Flexible Routing Tables in a Distributed Key-Value Store

Diplomarbeit

zur Erlangung des akademischen Grades Diplominformatiker

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: Magnus Müller
geboren am: 29. März 1987
in: Schwäbisch Hall

Gutachter: Prof. Dr. Alexander Reinefeld
Prof. Dr. Björn Scheuermann

eingereicht am: ...... verteidigt am: ......
Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, July 3, 2013
Abstract

We present our implementation of Flexible Routing Tables (FRT) in the distributed key-value store Scalaris. Classic routing table generators like Chord structure routing tables by considering node identifiers only. Secondary requirements such as latency properties are considered after the general structure, eliminating potential performance gains. FRT aims to allow multiple requirements when creating routing tables. The concept is based on lazy routing table entries learning and filtering entries when a routing table is full. Entries are filtered such that the resulting table is best among all other possible tables where one node was filtered. FRT uses a total order to define which of the possible routing tables is best. We implemented two instances of FRT, namely FRT-Chord and grouped FRT-Chord (GFRT-Chord). We compare our implementation with the existing implementation of Chord in Scalaris and evaluate the advantages and disadvantages of more flexible routing schemes.
## Contents

**Introduction** ix

1. Background 1
   1.1. Overlay Networks .................................................. 1
   1.2. Distributed Hash Tables ........................................... 3
   1.3. Consistent Hashing and Chord ...................................... 6
       1.3.1. Ranged Hash Functions and Consistent Hashing ............. 7
       1.3.2. Chord ......................................................... 8
       1.3.3. Routing Inconsistencies ..................................... 15
       1.3.4. Enhancing Chord ............................................. 17

2. Flexible Routing Tables 19
   2.1. Motivation ............................................................ 19
   2.2. The Concept of Flexible Routing Tables .......................... 20
       2.2.1. FRT Primitives ............................................... 21
       2.2.2. Expected Advantages ......................................... 22
       2.2.3. Examples of FRT .............................................. 23
   2.3. FRT-Chord ........................................................... 23
   2.4. GFRT-Chord ......................................................... 32
   2.5. Summary ............................................................. 35

3. Scalaris 37
   3.1. Scalaris Peer-to-Peer Layer Software Architecture ............. 38
       3.1.1. Message Hub: dht_node ....................................... 39
       3.1.2. Accessing the Peer-to-Peer Layer ............................. 40
       3.1.3. Scalaris Ring Maintenance ................................... 40
       3.1.4. Routing Table Construction .................................. 41
       3.1.5. The Routing Table Behavior .................................. 41
       3.1.6. Key Lookups in Scalaris ..................................... 43
Contents

4. Implementation of Flexible Routing Tables 45
   4.1. Routing Table Structure ................................................. 46
   4.2. Generalized Entry Learning and Filtering .............................. 49
      4.2.1. Communication between Routing Tables ............................ 49
      4.2.2. Initialization and Entry Learning .................................. 50
      4.2.3. Entry Filtering ...................................................... 59
   4.3. FRT-Chord and GFRT-Chord Implementation ............................. 59
      4.3.1. FRT-Chord .............................................................. 60
      4.3.2. GFRT-Chord .............................................................. 60

5. Evaluation 63
   5.1. Finger Placement ........................................................... 64
      5.1.1. Structure of the Experiment ......................................... 64
      5.1.2. Experiments on Densely-Filled Rings ............................... 66
      5.1.3. Experiments on Sparsely-Filled Rings ............................. 68
      5.1.4. Finger Placement Histograms ........................................ 69
      5.1.5. Conclusion .............................................................. 71
   5.2. Convergence Speed .......................................................... 72
      5.2.1. Convergence Speed for Fixed Active Learning Lookups Interval .... 73
      5.2.2. Convergence Speed for Different Active Learning Lookup Intervals .... 75
   5.3. PlanetLab: Average Hops and Latency Experiments ...................... 79
      5.3.1. Structure of the experiment ......................................... 79
      5.3.2. Expectations ............................................................ 82
      5.3.3. Results ................................................................. 83
      5.3.4. Conclusion .............................................................. 85
   5.4. Scalaris under Churn ....................................................... 85
      5.4.1. Structure of the Simulation ......................................... 86
      5.4.2. Expectations ............................................................ 89
      5.4.3. Results ................................................................. 89
      5.5. Maintenance Overhead .................................................... 90

6. Future Work 95
   6.1. Improving FRT-Chord and GFRT-Chord .................................. 95
   6.2. Improving our Evaluation .................................................. 96

7. Conclusion 97
Introduction

Over the last forty years, distributed systems have always been an important field of research in computer science. In the last decade we have seen the rise of multiple companies whose success is based solely on offering internet-based services to millions of customers worldwide. In contrast, companies with more old-fashioned business models vanished as they did not adapt fast enough to the change in demanded services.

The Internet has been one of the driving powers behind the development of distributed systems. Its growth forces distributed systems development to strive for more scalability. The Internet’s dynamic nature, with a huge number of participating computers joining and leaving the network at each instant in time, motivates research on availability, consistency and resilience.

The demand for distributed services still increases and is far from being satisfied. Distributed databases, distributed messaging systems, distributed storage and many other applications still offer an abundance of open questions to be answered in the future. This includes how to cope with heterogeneity, how to maintain resilience or in which way the trade-off consistency versus availability should be understood and handled [Aba12].

In this thesis, we consider an important class of distributed systems: distributed hash tables (DHTs). DHTs provide services to find the computer associated with a key. This is used, among other applications, to build databases which store key-value pairs (key-value stores). The user interface of such databases is often kept simple and allows searching, storing and deleting a data item associated with a data key. DHTs provide the underlying lookup functionality of such databases. Given a key $k$, a DHT provides a function $\text{lookup}(k)$ to retrieve the computer which is responsible for $k$. The term “responsibility” is used differently depending on the application of the DHT. In a key-value store, it means that the computer is responsible for any item stored under the given key $k$. In other systems, such as a multicast messaging system, the computer responsible for key $k$ might be responsible for forwarding requests to a central server.

The term DHT can at least be traced back to the parallel programming language Linda [CGL86] from 1986. File sharing applications increased the popularity of the concept rapidly in the early
Introduction

Figure 1.: The structure of a physical network. Not all nodes are connected directly with each other, but in this example each node can reach each other node via intermediate nodes. Thus, node a could send a message to node b. Such a message must be routed either through c and d or through an alternative route.

2000s. Examples of such applications are Napster, Gnutella, and, more recently, BitTorrent. Research in DHTs focused on investigating how such peer-to-peer systems can be structured in order to reduce bottlenecks and avoid single point of failures, while providing maintenance and request management at low costs.

