the form of a *Unified Modeling Language (UML)* class diagram [Gro15], that visualizes the static structure of the software.

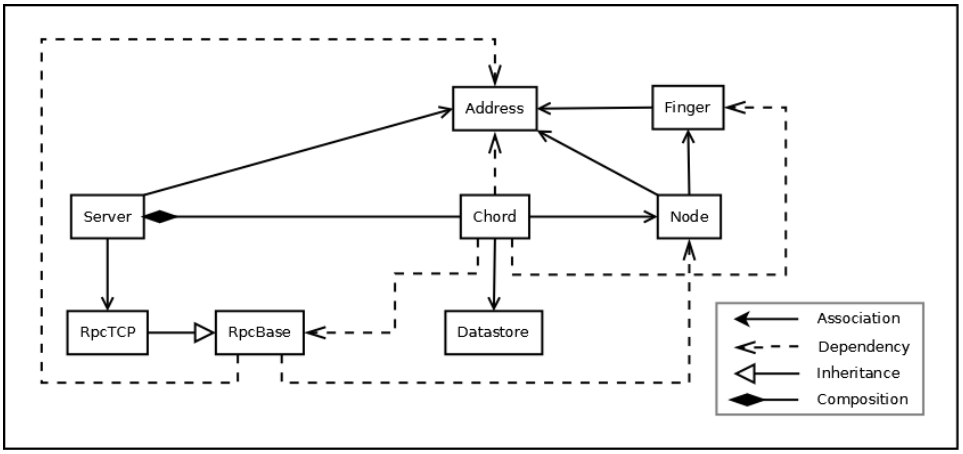

Figure 3: The class structure of the Chord implementation.

The main control object is the *Chord* class. It contains the Chord protocol logic and maintains its state in the *Node* entity object that represent a Chord node instance. The *Finger* and the *Address* entity objects are used by the *Chord* control object, but the *Node* object keeps track of them.

The *Datastore* interface object is a representation of storage that the *Chord*

control object is using to retrieve and store data. In this implementation it is a simple dictionary data structure composed of key-value pairs.

The *Server* interface object is responsible of the runtime context. It and the *RpcTcp* object contain the logic how different Chord nodes find each other and communicate. The *RpcBase* is an abstract class that defines required methods the Chord nodes use to interact with each other but does not contain any implementation details. The communication between nodes is done using *Transmission Control Protocol / Internet Protocol (TCP/IP)* protocols [Pos81b] [Pos81a], but the Chord protocol is unaware of its or Ukko cluster's existence.

The *Address* and *Node* entity object are used as common request and response models in the communication between the *Server* interface and the *Chord* control objects. The other data moving between the interface and control objects and between the Chord nodes in the network is *JavaScript Object Notation (JSON)* [Bra14] encoded data. All entity data, whether being stored into files or transmitted over the network as a byte stream, is first serialized into JSON.

The *Server* interface object is a threading TCP server that queues, verifies and parses the incoming requests and then forwards them to the *Chord* control object.

The software development process relied on utilizing *Test Driven Development* [Mar03], where the tests are implemented before the production code. This has the effect that the resulting production code will be completely tested.

In total, 74 automated test were implemented. These test can be divided into two categories, unit and integration tests. The unit tests concentrate on testing an individual method of a specific class, whereas the integration tests are used to test the coordinated behavior of several classes working together [You08].

The automated test act as a specification and documentation to the code [Mes07]. They also provide a safety net against regression when the code is being changed.

The Python *unittest* library [Fou17] was used to implement the automated test. The tests cover *64%* of the source code lines of all control, entity and interface objects in the protocol implementation. Of the *Chord* control object code, *93%* of the source code lines are covered. The code coverage measurement was taken using the code coverage tool of the *PyCharm* integrated development environment (IDE) [s.r17].

The core Chord protocol implementation contains *1324* lines of Python code. When including the utility programs, helper and test classes the overall implementation contains *7902* lines of Python source code. The measurement was taken using the *statistics* plugin [Top17] of the PyCharm IDE.

The stabilization routine described in [SMK+01] was not implemented, so concurrent joining and leaving of nodes was not supported. However, this had no effect on the performance testing done in the scope of the thesis. The main benefit would have been, that the building the Chord networks used for the testing could have been faster.

The handling failed nodes was based solely on the use of finger tables, meaning that no additional successor list was used in the nodes.

The identifiers for the Chord nodes were constructed by taking the string formatted IP address and port number of the Chord node's address and joining them together using a colon (":") and taking the SHA-1 hash value of the resulting string.

It was necessary a add functionality in addition to the basic Chord protocol, in order to support the gathering of statistics. This functionality triggered queries, that went through all nodes of the network using the successor pointer and either caused configuration files to be written to the file system or a response to be returned that contained the data that had been requested.

6 Test Results and Analysis

This paragraph contains the analysis of the test measurements. Each of the six performance statements listed in the paragraph 4 are covered separately.

6.1 Keys stored in the nodes of the network

The distribution of data items in the Chord network was measured according to the plan specified in the paragraph 4. Measurements were taken on 23 different Chord network configurations, varying from 10 to 5000 nodes in size. The total number of data items stored in a Chord network was always 10 times the number of Chord nodes in the network. E.g. the network of 5000 nodes contained 50000 data items.

In the context of the analysis, the term *key* refers to a data item if not stated otherwise.

The measurements taken for each network configuration are visualized in the box plot in figure 4. The average, median, minimum and maximum number of data items stored in a single Chord node with respective to all used network configurations are listed in the table 4.

The top and bottom edges of the boxes in the boxplots used in this theses are containing values between the upper and lower quartiles, also known as the *interquartile range (IQR)*. The band inside the box represents the median. The whisker contain the data between the *1.5 * IQR* of the lower quartile and

*1.5 * IQR* of the higher quartile. The points outside the whiskers represent individual data points, the outliers beyond the defined range.

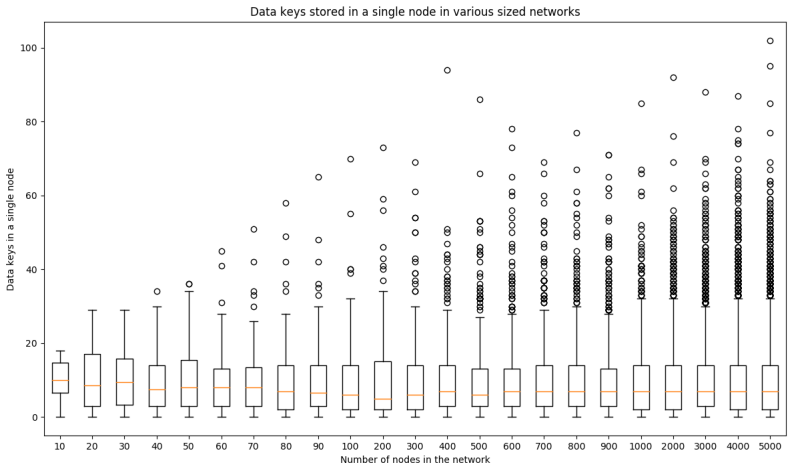

Figure 4: The number of data items (keys) stored in a single node in various sized networks.