Basic Terminology and Syntax In this thesis, we consider the task of distributing data over multiple machines (nodes) connected by a network. Not all nodes are necessarily directly connected with each other, hence the network consists of both a set of nodes and a set of network links between the nodes. Nodes which are connected via a network link are called neighbors. We will call the graph which represents the arrangement of nodes and their connecting links as the network’s topology, or simply as topology.

A node stores information on its known neighbors in its routing table. When non-directly connected nodes want to communicate, intermediate nodes have to forward the messages along the network links to their neighbors until the message reaches its intended recipient. We call this forwarding of messages routing. A request routed from node a to node b follows a path through the network along the intermediate nodes. The number of links used in this path is the number of hops needed to cross the network from a to b. The latency of a request is the time it takes to route the request from a to b. This is the length of the interval between the request leaving node a and entering node b. Both latency and the number of network hops are important metrics to evaluate how well a distributed algorithm is performing. The diameter of a topology is the maximum number of hops needed to route a message between any pair of nodes. So in a fully meshed network, where each node is connected with each other node, the diameter is 1.

This thesis falls into the general subject of peer-to-peer systems. A peer-to-peer system is a distributed system consisting of nodes which execute the same algorithms and no node has
a preassigned role in the network. Thus, no central authority should exist. The grade of this uniformity of roles differs between different systems. In a fully-decentralized system, all nodes execute the same algorithms and perform the same tasks. This requirement might be relaxed, e.g. for bootstrapping the system. We will not distinguish the different levels of uniformity, but follow the paradigm of avoiding preassigned roles wherever possible.

This thesis will display multiple distributed algorithms. In this paragraph, we will introduce the pseudo-code used for the algorithms presented in Chapter 1. Algorithms in later chapters are written in the programming language Erlang, which we will not introduce here.

Each node is considered to be a state machine which can send and accept messages. Received messages are handled by message handlers. Algorithm 1 displays an example for a message handler. As can be seen in the example, the name of a message handler is node.m(p), where node is the node handling the received message m and the list p contains additional information piggybacked onto the message. A node contains local state, e.g. its routing table. This state is queried using the notation n.local_information and assigned to with n.local_information←value.

Algorithm 1 Syntax of Algorithms

```plaintext
message c.message(p₁, p₂,..., pₘ) ▷ Name of the handled message and parameters
  if c.pred then ▷ Query node’s local information
    return local key ▷ Sending an answer to the client which issued the call
  end if
  c.pred.message() ▷ Send an empty message with one parameter to the node c.pred
  c.pred.message(p) ▷ Send a message with one parameter to the node c.pred
end message
```

Aims of this Thesis

This thesis aims to investigate if lazy topology generation is better or worse than static topology generation. In lazy topology generation, nodes learn about other nodes “by accident”, e.g. when forwarding lookup requests. Encountered nodes can then be added to a node’s routing table. In static topology generation, nodes actively try to maintain the topology. So, nodes periodically execute maintenance algorithms to find nodes which should be added to their routing tables.

Our goal is to improve lookup routing in Scalaris [SSR08], a distributed key-value store. We want to improve both the latency of lookups as well as the reliability. The current algorithms...
Introduction

in Scalaris generate the topologies statically. Our lazy topology generation approach is based on the work by Nagao and Shudo [NS11]. The authors proposed a generalized topology generation framework called flexible routing tables (FRT) consisting of three primitives. The first primitive, Guarantee of Reachability, maintains the topology such that the topology is not partitioned. The second primitive, Entry Learning, allows a peer to find out about other peers in the system. The third primitive, Entry Filtering, is used by peers to unlearn known peers. We will implement two instances of the generalized approach, called FRT-Chord and GFRT-Chord. Nagao and Shudo defined FRT-Chord such that the resulting topology is similar to the one created by the topology generating algorithm Chord [SMLN+03], which is also used in Scalaris. To show that FRT-Chord is extensible, the authors proposed GFRT-Chord which adds peer groups to further speed up lookups. Our goal is to show that the proposed algorithms are as flexible as described in [NS11] and that they outperform the Chord implementation in Scalaris.

Expected Results

We expect our implementation of FRT-Chord and GFRT-Chord to outperform existing routing schemes in Scalaris in terms of lookup path lengths. As GFRT-Chord can take the underlying network topology into account for overlay construction, we expect GFRT-Chord to perform better than both FRT-Chord and Chord in terms of latency. Additionally, FRT-Chord and GFRT-Chord should not exhibit a larger maintenance overhead than the current routing scheme. Our overall expectation is that both FRT-Chord and GFRT-Chord outperform Chord latency-wise and that both schemes do not increase the cost of running Scalaris in a real world setting. Additionally, FRT-Chord and GFRT-Chord are extensible as the implementation of the underlying FRT primitives can be adapted as needed.

Outline

This thesis is based on a large field of research. Consequently, we will begin by introducing the main terms and background knowledge needed for this thesis in Chapter 1. We will proceed by presenting the concept of flexible routing tables and two instances, FRT-Chord and GFRT-Chord, in Chapter 2. This is followed by an introduction to Scalaris and its peer-to-peer layer software architecture. With the knowledge on Scalaris’ peer-to-peer layer, we are able to introduce our implementation of FRT-Chord and GFRT-Chord in Chapter 4. This discussion
is followed by our evaluation in Chapter 5. We conclude the thesis with an outlook on future work and a conclusion in Chapters 6 and 7.

**Contributions**

The main contribution of this work is a fully working implementation of FRT-Chord and GFRT-Chord in Scalaris. Our implementation is extensible, as will be shown in Section 4. We devised an optimization to speed up topology maintenance for both FRT-Chord and GFRT-Chord. Our thorough evaluation in Chapter 5 compares the difference between Scalaris’ Chord implementation with our implementations of FRT-Chord and GFRT-Chord by using both simulations and experiments on PlanetLab, giving a good overview of the benefits of FRT-Chord and GFRT-Chord over Chord.
1. Background

This chapter will introduce the knowledge necessary for subsequent chapters. We start by introducing the concepts overlay networks and distributed hash tables (DHTs). The concepts behind mainline DHTs today descend from one of two ideas: consistent hashing [KLL+97] and the Plaxton Mesh [PRR97]. We will explain consistent hashing in order to understand the following section on a uniform distributed hash table called Chord. This will give us enough knowledge to grasp the basics of a flexible routing tables (FRT) and its implementations FRT-Chord and GFRT-Chord. This thesis contributes an implementation of FRT [NS11] to Scalaris.

1.1. Overlay Networks

Physical networks tend to exhibit an increasing number of problems when they grow. The networks’ topology cannot be observed reliably anymore. This can be due to nodes joining and leaving the system (churn), failing connections between nodes and because of timeouts from link congestion. Hence, it is often preferable when implementing networked applications to abstract the physical network and build the application on a virtual network instead. The virtual networks are supposed to exhibit preferable properties, e.g. that they fix themselves on partitioning or that they allow to search for items faster than in the underlying network. Such networks are named overlay networks, as they build a network layer above an existing network. We distinguish unstructured and structured overlay networks.

Unstructured Overlay Networks  Unstructured Overlay Networks are overlay topologies based on random graphs. The topology consists of nodes which select their immediate neighbors using methods similar to broadcasting [SRS08], gossip protocols or random walks [CCR04]. The topology is unconstrained and generated at random by the participating nodes. To route a request to the node responsible for the request, broadcasting or random walks must thus be used. This allows queries to be flexible but complicates efficient use of the system.
1. Background

Figure 1.1.: Overlay networks can be unstructured. This example depicts an overlay network build on top of a physical network. The overlay network’s topology was build randomly and is partitioned although the underlying network does not exhibit partitioning. This can be due to nodes learning about other nodes over time: a node in the overlay might not yet know about nodes in another partition.

Popular applications for unstructured overlay networks are the Internet as build by TCP/IP, domain servers and the peer-to-peer file sharing application Gnutella. Sometimes unstructured overlay networks behave as small-world networks [Kle00], exhibiting small network diameters and a surprising efficiency although the topology itself is not constrained. This can be due to physical properties of the underlying network: when nodes choose their neighbors according to the communication latency between the nodes, clusters of well-connected nodes with fast links automatically form up in the generated graph.

Structured Overlay Networks  To overcome efficiency and performance weaknesses in unstructured overlay networks, Structured Overlay Networks (SONs) have been proposed. The general aim of SONs is to minimize the number of hops and latency required to route requests from the issuing node to the nodes which have to fulfill the request [CCR05].

SONs are created using decentralized topology-generating algorithms. Each node runs the algorithms to generate a part of the topology. The algorithms generate overlay networks which exhibit required guarantees. Examples for such guarantees are, among many others, restrictions on the diameter of the virtual network to minimize path length and latency, latency expectations for neighbors or that more than one path should exist between any pair of nodes. As distributed systems are inherently dynamic, coping well with churn (nodes leaving and joining the system) is often also expected. Depending on the application using the overlay network, an implementor has to chose the topology which fits the requirements best. Depending on the graph topology at hand, some requirements are more easily met than others. The topology of a SON must be
1.2. Distributed Hash Tables

Hash tables are a well-known family of data structures, represented by fixed-size tables of buckets. They are used to store key-value pairs efficiently in terms of access time. Finding a key-value pair within a hash table should take a constant amount of time, no matter how many pairs are stored in the table.

Formally, a hash table is a set of buckets $\mathcal{B}$, where each bucket $b \in \mathcal{B}$ can hold a large number of key-value pairs. Consider key-value pair $p = (k, v) \in \mathcal{K} \times \mathcal{V}$, where $\mathcal{K}$ is the set of keys and $\mathcal{V}$ is the set of values. Hash tables use functions $h : \mathcal{K} \mapsto \mathcal{B}$ to map a key $k$ to the bucket.
1. Background

where it should be found. The pair \( p \) is then stored in bucket \( h(k) \). As the bucket can contain more than one key-value pair, it must be searched again in the bucket. Hence, the number of elements per bucket should be small to find \( p \) in constant time.\(^1\) This is accomplished by the function \( h \): it should spread the available keys as evenly as possible over all available buckets. When the buckets are too full, more buckets are added and the stored items reshuffled to achieve constant-time operations.

A hash table offers three operations, each of which must compute the location of the item using \( h(k) \):

\( \text{put}(k, v) \) is used to store a key-value pair in the hash table.

\( \text{get}(k) \) is used to retrieve a key-value pair from the hash table.

\( \text{delete}(k) \) is used to delete a key-value pair from the hash table.

**Distributed Hash Tables** To store more data in a hash table than a single node can hold, the concept can be generalized into *Distributed Hash Tables (DHTs)* by spreading the data over the participating nodes. In a DHT, the state of the hash table is distributed over nodes. Each node has its own view on the DHT, where it knows the part of the hash table it stores itself and how it can request data stored on other nodes. Each node can be queried for any key \( k \in \mathcal{K} \).

The main service offered by DHTs is to lookup the responsible node for a key \( k \) [Gho06, p. 2]. This is similar to the hashing functions described before but more complex, as a lookup request needs to be routed to the node responsible for the key. So where a regular hash table computes the bucket that is responsible for key \( k \) by evaluating \( h(k) \), a DHT must potentially route the lookup request \( \text{lookup}(k) \) through many nodes until the right node is found. This happens every time a node receives requests for keys which it is not responsible for.

A DHT should be able to handle a large amount of data and nodes efficiently [Gho06, p. 2]. This includes finding the location of keys efficiently no matter where in the network they are stored. Some DHTs, e.g. Gnutella, use unstructured overlay networks and suffer from inefficiencies. Many DHTs therefore use SONs or hybrids of unstructured overlays and SONs. Technically, overlay networks and DHTs are often hard to distinguish. The DHTs in this thesis generate overlay networks.

\(^1\)Note that retrieving \( p \) might take longer than looking up its location. Hash tables only try to guarantee to lookup the location in constant time. Storing and retrieving items can take considerably longer as those operations also depend on other factors such as the size of the item.
The main aim of a DHT is to minimize the time needed to lookup keys. Many DHTs try to accomplish this by minimizing the needed number of hops. This is reasonable, as the time needed to cross a network link is often longer than the time needed for local computations. When a DHT implements a SON, this may oversimplify the actual time needed: the latency triangle inequality does not necessarily hold in a SON. Depending on the underlying physical network, using more hops in the overlay can be faster than using less hops. To mitigate this problem, some DHTs are designed to use latency additional information on routing or for topology generation. Common approaches are to pick nodes with low round-trip latency as neighbors while still creating a topology with the necessary diameter properties. Other approaches try to route over the fastest links while taking the risk that maybe more hops than theoretically needed are used.

**Routing Table Size and Maintenance Overhead** Another consideration in DHTs is the size of nodes’ routing tables, as depicted in Figure 1.3. Large routing tables allow lookup operations to be faster in terms of hops and latency, but every routing table entry creates maintenance overhead [Gho06, p. 6]. As DHTs will encounter churn, a node has to maintain each of its routing table entries to a certain degree. For example, a node might check if a neighboring node is still alive. Consequently, maintenance costs are increased with each additional routing table entry.² Depending on how much overhead per entry is produced, a size bigger than some constant \( c \) might not be feasible [XKY06]. showed that in general a routing table the size of \( O(\log n) \) is sufficient to route a lookup request in \( O(\log n) \) hops to the node responsible for the key, where \( n \) is the number of nodes in the system.