The stored data items were randomly distributed in the network. If the identifier keys of the Chord nodes and the used data item keys would be evenly distributed to the identifier key space, it could haven been possible that all nodes would contain the expected $K/N = 10$ data items. However, this is highly improbable in real life. As can be seen in the figure 4, actual distribution of data items in the network is never even.

As listed in the table 4, the Chord node that has the most data items contains 18% of the total data items in the Chord network when the network size is 10. The percentage of total data items stored the single node containing most data items gradually goes down to 0.204% in the 5000 node network. This is to be expected. When the number of nodes increases, so does the probability that the data items are evenly distributed. It is highly probable that the median will remain around and under 10 when the network size increases from 5000 nodes, because there will always be more nodes that have less than 10 data items than those that have more.

To elaborate, for every Chord node that takes more than its optimum share of 10 data items, there must be at least one that has less. If a node has more than 20 data items, then there must be at least two nodes that have less than 10 data items and so on.

The first statement about the performance of Chord in section 4 states that:

"*Each node in a Chord network is responsible for K/N keys, where K is the total number of keys stored in the network and N is the the total number of nodes in the network.*" This is interpreted to mean that each node is responsible on average K/N keys, since in practice it is impossible for each node to contain exactly K/N keys due to the random distribution of node id keys and data item keys in the id key space.

The average median value of the medians listed on the table 4 is *7.13* and as explained above, this is expected due to the skewness of the data item distribution caused by a set of outlier nodes each containing a large number of data items. As it is highly probable that the median value will close on the average value of K/N, the first performance assumption is considered to be true.

Network size (nodes)	Total data items in network	Expected data items	Avg data items in one node	Median data items in one node	Min data items in one node	Max data items in one node	Max % of total data items in one node
10	100	10	10	10	0	18	18
20	200	10	10	8	0	29	14.5
30	300	10	10	9	0	29	9.67
40	400	10	10	7	0	34	8.5
50	500	10	10	8	0	36	7.2
60	600	10	10	8	0	45	7.5
70	700	10	10	8	0	51	7.29
80	800	10	10	7	0	58	7.25
90	900	10	10	6	0	65	7.22
100	1000	10	10	6	0	70	7
200	2000	10	10	5	0	73	3.65
300	3000	10	10	6	0	69	2.3
400	4000	10	10	7	0	94	2.35
500	5000	10	10	6	0	86	1.72
600	6000	10	10	7	0	78	1.3
700	7000	10	10	7	0	69	0.986
800	8000	10	10	7	0	77	0.963
900	9000	10	10	7	0	71	0.789
1000	10000	10	10	7	0	85	0.85
2000	20000	10	10	7	0	92	0.46
3000	30000	10	10	7	0	88	0.293
4000	40000	10	10	7	0	87	0.217
5000	50000	10	10	7	0	102	0.204

Table 4: Number of data items (keys) stored in a node with different network sizes.

6.2 Nodes receiving keys when joining the network

The number of data items being transferred to a node when it joins a stable Chord network was measured according to the plan. The gathered measurement data was used to calculate the average transferred data items in relation to the size of the Chord network. The average transferred data items in relation to the network size is visualized in the figure 5.

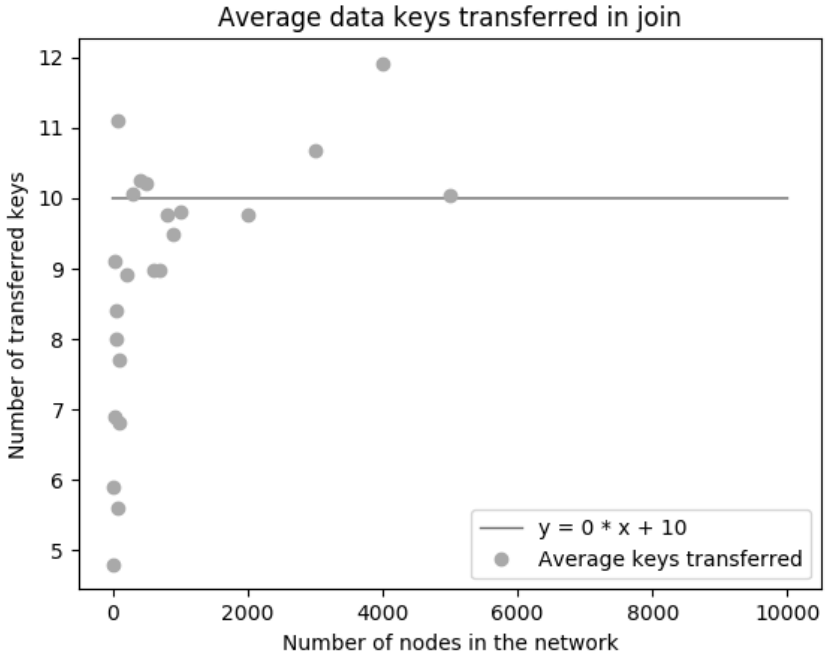

Figure 5: The average number of data items (keys) transferred when a node joins in relation to various network sizes.

The use of average transferred data items is a good approach, because the individual measurements inside the each network configuration can vary greatly. This is visualized in the figure 6 that shows the number of transferred data items for the first measurement in each network configuration.

The term network configuration is synonymous with network size in the context of the analysis.

The measurements taken for each network size are visualized in the box plot in the figure 7. The average, median, minimum and maximum data items that were transferred respective with the used network sizes are listed in the table 5 to help interpret the figure 7.

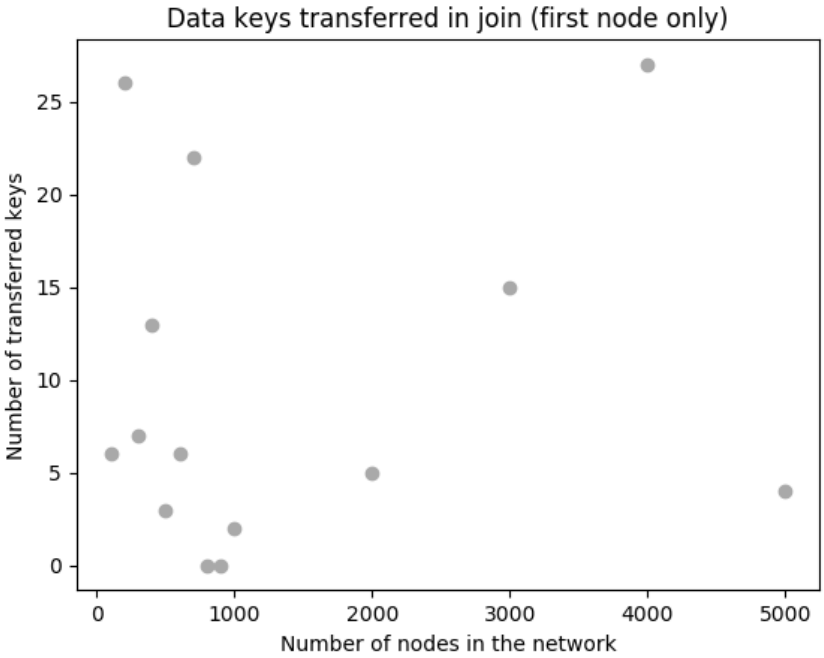

Figure 6: The number of data items (keys) transferred when the first node joins.