**Requirements** Ghodsi [Gho06, p. 10] specifies three properties which distributed hash tables should provide. Firstly, DHTs should scale with the number of nodes and thus routing must scale well. So for a number of nodes increasing linearly, the average distance between nodes should increase at most logarithmically. This implies that DHTs must be decentralized, although the degree of decentralization differs between implementations. Secondly, DHTs should load-balance well. As they are intended to store large amounts of data, no single node should suffer much larger loads than other nodes. This is accomplished by spreading data well over all nodes. Thirdly, a DHT should cope well under churn. As DHTs are distributed in nature, they have to work around the fact that nodes join and leave at will.

²Not all DHTs actively monitor neighboring nodes. A notable exception is Kademlia [MM02], the DHT backing the BitTorrent file sharing application. In Kademlia, routing tables tend to be large and requests are sent in parallel via UTP to a set of nodes. The size of the routing table is chosen such that, under certain statistical assumptions on the lifetime of nodes, at least one node responds to the request.
1. Background

Figure 1.3.: Asymptotic trade-off curve between the routing table size and the network diameter from [XKY06]. In general, the bigger the maximal routing table sizes, the smaller is the network diameter. Increasing the routing table size increases maintenance costs for the routing table entries. Monitoring neighboring nodes is costly and consequently the topologies of DHTs are designed to trade-off path lengths against maintenance cost.

1.3. Consistent Hashing and Chord

Regular hashing algorithms try to spread items as balanced as possible over a fixed set of buckets. This allows very fast lookups, but causes high costs when the number of buckets changes, as items have to be moved to other buckets (rehashing). This is not a problem when the number of buckets remains stable for long periods, but becomes a burden when trying to distribute a hashing algorithm over nodes connected via a network. Especially in the Internet, were churn is a problem for every distributed system of non-negligible size, the number of nodes changes all the time. Therefore, frequent rehashing is required.

The global state of a distributed system of considerable size can not be observed consistently. This becomes another problem of distributing hash algorithms: Not only is the set of buckets dynamic, but two nodes might see a different subset of buckets at one instant of time. A distributed hash table must deal with such differing views on the set of buckets.

In this section we will introduce the concept of consistent hashing as proposed by Karger et. al. in [KLL+97], which addresses both the problem of costly rehashing as well as handling differing views on the set of buckets. Following this introduction, we will describe Chord [SMLN+03], a DHT build on consistent hashing.
1.3. Consistent Hashing and Chord

1.3.1. Ranged Hash Functions and Consistent Hashing

Consistent hashing describes a family of hashing algorithms which try to handle dynamic changes to the set of buckets gracefully. In a distributed hashing algorithm, where buckets are spread over different nodes, this means that events such as node joins and leaves should cause as little movement of stored items as possible. Only as many items as needed to keep the overall load of nodes should be moved. [KLL+97, p. 6] generalizes this requirement into the concept of ranged hash functions. A ranged hash function differs to an ordinary hash function: when mapping, not only the key to be mapped is considered, but also a node’s view on the set of buckets. Given the set of items \( I \) and a set of buckets \( B \), a view \( \mathcal{V} \) is any subset of \( B \). Then, let a ranged hash function be \( f : 2^B \times I \mapsto B \). For families of such ranged hash functions, Karger et. al [KLL+97] define four »notions of consistency« which a ranged hash family should fulfill.

**Balance** A ranged hash family is called balanced if the fraction of items which is mapped to each bucket is low, so for a view \( \mathcal{V} \), only \( O\left(\frac{1}{|\mathcal{V}|}\right) \) items should be mapped to each bucket, with high probability (w.h.p.).

**Monotonicity** As items should only move when necessary, items stored in one bucket can only move to another bucket if the target bucket enters the view. Thus, for views \( \mathcal{V}^{\text{old}} \subseteq \mathcal{V}^{\text{new}} \), items should not move from a bucket \( b^{\text{new}} \in V^{\text{new}} \) to an old bucket \( b^{\text{old}} \in \mathcal{V}^{\text{old}} \) or between old buckets \( b^{\text{old},1}, b^{\text{old},2} \in \mathcal{V}^{\text{old}} \).

**Spread** The views of different nodes should be as consistent as possible. Their respective ranged hash functions should map an item \( i \) to as few buckets as possible. This is needed to quickly find items. The location of an item is a consensus decision. Such a consensus can be reached faster if the number of different opinions is low. Consequently, given all views, the ranged hash function should spread the item over as few buckets as possible.

**Load** The load is the maximum load of all buckets. This number should be as small as possible.

Before we discuss an implementation of a ranged hash family called consistent hashing, we will introduce one more property of ranged hash functions which is explained informally in [KLL+97]. The authors motivate the smoothness of a ranged hash function as the amount of items to be moved when one view \( \mathcal{V}^1 \) evolves into a view \( \mathcal{V}^2 \) due to a change in the number of known buckets. In distributed hash tables, this would be due to a change in the number of nodes or because a node learns about another node just now. Naturally, the number of moved

---

3With high probability means that an event occurs with a probability of at least \( 1 - \frac{1}{N} \) for a parameter \( N \in \mathbb{R} \). Here, \( N \) is the number of buckets.
items should be as small as possible and consequently the ranged hash function should change as smoothly as possible. Quoting [KLL+97]: »The “smoothness” property implies that smooth changes in the set of caching machines are matched by a smooth evolution in the location of cached objects«. So, a smooth change in the set of buckets B results in a smooth change in the item mapping.

**A Consistent Hashing Algorithm**

One implementation of ranged hash functions is called *Consistent Hashing* [KLL+97]. Consistent hashing aims to improve caching web pages over multiple caching nodes while keeping the overhead to handle churn as small as possible [Gho06, p. 3]. The concept is general enough to be used for DHTs as well, as we will see in Section 1.3.2. We will discuss non-distributed consistent hashing as described by Karger et. al. in [KSB+99] before explaining the distributed implementation Chord in Section 1.3.2.

Consider a base hash function $h_b : K \mapsto [0, 1]$. The interval $[0, 1]$ can be thought of as a circle with unit circumference. $h_b$ is also used to map the buckets onto that circle. Each bucket is assigned to an identifier $id \in K$ using a mapping function $h_m : B \mapsto K$. The position of bucket $b$ on the unit circumference circle is consequently $h_b(h_m(b))$. See Figure 1.4 for an example. Assigning an item to a bucket is straightforward: an item is mapped to the first bucket encountered when moving clockwise along the circle. In Figure 1.4, $i_1$ is mapped to the bucket $N_1$ and $i_2$ is mapped to the bucket $N_2$.

**1.3.2. Chord**

*Chord* is a fully distributed, uniform hash table [SMLN+03]. All participating nodes fulfill the same tasks and only the first node is special to boot up the system. The basic topology of Chord is the same as for consistent hashing with nodes and items being located on a ring. Each node functions as a bucket. From now on, we will call the space from which identifiers for keys and nodes are chosen the *identifier space*. In consistent hashing, the identifier space would be $I_{\text{Cons}} = [0, 1]$. In contrast to consistent hashing, Chord’s identifier space is a modular circle $I_{\text{Chord}} = \{0, 1, \cdots, 2^m - 1\}$, for some constant $m$. The constant $m$ depends on the base hash function, as $m$ should equal the bit length of value hashed with that function. [SMLN+03] proposes *SHA-1* as the base hash function, but any other cryptographically strong hash function can be used. A node $n$’s identifier $id(n)$ is its hashed IP address. We will use the identifier $id(n)$ of node $n$ as a synonym for its $n$.
1.3. Consistent Hashing and Chord

Figure 1.4.: Example of a consistent hashing implementation using a circle with unit circumference. \( N_1, \ldots, N_4 \) are buckets placed on nodes, \( i_1, \ldots, i_3 \) are items stored in buckets. A bucket is responsible for all items placed between itself and its predecessor in counter-clockwise direction.

The base hash function should distribute keys uniformly over the ring. This is important as the hash function balances load: if the distance between two nodes is roughly equal, they will be responsible for a similar number of items. This follows from the fact that Chord is derived from consistent hashing and consequently the balance, load and spread properties hold for it, assuming the base hash function is well-chosen. »Theorem IV.1« [SMLN+03] formalizes this informal description:

For any set of \( N \) nodes and \( K \) keys, with high probability, the following is true.

1. Each node is responsible for at most \( (1 + \varepsilon) \frac{K}{N} \) keys.

2. When an \((N + 1)\)th node joins or leaves the network, the responsibility for \( O(\frac{K}{N}) \) keys changes hands (and only to or from the joining or leaving node).

When consistent hashing is implemented as described above, the theorem proves a bound of \( \varepsilon = O(\log N) \). The consistent hashing paper shows that can be reduced to an arbitrarily small constant by having each node run \( \Omega(\log N) \) virtual nodes [...]

Both properties above (load per node and number of items to be moved on join or leave) are important for a distributed hash table. To further our discussion of Chord, we will now consider
1. Background

the resulting topology and how request routing is implemented. Afterwards, we will discuss the ring maintenance to maintain the topology, then Chord’s efficiency in terms of hops for routing a request, and we will conclude with the problem of topology inconsistencies.

Topology and Routing All nodes in a Chord ring have one predecessor and one successor. The correctness of routing and ring maintenance are based on well-maintained successor and predecessor links. The predecessor and successor links form the ring structure of Chord. Requests are routed only in clockwise direction around the ring. To further understand Chord’s topology, we need to define the distance $d$ between two identifiers $n, m \in I$ [NS11]:

\[
    d(n,m) = \begin{cases} 
    2^m & \text{if } n = m \\ 
    m - n & \text{if } m > n \\ 
    2^m - n + m & \text{if } m < n 
    \end{cases} 
\]

(1.1)

It is important to note that the distance from an identifier to itself equals $2^m$ and not zero. The distance is asymmetric as requests are routed in clockwise direction only.

A simple routing algorithm would route a request along the successor links until the node responsible for the key is found, resulting in $O(N)$ overlay hops given $N$ nodes. To minimize the number of hops needed to fulfill request, additional links are added to the topology, called fingers, shortcutting the ring. Figure 1.5 demonstrates this: node $s$ contains a routing table with a set of fingers $\{f_i\}_{i=1,...,m}$. The last finger $f_m$ points the farthest along the ring, at least halving the distance $d(s,s)$. The second-to-last finger $f_{m-1}$ halves the distance between the node $s$ and $f_m$, and so on. Hence, the fingers bisect the identifier space, yielding a diameter of $O(\log N)$ w.h.p. and thus speed up request routing. We will discuss the algorithm to generate the fingers after we discussed the routing algorithm used in Chord.

To lookup a key $k$ in Chord, Algorithm 2 is used. If the current node $s$ is not responsible for the key is found, resulting in $O(N)$ overlay hops given $N$ nodes. To minimize the number of hops needed to fulfill request, additional links are added to the topology, called fingers, shortcutting the ring. Figure 1.5 demonstrates this: node $s$ contains a routing table with a set of fingers $\{f_i\}_{i=1,...,m}$. The last finger $f_m$ points the farthest along the ring, at least halving the distance $d(s,s)$. The second-to-last finger $f_{m-1}$ halves the distance between the node $s$ and $f_m$, and so on. Hence, the fingers bisect the identifier space, yielding a diameter of $O(\log N)$ w.h.p. and thus speed up request routing. We will discuss the algorithm to generate the fingers after we discussed the routing algorithm used in Chord.

To lookup a key $k$ in Chord, Algorithm 2 is used. If the current node $s$ is not responsible for the key $k$ (hence, if $k$ is not placed between $s$ and it’s predecessor $s.pred$), then the request must be forwarded to another node. If the successor $s.succ$ of $s$ is responsible for the key, the request is directly forwarded to this node. Otherwise the request is forwarded to the finger $f_i$ farthest away from $s$ while still being in front of the key $k$.

We will use a different routing algorithm later in this thesis, depicted in Algorithm 3. When considering Chord in the current chapter, we will use the algorithm as presented in the original paper [SMLN+03]. Both algorithms have in common that each routing step should at least halve the distance to $k$. This expectation is met if the distance between the fingers grows exponentially. Consequently the maximum path length for a lookup is w.h.p. $O(\log 2^m = m)$. The bound can be sharpened further, as will be discussed later.
1.3. Consistent Hashing and Chord

Figure 1.5.: A Chord ring example. Node $s$ has links (fingers) to four other nodes \{${n_0, n_1, n_2, n_3}$\}, as depicted by its routing table. Node $s$ also knows its immediate predecessor and successor nodes. The fingers are not picked at random, but must satisfy a property for $s$. $n_0$ halves the distance of $s$ to itself, $n_1$ halves the distance between $s$ and $n_0$, $n_2$ halves the distance between $s$ and $n_1$, and so on. A node can have up to $\log_2 m$ fingers. $s$ views every known node as responsible for the identifier range between itself and the respective predecessor. Finger $f_j$ is the node with the largest identifier in $[s + 2^{j-1}, s + 2^j)$. Requests to keys within that interval are forwarded to finger $i - 1$.

Algorithm 2 Finding the Successor of an Identifier in Chord

```plaintext
message s.find_successor(id)
  if id ∈ (s, s.succ] then
    return s.succ
  else
    return s.succ.find_successor(id) ▷ Forward around the ring
  end if
end message
```

Ring Bootstrap and Maintenance A Chord ring is bootstrapped from a first node, as depicted in Algorithm 4. The first node forms a ring by using itself as its successor. It also sets its predecessor to nil.