The average transferred data item count was compared to the expected model of $y = 0 * x + 10$. The expectation was that the joining node would receive $O(K/N)$ data items, where K is the number of data items and N the number of nodes in the network. Since the number of data items was always ten times the number of nodes with all used network sizes, the expectation was that $O(10)$ data items would be received.

In practice the number of received data items should be less than $O(10)$, since the measurements where taken so that 100 new nodes would join to a stable network and no new data items would be added between the joins. This means that the number of data items would remain what it was in the beginning of a test run with a certain network size, but the ratio of K/N would gradually go down. However, the effect is smaller with the larger network sizes than with the small ones, because the new 100 nodes constitute a smaller portion of the total number of nodes in the network when the network size increases.

The random distribution of the data items in a Chord network caused a lot of noise in the networks with a small number of nodes. In attempt to

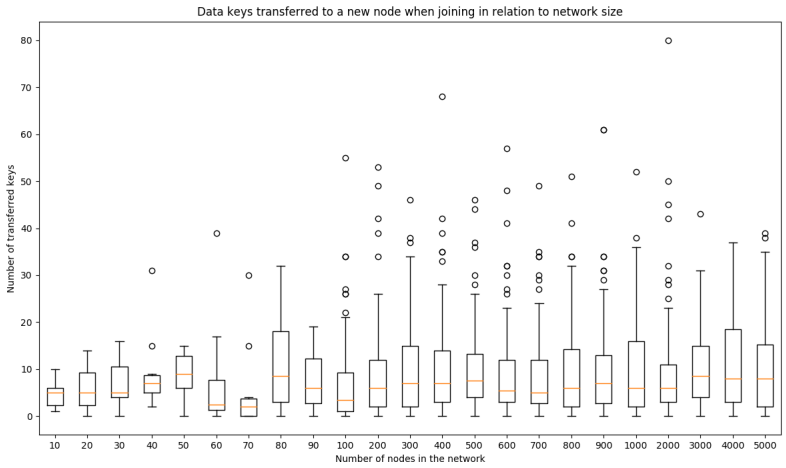

Figure 7: The number of data items (keys) transferred when a node join in relation to various network sizes.

minimize the effect of the noise, only networks with 500 nodes or more where used measure the quality of the fit with the mean squared error (MSE) and the coefficient of determination (R^2) values against the expected model. The results are listed in the table 6.

The expected model has the MSE of *0.665* and R^2 of *-0.00229*. Considering only the MSE, based on the table 6, the mean squared error increases when the expected number of received data items move away from 10. This happens with values below and above 10. Based on this $O(K/N) = O(10)$ is a good estimate.

Based on [Woo15], R^2 can be negative if the fit of the model to the data is purely imaginary.

The negative R^2 values in the table 6 are likely caused by the fact, that the data points visualized in the figure 5 and used to calculate the R^2 were not used in the model generation. The R^2 values were ignored when estimating the quality of fit for the expected model.

The second statement about the performance of Chord in section 4 states that: "*When the (N+1)st node joins the network, the joining node receives $O(K/N)$ keys from its successor.*" Based on the analysis above, this assumption is true.

Network size (nodes)	Total keys in network	Avg transferred	Median transferred	Min transferred	Max transferred
(10, 110]	100	4.8	5	1	10
(20, 120]	200	5.9	5	0	14
(30, 130]	300	6.9	5	0	16
(40, 140]	400	9.1	7	2	31
(50, 150]	500	8.4	9	0	15
(60, 160]	600	8.0	2	0	39
(70, 170]	700	5.6	2	0	30
(80, 180]	800	11.1	8	0	32
(90, 190]	900	7.7	6	0	19
(100, 200]	1000	6.82	4	0	55
(200, 300]	2000	8.91	6	0	53
(300, 400]	3000	10.06	7	0	46
(400, 500]	4000	10.26	7	0	68
(500, 600]	5000	10.21	8	0	46
(600, 700]	6000	8.97	6	0	57
(700, 800]	7000	8.99	5	0	49
(800, 900]	8000	9.76	6	0	51
(900, 1000]	9000	9.49	7	0	61
(1000, 1100]	10000	9.8	6	0	52
(2000, 2100]	20000	9.76	6	0	80
(3000, 3100]	30000	10.68	8	0	43
(4000, 4100]	40000	11.91	8	0	37
(5000, 5100]	50000	10.04	8	0	39

Table 5: The number of data items transferred to a node when it joins related to the network size.

Model	MSE	R^2
y = 0 * x + 13	9.9	-13.9
y = 0 * x + 12	4.82	-6.26
y = 0 * x + 11	1.74	-1.63
y = 0 * x + 10	0.665	-0.00229
y = 0 * x + 9	1.59	-1.39
y = 0 * x + 8	4.51	-5.79
y = 0 * x + 7	9.43	-13.2

Table 6: The expected and comparative models for the moved data items when a node joins and their respective statistical measurements.

6.3 Finding a successor in a stable network

The lookup length measurements in a stable Chord network were executed as planned. The gathered measurement data was used to calculate the average lookup length for each Chord network size. The average lookup length in relation to the network size is visualized in the figure 8.

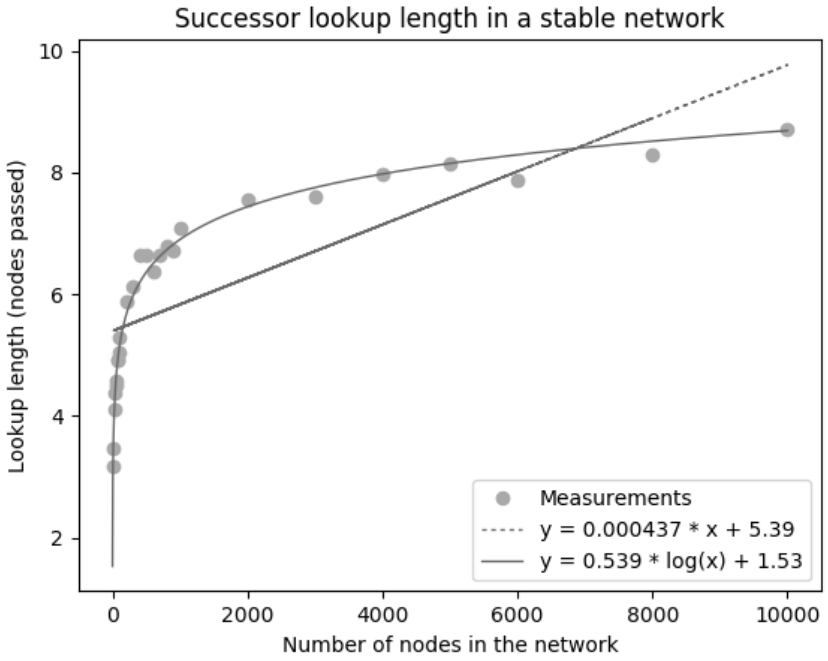

Figure 8: The average measurements and the linear and logarithmic models for the lookup in a stable network.