When a node wants to join a ring, it needs to know at least one node of the ring. The message handler Algorithm 5 for `join(n)` is responsible for that: the current node $c$ uses $n$ to lookup its
1. Background

Algorithm 3 Enhanced Chord Routing

```plaintext
message s.lookup(k)
    if k ∈ (s.pred, s] then  ▷ s is responsible for k
        return local key
    else if k ∈ (s, s.succ] then  ▷ The succeeding node is responsible for k
        s.succ.lookup(k)
    else  ▷ Forward along the ring
        l ← longest finger fi such that d(fi, s) > d(k, s)
        l.lookup(k)
end if
end message
```

Algorithm 4 Chord Bootstrap

```plaintext
message c.create()
    pred ← nil
    succ ← c  ▷ Node knows only about itself
    end message
```

successor.

The ring is not immediately consistent after a node joined the ring. Although the new node should now be able to reach any other node in the ring, other nodes cannot reach it yet. The new node’s predecessor is not yet aware of the new node, and no finger entries point to the new node. This inconsistency is handled by the periodic ring maintenance as shown in Algorithm 6.

Chord maintains the ring proactively: after a period of time, a node refreshes its fingers and verifies that its successor and predecessor nodes are still part of the ring. Finger maintenance is performed by the message handler fix_fingers: counting i down from m to 1, each key $c + 2^{i-1}$ is looked up. The node responsible for the key is going to be the finger $f_i$. The successor and predecessor have to be checked periodically as well. The successor of a node is verified in c.stabilize(), which asks the successor node for its predecessor. If c.succ.pred is not c, then a node exists between c and c.succ. c chooses this node n’ to be its successor and notifies n’ about its decision. The successor n’ receives a message notify and assigns c to n’.pred if c is closer to n’ than the current n’.pred.

Predecessor nodes are checked periodically by check_predecessor. The message handler validates that c.pred is alive using a failure detector. c.pred is set to nil if the predecessor is not alive. The periodic stabilization of the real predecessor p will notify c later about the
Algorithm 5 Chord Join Ring

message $c$.$join(n)$ \quad $\triangleright$ $c$ should join with help of $n$

\[
c$.pred$ \leftarrow nil\]
\[
c$.succ$ \leftarrow n$.find_successor($c$)
\]
end message

message $c$.find_successor($id$) \quad $\triangleright$ Ask $c$ for the successor of identifier $id$

if $id \in (c,c$.succ$)$ then

\[
return c$.succ$
\]
else

\[
n \leftarrow closest_preceding_node(id)$
\]
\[
return n$.find_successor(id)$
\]
end if
end message

message $c$.closest_preceding_node($id$) \quad $\triangleright$ Ask $c$ for its closest finger before $id$.

for $i = m,m-1,\ldots,1$ do

if Finger $i \in (c,id)$ then

\[
return$ Finger $i$
\]
end if
end for

\[
return c$
\]
\quad $\triangleright$ No closest finger means that $id \in (c,c$.succ$)$
end message

existence of $p$.

A Chord ring transitions between consistent and inconsistent states. »Theorem IV.3« in the original paper [SMLN+03] claims that inconsistent states are transient:

If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor pointers will form a cycle on all the nodes in the network.

During the inconsistent state of the ring, lookups might not reach the node responsible for the key. Eventually, the ring is expected to have settled down and routing should again work as intended. There are corner cases which prevent the ring from fixing itself. If a network split occurs and the ring is partitioned into non-linked intervals (so no node from interval $a$ can reach a node from interval $b$ and vice versa), ring maintenance will create a new ring for each
interval. This has to be detected and handled outside of Chord. Such pathological cases are ignored, as they are claimed to not occur from nodes joining the ring.

**Efficiency** We consider two aspects of efficiency for Chord: the expected number of hops to lookup a key \( k \), and the expected size of the routing table per node.

The diameter of a Chord ring is, w.h.p., \( O(\log N) \) and thus a key lookup using Algorithm 2 takes \( O(\log N) \) hops w.h.p. [SMLN\textsuperscript{+}03, p. 22]. Experimental results show that the expected number of hops for a lookup follow a normal distribution with an average of \( \frac{1}{2} \log N \) [SMLN\textsuperscript{+}03, p. 26].

The routing table of a Chord node \( s \) can have a maximum of \( m = \log_2 N \) entries, as the maximum distance according to Eq. (1.1), and each finger \( f_j \) halves \( d(s, f_{j+1}) \). Depending on the node

Algorithm 6 Chord Periodic Stabilization

```
message c.stabilize() ▷ Verify c’s successor and tell the successor about c
  x ← c.succ.pred ▷ Get the successor’s predecessor
  if x ∈ (c,c.succ) then ▷ Found new successor
    c.succ ← x
  end if
  c.succ.notify(c) ▷ Notify the successor that c might be its predecessor
end message

message c.notify(n) ▷ Update predecessor if needed
  if c.pred = nil or n ∈ (c.succ,c) then
    c.pred ← n
  end if
end message

message c.fix_fingers() ▷ Refresh routing table entries
  for i = 1,...,m do
    f_i ← find_successor(c + 2^{i-1})
  end for
end message

message c.check_predecessor() ▷ Uses a failure detector to check the predecessor
  if c.pred has failed then
    c.pred ← nil
  end if
end message
```
1.3. Consistent Hashing and Chord

distribution, not all routing table entries of a node are unique: when the ring is partitioned into intervals of length $\frac{2^m}{N}$ and assuming that nodes are distributed uniformly over the ring, each interval contains an average of one node [CS05]. This implies that for some node $s$, w.h.p. no other node will be found in the interval $(s, s + \frac{2^m}{N}]$. Consider fingers $f_j$ covering intervals $(s + 2^{j-1}, s + 2^j]$ such that $s + 2^j \leq \frac{2^m}{N}$ holds for $j$. Those fingers point into the interval $(s, s + \frac{2^m}{N}]$. It follows that $j \leq m - 2\log_2 N$, which means that the first $m - 2\log_2 N$ entries of the routing table are identical and point to $s$.succ.

The routing efficiency of Chord dependents on the current ring state. Due to churn, a Chord ring is inherently unstable. [SMLN+03] points to [LNBK02] which claims that Chord works efficiently under churn, provided that the stabilization routine is run at a certain rate.

**Note on Correctness** Request routing correctness relies on the successor links: in the worst case, routing can happen along the successor links and will reach the node responsible for the key eventually. If successor nodes fail, this invariant can break. Hence, [SMLN+03, p. 24] advises to use a list of successors of length $r$ instead of a single successor node. On failure, the immediate successor can be replaced with the second successor. If nodes fail with equal probability $p$, at least one successor survives out of $r$ with probability $1 - p^r$. The original paper contains »Theorem IV.5« to formalize this argument:

If we use a successor list of length $r = \Omega(\log N)$ in a network that is initially stable, and then every node fails with probability $\frac{1}{N}$, then with high probability `find_successor` returns the closest living successor for the query key.

We refer the reader to [Zav12], which shows that one can generate situations where joining or leaving nodes can disturb a Chord ring in ways which are unfixable by the basic protocol. Chord implementations regularly work around this issue, but only in recent work have some problems been revealed. In the upcoming section, we briefly discuss lookup consistency, which is closely related to the basic routing correctness.

1.3.3. Routing Inconsistencies

Chord’s transitioning between different states of stable rings creates inconsistencies, as has been noted above. Although many inconsistencies are fixed automatically by periodic stabilization, they cannot be ignored, as lookup inconsistencies can lead to data inconsistency. We distinguish two types of ring inconsistencies: inconsistencies which will be fixed over
1. Background

Figure 1.6.: Example for a lookup inconsistency from [SSM+08, p. 3]. Node $N_1$ false detects $N_2$ and $N_3$ as dead. Consequently, looking up $k$ at node $N_1$ returns $N_4$, whereas the same lookup at node $N_2$ returns $N_3$.

Transitory inconsistencies, and inconsistencies which break invariants of Chord. We will discuss fixable transient inconsistencies and cite work by Shafaat et. al. [SSM+08] on how to reduce problems to a negligible factor. Additional work on the topic can be found in [Zav12], where Zave discusses circumstances under which the Chord protocol breaks.

**Transient Inconsistencies** [SSM+08] defines a DHT’s *Configuration* as the set of all nodes and the nodes’ fingers. The others also define the consistency of a configuration:

> A configuration of the system is consistent if, in that configuration, lookups made for the same key from different nodes, return the same node.

We call this property the *lookup consistency*. Chord’s routing weakens this property to *eventually lookup consistent*. While the ring is inconsistent, lookups for the same key can return different nodes. When the ring is fixed by periodic stabilization, lookups for the same key are expected to return the same nodes. Thus, eventually the configuration will be consistent. This works fine when the DHT is used to serve cached replicas of static content, but is problematic if the served content is dynamic. Applications which depend on stronger consistency guarantees have to implement additional tools on top of a DHT to work around the eventual consistency.

Figure 1.6 depicts an example how a ring inconsistency can lead to inconsistent lookups. Consider a slice of the ring with four consecutive nodes. Node $N_1$ believes both $N_2$ and $N_3$ to be dead, thus setting its successor to $N_4$. The other nodes are well-connected as intended by Chord. Looking up key $k$, which is located between node $N_2$ and $N_3$, at nodes $N_1$ and $N_2$ returns different results. $N_1.\text{lookup}(k)$ will thus yield $N_4$, whereas $N_2.\text{lookup}(k)$ will yield the correct $N_3$. This type of inconsistencies can be solved by forwarding a request to the node...
1.3. Consistent Hashing and Chord

Figure 1.7.: When using local responsibilities to avoid lookup inconsistencies, keys can become unavailable until a ring inconsistency is fixed. Such responsibility inconsistencies can slow down lookups and create path lengths longer than $O(\log N)$. This example from [SSM+08, p. 3] depicts such a situation. Node $N_2$ joins the ring (i), fails (ii) and comes back soon after $N_1$ has detected it as dead (iii). Looking up keys between $(N_1,N_2)$ at node $N_1$ skip $N_2$ and are declined by $N_3$. This results in longer lookup paths as requests have to be routed around the ring again.

Responsibility inconsistencies are transient if the ring inconsistency is fixed over time, but they can slow down lookups considerably. To solve the issue of lookup inconsistencies while still guaranteeing the availability of keys, [SSM+08] proposes to use non-disjoint replica sets: by replicating a key to other nodes, the probability of lookup inconsistencies can be reduced to a negligible quantity. We will not discuss this proposal further, as using replicas to fix routing issues is out of this thesis’ focus.

1.3.4. Enhancing Chord

The Chord version presented in this section leaves multiple opportunities for enhancement. In the years after Chord had been proposed, a large number of optimizations was published. We give a very short summary of three optimizations.

**S-Chord** Gennaro and Sala proposed a protocol based on a Chord ring [MCR03]. In contrast to Chord, they propose that fingers should be spread symmetrically around the ring. With
1. Background

that, path lengths may be reduced.

2 Chord-halved Mesaros et. al propose another protocol based on a Chord ring [CS05]. 2 Chord-halved eliminates the need for periodic finger stabilization by informing nodes which are interested in topology changes about the changes.

Proximity Neighbors Selection Chord minimizes only path lengths. If latency should be partially minimized as well, proximity neighbor selection (PNS) can be used [GGG+03a]. In PNS, the Chord protocol is changed such that not the last node of an interval is chosen as a finger, but any node of the same interval with minimal latency.

The optimizations have in common that they require an adaption of Chord’s basic algorithms. Optimizations to Chord, or any classic DHT algorithm in general, require fundamental changes to the protocols. In the following section, we will present flexible routing tables (FRT). FRT provides a framework to consider more requirements simultaneously when building routing tables. It aims to be extensible in order to allow optimizations without requiring that the underlying protocol must undergo fundamental changes.
2. Flexible Routing Tables

Nagao and Shudo proposed a generalized approach to designing routing tables for SONs in [NS11], called *Flexible Routing Tables (FRT)*. Their concept is the basis of this thesis. In this chapter, we will present two FRT instances, FRT-Chord and GFRT-Chord, and their theoretical foundations. Our implementation of the two algorithms will be presented in Chapter 4.

2.1. Motivation

Classic DHTs like Chord focus on preserving topology requirements when building the structured overlay network. Chord as presented in Section 1.3.2 uses the factor 2 as the base of the exponentially increasing distances between routing table entries. Some implementations use other values to add more fingers, but will still use a constant base. It is due to the constant base that a Chord node’s routing table contains duplicate entries in the lower part.\(^1\) This is one example among others were topology restrictions impact the set of routing tables \(P\) a node may generate. We define \(P \subseteq P(\mathcal{N})\), where \(\mathcal{N}\) is the set of all nodes and \(P\) computes a set’s power set. A node chooses its routing table \(E \in P\) by computing it using the supplied algorithms.\(^2\)

In Chord, a node chooses \(E\) by computing the entries with the algorithms described in Section 1.3.2. To motivate flexible routing tables, we begin with the example given in Figure 2.1. The example depicts a slice of a Chord ring and some nodes on that slice. Node \(s\) must generate its routing table. We assume that \(s\) has information about all other nodes in the slice and can thus theoretically use every node as an entry for the routing table. Node \(s\) is allowed to use up to three routing table entries, as the slice consists of three intervals. Each interval is twice as large as the interval in front of it. Consider Chord for this example. Here, node \(s\) picks the routing table \(\{1, 4\}\), as described in the caption of Figure 2.1. The Chord rules of picking a routing table from \(P\) can be relaxed such that any of routing tables marked green could be

\(^1\)We discussed this in Section 1.3.2 before: up to \(m - 2\log_2 N\) routing table entries point to the same node, where \(N\) is the number of nodes in the ring and \(m\) defines the identifier space \(I = [0, 2^m - 1]\).