The measurements taken for each network size, that were used to calculate the average lookup length, are visualized in the boxplot in the figure 9. The average, median, minimum and maximum lookup lengths for all used network sizes are listed in the table 8 to help interpret the figure 9.

The average lookup length data was used to generate the linear and logarithmic regression models listed in the table 7. The quality of the fit is assessed using the included mean squared error (MSE) and the coefficient of determination (R^2) values. The models are also visualized in the figure 8.

The logarithmic regression model has a smaller MSE (*0.0374*) than the linear regression model (*1.02*). The logaritmic model also has a better R^2 value of *0.984* compared to the *0.571* of the linear model. Based on this, the

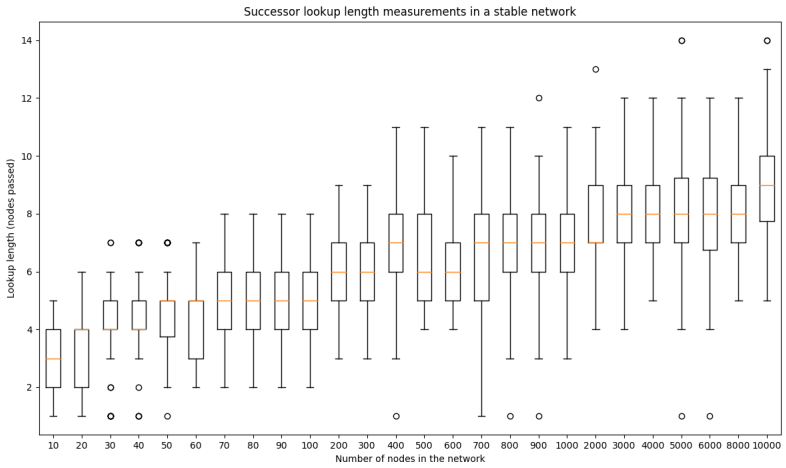

Figure 9: The lookup length measurements with various network sizes in a stable network.

Model	MSE	R^2
y = 0.000437 * x + 5.39	1.02	0.571
y = 0.539 * log(x) + 1.53	0.0374	0.984

Table 7: The models and the quality of the fit for the lookup in a stable network.

logarithmic regression model explains the relationship between the lookup length and the network size better, than the linear regression model.

The third statement about the performance of Chord in section 4 stated that: "*Finding a successor of an arbitrary key in a N-node network requires the query to be routed through* $O(logN)$ *nodes.*" Based on the analysis above, this assumption is true.

Nodes in network	Number of executed lookups	Avg hops	Median hops	Min hops	Max hops
10	100	3	3	1	5
20	100	3	4	1	6
30	100	4	4	1	7
40	100	4	4	1	7
50	100	5	5	1	7
60	100	5	5	2	7
70	100	5	5	2	8
80	100	5	5	2	8
90	100	5	5	2	8
100	100	5	5	2	8
200	100	6	6	3	9
300	100	6	6	3	9
400	100	7	7	1	11
500	100	7	6	4	11
600	100	6	6	4	10
700	100	7	7	1	11
800	100	7	7	1	11
900	100	7	7	1	12
1000	100	7	7	3	11
2000	100	8	7	4	13
3000	100	8	8	4	12
4000	100	8	8	5	12
5000	100	8	8	1	14
6000	100	8	8	1	12
8000	100	8	8	5	12
10000	100	9	9	5	14

Table 8: The measurement details with average, min and max lookup lengths in hops for the lookup in a stable network.

6.4 Messages exchanged to update the routing invariants when a node joins the network

The number of messages being exchanged to update the routing invariants of a node that joins a stable Chord network was measured according to the plan. The gathered measurement data was used to calculate the average exchanged messages in relation to the size of the Chord network. The average exchanged messages in relation to the network size is visualized in the figure 10.

The use of average transferred messages provides a good estimate, because the individual measurements inside the different network configurations can vary greatly. This is visualized in 11 that shows the number of exchanged messages for the first measurement in each network configuration.

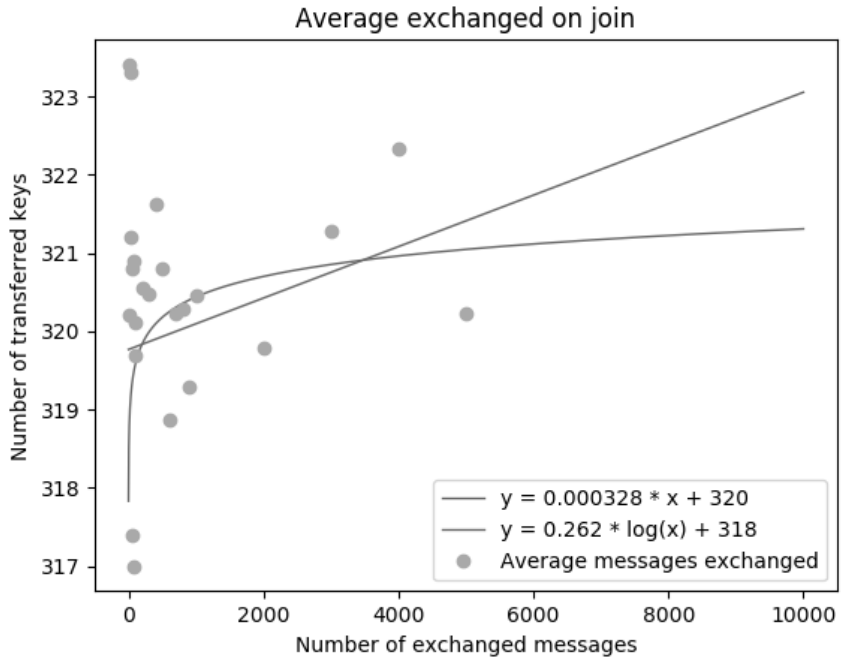

Figure 10: The average number of messages exchanged to update the routing invariants when a new node joins in relation to various network sizes.

The measurements taken for each network size are visualized in the box plot in the figure 12. The average, median, minimum and maximum number of exchanged message for all used network sizes are listed in the table 9 to help interpret the figure 12.

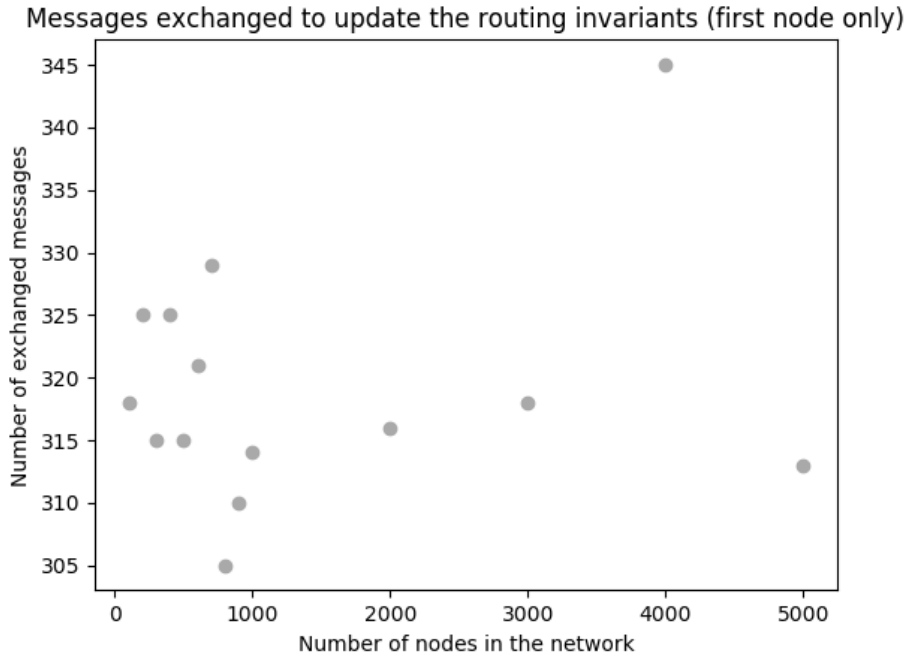

Figure 11: The number of messages exchanged to update the routing invariants when the first node joins a network of certain size.

The average exchanged messages data was used to generate the linear and logarithmic regression models listed in the table 10. The quality of the fit is assessed using the included mean squared error (MSE) and the coefficient of determination (R^2) values. The models are also visualized in the figure 10.

The number exchanged messages fluctuated greatly when the network size is small. In an attempt to increase the probability of isolating a pattern from the noise, only the results from networks with 50 or more nodes where used to generate the regression models.

The resulting logarithmic regression model has a smaller MSE (*1.29*) than the linear regression model (*1.38*). The logaritmic model also has a better R^2 value of *0.184* compared to the *0.133* of the linear model. Even though the logarithmic regression model explains the relationship between the exchanged messages and the network size better than the linear regression model, it does not do so convincingly. The absolute difference of MSE is *0.09* which is equals to increase of 6.98% from 1.29 to 1.38. Additionally, both R^2 values are quite low and do not indicate a good fit. One affecting issue is that the different average values are close to each other with an absolute difference of

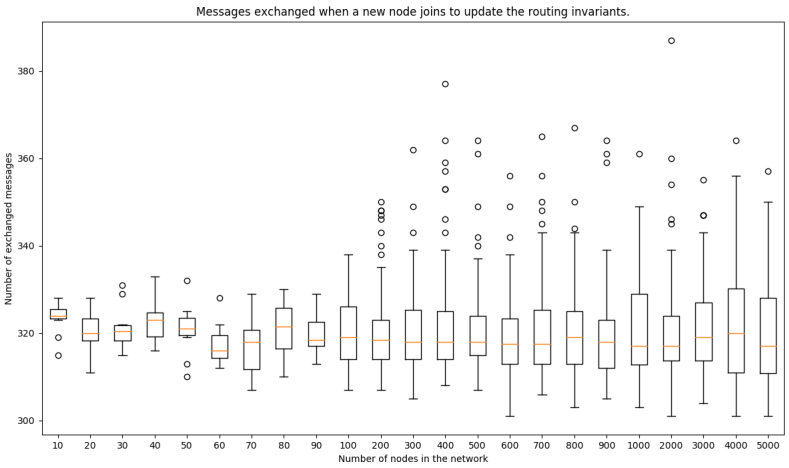

Figure 12: The number of messages exchanged when a node joins to update the routing invariants in relation to various network sizes.

8 between the minimum average of 316 and the maximum average of 324.

The fourth statement about the performance of Chord in section 4 states that: "*When a node joins the network,* $O((logN)^2)$ *messages are needed to update the routing invariants of the affected nodes.*" Based on the analysis above, this assumption cannot be verified or questioned.

Executing more test runs and including larger networks could be used in attempt to find a clearer pattern and lower the noise.

Network size (nodes)	Avg	Median	Min	Max
(10, 110]	323.4	324	315	328
(20, 120]	320.2	320	311	328
(30, 130]	321.2	320	315	331
(40, 140]	323.3	323	316	333
(50, 150]	320.8	321	310	332
(60, 160]	317.4	316	312	328
(70, 170]	317.0	318	307	329
(80, 180]	320.9	321	310	330
(90, 190]	319.7	318	313	329
(100, 200]	320.12	319	307	338
(200, 300]	320.56	318	307	350
(300, 400]	320.47	318	305	362
(400, 500]	321.62	318	308	377
(500, 600]	320.8	318	307	364
(600, 700]	318.87	317	301	356
(700, 800]	320.22	317	306	365
(800, 900]	320.28	319	303	367
(900, 1000]	319.29	318	305	364
(1000, 1100]	320.45	317	303	361
(2000, 2100]	319.78	317	301	387
(3000, 3100]	321.27	319	304	355
(4000, 4100]	322.33	320	301	364
(5000, 5100]	320.23	317	301	357

Table 9: Messages needed to update the routing invariants of a node when it joins.

Model	MSE	R^2
y = 0.000328 * x + 320	1.38	0.133
y = 0.262 * log(x) + 318	1.29	0.184

Table 10: The expected model for the data moved data items.

6.5 Finding a successor in a network where half of the nodes are missing routing invariants

The lookup length measurements in a Chord network where half of the Chord nodes were missing routing invariants were executed as planned. The gathered measurement data was used to calculate the average lookup length for each Chord network size. The average lookup length in relation to the network size is visualized in the figure 13.

The measurements taken for each network size, that were used to calculate the average lookup length, are visualized in the box plot in figure 14. The average, median, minimum and maximum lookup lengths for all used network sizes are also listed in the table 12 to help interpret the figure 14.

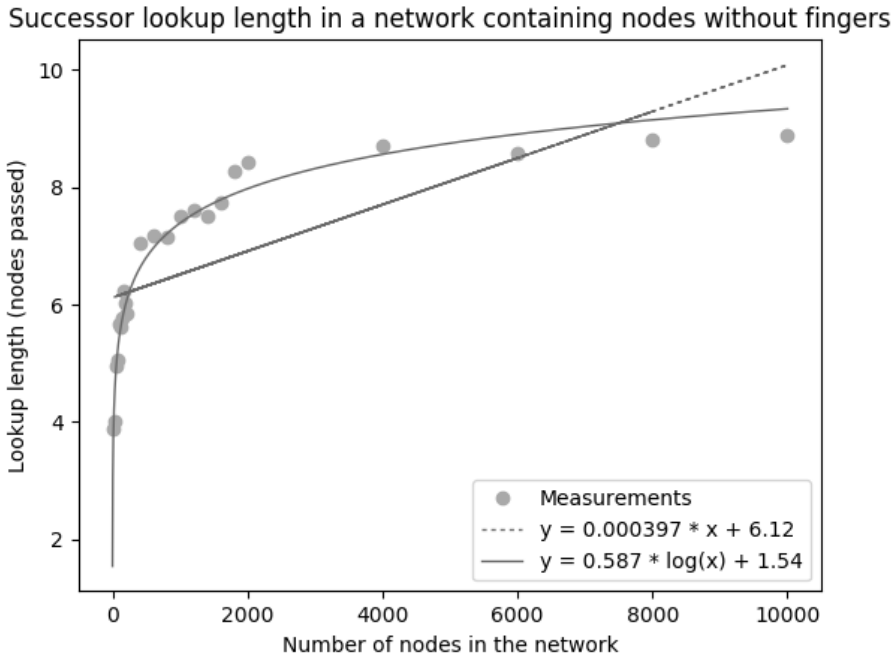

Figure 13: The average measurements and the linear and logarithmic models for the lookup in a network were half of the nodes are missing finger tables.

The average lookup length data was used to generate the linear and logarithmic regression models listed in the table 11. The quality of the fit can be assessed using the included mean squared error (MSE) and the coefficient of determination (R^2) values. The models are also visualized in the figure 13.

The logarithmic regression model has a smaller MSE (*0.082*) than the linear

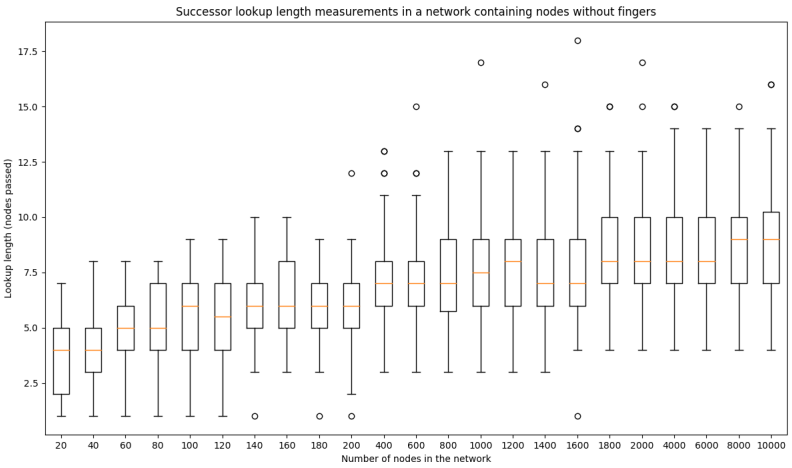

Figure 14: The lookup length measurements with various network sizes in a network were half of the nodes are missing finger tables.

regression model (*1.11*). The logaritmic model also has a better R^2 value of *0.963* compared to the *0.5* of the linear model. Based on this, the logarithmic regression model explains the relationship between the lookup length and the network size better, than the linear regression model.

The fifth statement about the performance of Chord in section 4 stated that: "*When N new nodes, each without finger pointers, join an existing stable network of N nodes, the successor lookup of a key will still take* $O(logN)$ *messages.*" Based on the analysis above, this assumption is true.

Model	MSE	R^2
y = 0.000397 * x + 6.12	1.11	0.5
y = 0.587 * log(x) + 1.54	0.082	0.963

Table 11: The models and the quality of the fit for the lookup in a network where half of the nodes are missing finger tables.

Nodes in network	Number of lookups	Avg	Median	Min	Max
20	100	4	4	1	7
40	100	4	4	1	8
60	100	5	5	1	8
80	100	5	5	1	8
100	100	6	6	1	9
120	100	6	6	1	9
140	100	6	6	1	10
160	100	6	6	3	10
180	100	6	6	1	9
200	100	6	6	1	12
400	100	7	7	3	13
600	100	7	7	3	15
800	100	7	7	3	13
1000	100	8	8	3	17
1200	100	8	8	3	13
1400	100	8	7	3	16
1600	100	8	7	1	18
1800	100	8	8	4	15
2000	100	8	8	4	17
4000	100	9	8	4	15
6000	100	9	8	4	14
8000	100	9	9	4	15
10000	100	9	9	4	16

Table 12: The measurement details with average, min and max lookup lengths for the lookup in a network were half of the nodes are missing finger tables.

6.6 Finding a successor in a network where half of the nodes have been shutdown

The lookup length measurements in a partially shutdown Chord network were executed as planned. The gathered measurement data was used to calculate the average lookup length for each Chord network size. The average lookup length in relation to the network size is visualized in the figure 15.

The measurements taken for each network size, that were used to calculate the average lookup length, are visualized in the boxplot in the figure 16. The average, median, minimum and maximum lookup lengths for all used network sizes are listed in the table 14 to help interpret the figure 16.

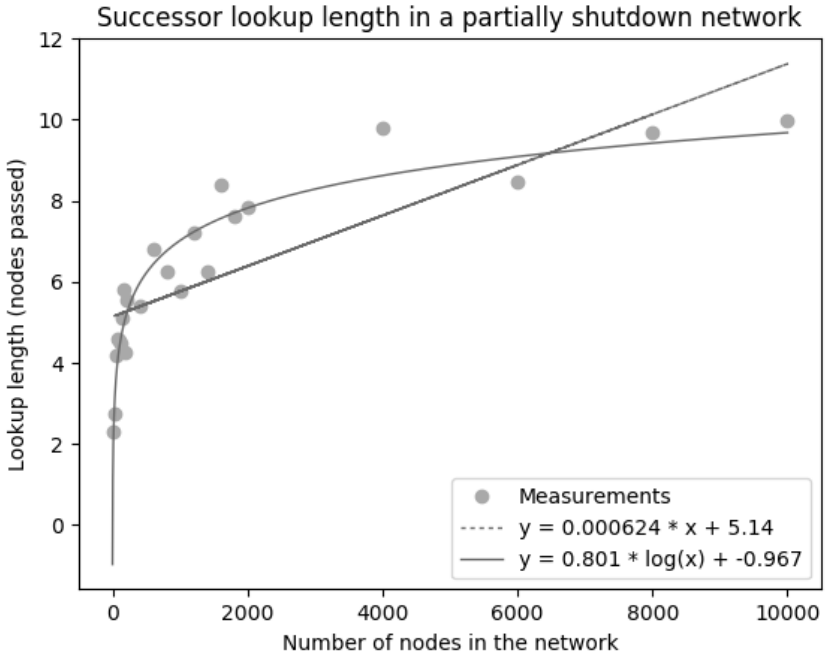

Figure 15: The average measurements and the linear and logarithmic models for the lookup in a network were half of the nodes have been shutdown.

Based on the table 14, on average 34% of the lookups were successful when taken account all 23 test runs with different network sizes and with the network shutdown rate of 50%. The table 15 shows how the number of failed lookups increases when the portion of the network that has been shutdown increases. The original size of the network is 10,000 Chord nodes.

To measure the failure rate on each stage, five runs of 100 lookup queries

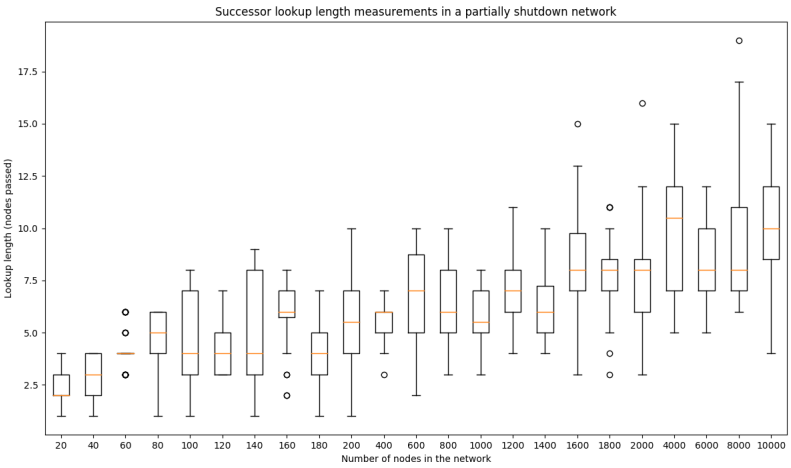

Figure 16: The lookup length measurements with various network sizes in a network were half of the nodes have been shutdown.

were executed and failures counted.

A lookup is considered to have failed, when the lookup query cannot be routed to a node that is responsible of the key being used in the lookup. This happens if the target node has been shutdown or there is no path to the target node due to a large portion of the preceding nodes having been shutdown.

As seen in the table 15, the failure rate exceeds 50% when 38% of the original network was shutdown. Note that this test was only executed one time for each network configuration and size. Therefore the results cannot be considered to be definite, but it does indicate that routing to nodes will start failing even if a small part of the network has been shutdown.

The average lookup length data was used to generate the linear and logarithmic regression models listed in the table 13. The quality of the fit is assessed using the included mean squared error (MSE) and the coefficient of determination (R^2) values. The models are also visualized in the figure 15.

The logarithmic regression model has a smaller MSE (*0.381*) than the linear regression model (*1.63*). The logaritmic model also has a better R^2 value of *0.913* compared to the *0.628* of the linear model. Based on this, the logarithmic regression model explains the relationship between the lookup length and the network size better than the linear regression model.

Whereas the logarithmic nature of the lookup cost is clear, it can be ques-

tioned whether the results hold *with high probability* due to the high failure rate. The results show that the rate of failing to route a request goes down when the size of the network increases. Additionally, the *successor list* allowing nodes to track several consecutive successors and described in [SMK+01] was not implemented. It is assumed that the use of the successor list would have considerably decreased the failure rate. Based on this, it is assumed that the results will hold with probability at least $1 - 1/n^c$, where n is the number of nodes for any $c \geq 1$ and that the probability will close to *1* with a high n.

The sixth statement about the performance of Chord in section 4 stated that: "*When half of nodes of a stable N node network fail and are disconnected from the network, the successor lookup of a key will still take $O(logN)$ messages.*" Based on the analysis above, this assumption is true.

Model	MSE	R^2
y = 0.000624 * x + 5.14	1.63	0.628
y = 0.801 * log(x) + -0.967	0.381	0.913

Table 13: The models and the quality of the fit for the lookup in a network with half of the nodes having been shutdown..

Nodes in network	Total lookups (N)	Successful lookups (N)	Avg	Median	Min	Max
20	100	48	2	2	1	4
40	100	58	3	3	1	4
60	100	41	4	4	3	6
80	100	43	5	5	1	6
100	100	35	5	4	1	8
120	100	43	4	4	3	7
140	100	33	5	4	1	9
160	100	28	6	6	2	8
180	100	40	4	4	1	7
200	100	44	6	6	1	10
400	100	18	5	6	3	7
600	100	38	7	7	2	10
800	100	41	6	6	3	10
1000	100	22	6	6	3	8
1200	100	28	7	7	4	11
1400	100	16	6	6	4	10
1600	100	30	8	8	3	15
1800	100	31	8	8	3	11
2000	100	20	8	8	3	16
4000	100	32	10	10	5	15
6000	100	24	8	8	5	12
8000	100	33	10	8	6	19
10000	100	31	10	10	4	15

Table 14: The measurement details with number of successful lookups and average, min and max lookup lengths for the lookup in a network where half of the nodes have been shutdown.

Running nodes (N)	Network up (%)	Fails on run 1	Fails on run 2	Fails on run 3	Fails on run 4	Fails on run 5	Total fails	Fail (%)
9570	96	1	4	5	1	4	15	3
9500	95	10	8	6	9	7	40	8
9250	92	12	12	13	10	10	57	11
9000	90	21	12	11	8	18	70	14
8750	88	15	14	20	22	19	90	18
8500	85	24	12	23	22	14	95	19
8250	82	17	22	26	21	28	114	23
8000	80	21	31	37	22	21	132	26
7750	78	30	31	26	32	24	143	29
7500	75	30	26	30	30	30	146	29
7250	72	39	37	32	28	32	168	34
7000	70	38	40	40	43	30	191	38
6750	68	47	46	39	47	37	216	43
6500	65	50	35	47	45	37	214	43
6400	64	38	49	49	47	49	232	46
6300	63	42	51	41	48	51	233	47
6200	62	52	49	56	54	54	265	53
6100	61	56	39	52	40	43	230	46
6000	60	48	60	59	48	60	275	55
5900	59	60	54	62	52	57	285	57
5800	58	50	61	59	63	54	287	57
5700	57	52	53	56	54	60	275	55
5600	56	62	60	63	59	56	300	60
5500	55	57	57	59	65	63	301	60
5400	54	61	61	66	62	59	309	62
5300	53	60	62	62	69	66	319	64
5200	52	61	66	66	58	63	314	63
5100	51	56	65	64	70	59	314	63
5000	50	65	71	75	68	67	346	69

Table 15: The failure rate of lookups in a partially shutdown network. Five measurements taken with different parts of the network having been shutdown.

7 Conclusions

After analyzing the results, the following conclusions can be drawn. Let K be the total number keys stored in the network that consists of total N number of Chord nodes. Each node in a Chord network is responsible for K/N keys and when the $(N+1)^{st}$ node joins the network, the joining node receives $O(K/N)$ keys from its successor. Finding a node responsible of a key takes $O(logN)$ hops over the network. The lookup requests take $O(logN)$ number of hops, even when half of the nodes in the network are either without valid finger pointers or have failed.

The chosen test approach of generating linear and logarithmic regression models and comparing them using the mean squared error (MSE) and the coefficient of determination (R^2) was strong enough to draw clear conclusions about the nature of the performance statements stated above.

No conclusions could be drawn about the performance assumption, that $O((logN)^2)$ messages are needed to update the routing invariants of the affected nodes when a new node joins the network. The generated model could not be used to either verify or question this point. It is assumed, that using bigger network size beyond the 5000 Chord nodes that was now used, could result in obtaining measurements suitable for building a more accurate model.

In all tests, a singular and predefined Chord node was used to execute the tests. It might have been prudent to use several different, randomly chosen nodes as entry points to the Chord network. This would have required more test runs per test, but would have negated, or lessened, any bias that came from using a singular node as a starting point. It can be that the local network topology around that one node skewed the results in a significant manner.

Additionally, it would have been prudent to implement the successor list explained in [SMK+01] for the testing of finding a successor in a network where up to half of the nodes have failed. It is assumed, that using the successor list would have provided better reliability to the results.

Setting up the test networks in the *Ukko* cluster was time consuming. On average it took from one to three minutes to have a new node join the network when the network size was big enough. The possibility to have nodes joining concurrently would have helped and opened the possibility to consider using bigger network sizes than the 10000 now used. This would have required to implement the stabilization routing described in [SMK+01].

References

[Ana17] Analytics, Continuum: *Anaconda overview*, 2017. https://www.continuum.io/anaconda-overview.

[Bra14] Bray, T.: *The javascript object notation (json) data interchange format.* Rfc 7159, RFC Editor, March 2014. http://www.rfc-editor.org/rfc/rfc7159.txt, http://www.rfc-editor.org/rfc/rfc7159.txt.

[Bur12] Burroughs, Edgar Rice: *A princess of mars*, 1912. http://www.gutenberg.org/ebooks/62.

[CSRL01] Cormen, Thomas H., Stein, Clifford, Rivest, Ronald L., and Leiserson, Charles E.: *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001, ISBN 0070131511.

[dev17] developers, NumPy: *Numpy*, 2017. http://www.numpy.org/.

[dt17] team, The Matplotlib development: *Matplotlib: Python plotting*, 2017. https://matplotlib.org/.

[EKB+13] Eckenfels, B, Kleen, A, Blundell, P, Cox, A, and Kempen, F van: *Manpage of ifconfig*, 2013. http://net-tools.sourceforge.net/man/ifconfig.8.html.

[EMH+13] Eckenfels, B, Micek, M, Hoang, T, Cox, A, and Welsh, M: *Manpage of netstat*, 2013. http://net-tools.sourceforge.net/man/netstat.8.html.

[Fou] Foundation, Free Software: *Screen user's manual.* https://www.gnu.org/software/screen/manual/screen.html.

[Fou17] Foundation, Python Software: *Python 3.6.0 documentation*, 2017. https://docs.python.org/3/.

[Git13] Git: *Manpage of netstat*, 2013. https://git-scm.com/.

[Git17] GitHub, Inc.: *Github*, 2017. https://github.com/.

[Gro15] Group, Object Management: *Omg unified modeling language(omg uml)*, 2015. http://www.omg.org/spec/UML/2.5.

[Gut71] Gutenberg, Project: *Free ebook by project gutenberg*, 1971. http://www.gutenberg.org/.

[hll17] learn https://github.com/scikit-learn/scikit: *Scikit-learn: machine learning in python*, 2017. http://scikit-learn.org/stable/index.html.

[(IE81] (IETF), Internet Engineering Task Force: *Internet protocol*, 1981. https://tools.ietf.org/rfc/rfc791.txt.

[Jac92] Jacobson, I.: *Object-oriented software engineering: a use case driven approach*. ACM Press Series. ACM Press, 1992, ISBN 9780201544350. https://books.google.fi/books?id=A6lQAAAAMAAJ.

[JWHT14] James, G., Witten, D., Hastie, T., and Tibshirani, R.: *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics. Springer New York, 2014, ISBN 9781461471370. https://books.google.fi/books?id=at1bmAEACAAJ.

[KLL+97] Karger, David, Lehman, Eric, Leighton, Tom, Panigrahy, Rina, Levine, Matthew, and Lewin, Daniel: *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web*. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM, ISBN 0-89791-888-6. http://doi.acm.org/10.1145/258533.258660.

[Mar03] Martin, R.C.: *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003, ISBN 9780135974445. https://books.google.fi/books?id=0HYhAQAAIAAJ.

[Mes07] Meszaros, G.: *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2007, ISBN 9780132797467. https://books.google.co.uk/books?id=-izOiCEIABQC.

[oST15] Standards, National Institute of and Technology: *Secure standard hash (shs)*, 2015. http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[Pos81a] Postel, Jon: *Internet protocol*. Std 5, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc791.txt, http://www.rfc-editor.org/rfc/rfc791.txt.

[Pos81b] Postel, Jon: *Transmission control protocol*. Std 7, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt, http://www.rfc-editor.org/rfc/rfc793.txt.

[RS04] Rogaway, Phillip and Shrimpton, Thomas: *Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance*, pages 371–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, ISBN 978-3-540-25937-4. https://doi.org/10.1007/978-3-540-25937-4_24.

[Sch07] Schneier, B.: *Applied cryptography: protocols, algorithms, and source code in C*. Wiley-India, 2007, ISBN 9788126513680. https://books.google.fi/books?id=A6ZO2D6ayNwC.

[Sha05] Shafranovich, Y.: *Common format and mime type for comma-separated values (csv) files*. Rfc 4180, RFC Editor, October 2005. http://www.rfc-editor.org/rfc/rfc4180.txt, http://www.rfc-editor.org/rfc/rfc4180.txt.

[SMK+01] Stoica, Ion, Morris, Robert, Karger, David, Kaashoek, M. Frans, and Balakrishnan, Hari: *Chord: A scalable peer-to-peer lookup service for internet applications*. SIGCOMM Comput. Commun. Rev., 31(4):149–160, August 2001, ISSN 0146-4833. http://doi.acm.org/10.1145/964723.383071.

[SMLN+03] Stoica, Ion, Morris, Robert, Liben-Nowell, David, Karger, David R., Kaashoek, M. Frans, Dabek, Frank, and Balakrishnan, Hari: *Chord: A scalable peer-to-peer lookup protocol for internet applications*. IEEE/ACM Trans. Netw., 11(1):17–32, February 2003, ISSN 1063-6692. http://dx.doi.org/10.1109/TNET.2002.808407.

[s.r17] s.r.o., JetBrains: *Pycharm: Python ide for professional developers by jetbrains*, 2017. https://www.jetbrains.com/pycharm/.

[Tar10] Tarkoma, Sasu: *Overlay Networks: Toward Information Networking*. Auerbach Publications, Boston, MA, USA, 1st edition, 2010, ISBN 143981371X, 9781439813713.

[Top17] Topinka, T.: *Statistic :: Jetbrains plugin repository*, 2017. https://plugins.jetbrains.com/plugin/4509-statistic.

[UoH15] Helsinki, Department of Computer Science University of: *High performance cluster ukko*, 2015. https://www.cs.helsinki.fi/en/compfac/high-performance-cluster-ukko.

[Woo15] Wood, Simon N.: *Core Statistics*. Institute of Mathematical Statistics Textbooks. Cambridge University Press, 2015.

[WS02] Wrede, R.C. and Spiegel, M.R.: *Schaums Outline of Advanced Calculus, Second Edition*. Schaums Outline Series. Mcgraw-hill, 2002, ISBN 9780071375672. https://books.google.fi/books?id=Ud9Iuq2HyvgC.

[WYY05] Wang, Xiaoyun, Yin, Yiqun Lisa, and Yu, Hongbo: *Finding Collisions in the Full SHA-1*, pages 17–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, ISBN 978-3-540-31870-5. https://doi.org/10.1007/11535218_2.

[You08] Young, M.: *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley India Pvt. Limited, 2008, ISBN 9788126517732. https://books.google.fi/books?id=8MT5hzTvwnwC.

[Zü17] Zürich, ETH: *Principles of distributed computing (lecture collection), chapter 7, maximal independent set*, 2017. https://disco.ethz.ch/courses/podc_allstars/lecture/chapter7.pdf.