\(^2\)We named the routing table \(E\) as an abbreviation of *edges*. The routing table entries describe nodes to which the owner of the routing table is connected via overlay edges.
2. Flexible Routing Tables

Figure 2.1.: Chord’s topology generator removes many routing tables from \( \mathcal{P} \). The example depicts a node \( s \) and 5 subsequent nodes. \( s \) can use up to three routing table entries for the displayed ring slice, as an entry should point into each of the intervals separated by vertical lines. The first interval contains no node, so the first routing table entry points to node 1. Node 1 is also the value of the second routing table entry. Consequently, out of all possible combinations to build routing tables of size three or less, a Chord node will pick a routing table from only three tables. The Chord algorithm presented in Section 1.3.2 picks the routing table \{1, 4\}.

used. Each of the highlighted routing tables has got an entry for each non-empty interval. This allows proximity neighbor selection (PNS) in \cite{GGG+03b}: Out of an interval’s nodes, the one reachable with minimal latency is chosen. This possibly speeds up lookup latency. Hence, as a second requirement after maintaining \( O(\log N) \) hops performance, the requirement to minimize latency can be added. Unfortunately, as can be seen in Figure 2.1, many possibly interesting routing tables are already dismissed and cannot be considered for PNS. Consider the routing table \{2, 4\} and further assume that the latency to node 2 is much less than the latency to node 1. Node 2 is very close to node 1 in terms of overlay distance and the latency benefit could outweigh the potential lookup penalty. Due to Chord’s algorithms, picking \{2, 4\} is not allowed, even if that node combination would outperform any other combination. The generalization of this phenomenon led to the idea of flexible routing tables. Instead of partitioning \( \mathcal{P} \) into two sets \( \mathcal{P}_{\text{allowed}} \) and \( \mathcal{P}_{\text{not allowed}} \), FRT aims to consider \( \mathcal{P} \) as a whole to pick the best routing table using possibly multiple requirements. It is thus natural to generate routing tables lazily instead of computing the entries using an algorithm. We will proceed with an introduction of FRT’s concepts.

2.2. The Concept of Flexible Routing Tables

Nagao and Shudo define a generalized approach to building routing tables for DHTs \cite{NS11}. Consider a node \( s \). This node continuously refines its routing table by learning new nodes.
When the maximum routing table size is reached, an entry must be deleted. As mentioned in Section 1.2, routing tables are often size-restricted to lower maintenance costs. Entry deletion (or filtering) is done by removing the entry such that the resulting routing table is the best from a set of routing table candidates $C$. $C$ consists of all routing tables built from the original table by removing one element. Note that this set is usually not computed, as the algorithms which will be described later directly compute the entry to be deleted. The definition which routing table among $C$ is best depends on the application. Nagao and Shudo generalize the notion into a total order $\leq_{\text{ID}}$, which is to be defined by the designer of a DHT. $C$ is then ordered using $\leq_{\text{ID}}$. The best routing table is defined as $E' \in C$ such that for all $E \in C$, $E' \leq_{\text{ID}} E$.

Common routing schemes partition the set of routing tables $\mathcal{P}$ into two subsets: a set of allowed and a set of non-allowed routing tables. Figure 2.1 exemplified this. A node samples its routing table from the set of allowed tables. This hinders combining multiple requirements into topology, e.g. if the routing protocol should minimize both the number of overlay hops as well as the lookup latency. In the example of a Chord ring, those two properties are diametrically opposed: as the latency triangle inequality does not always hold in the overlay, doing more hops than necessary can sometimes decrease latency. This conflicts with the property of minimizing hops. When $\mathcal{P}$ is partitioned by using the minimizing hops requirement before taking measures for latency minimization, subsequent optimizations like PNS will not generate the best possible result.

Nagao and Shudo propose that a routing scheme should not partition $\mathcal{P}$ prematurely. Instead of first designing the structure of a routing table and then sampling routing tables from $P_{\text{allowed}} \subseteq \mathcal{P}$, they propose that a routing table should be able to contain all nodes without being subject to restrictions such as node identifier conditions. Instead, sampling a routing table should happen by sampling the best routing table of a set of tables. This is different to the traditional approach of creating routing tables, where only selected nodes are allowed to be added to the routing table.

### 2.2.1. FRT Primitives

FRT is based on three primitives executed by each node.

**Guarantee of Reachability** This procedure is responsible for guaranteeing that routing succeeds.

**Entry Learning** The procedure *entry learning* is responsible for entering new routing information into the routing table. A node creates a new routing table entry for each encountered
2. Flexible Routing Tables

node which is not yet contained in the routing table. Encountering an unknown node can be on purpose, as is the case for Chord where nodes are sampled by querying specific keys, or can happen lazily, e.g. when forwarding the request of an unknown node.

**Entry Filtering** As mentioned above, if routing tables are size-restricted, a routing table can exceed the bound due to entry learning. When this happens, entries have to be filtered. *Entry filtering* ensures that size restrictions are met. For this, a routing table is sampled from \( C \). The entry to be filtered is chosen such that the resulting routing table \( E' \in C \) obeys \( E' \leq_{ID} E \) for all \( E \in C \).

An implementation of a DHT based on FRT has got algorithms to implement each primitive. We will describe FRT-Chord and GFRT-Chord in Sections 2.3 and 2.4 below, which exemplify how the primitives might be implemented. Note that, depending of the wanted topology, the algorithms of different DHTs based on FRT differ to a large degree.

2.2.2. Expected Advantages

FRT is expected to have features which are hard to achieve in traditional routing table designs. Firstly, it allows to move seamlessly between fully-meshed and multi-hop networks. In networks with a small number of nodes, it can be useful to connect all nodes with each other directly. FRT-based routing schemes allow this, as entry filtering is only done when a routing table reaches its maximum size. The maximum size can itself be dynamic, situation-dependent and different for different nodes. When the routing tables are filled and thus a multi-hop network must be set up, nodes adapt to that new need themselves by starting to filter routing table entries. Secondly, FRT uses all encountered information, as routing information is only lost when routing table entries are filtered.\(^3\) A node which is learned by entry learning will be added to the table, regardless which properties will be enforced later. This contrasts other routing schemes like Chord, which will only add nodes to its routing table whose identifier fits into a certain category, i.e. is part of a specific identifier range. Thirdly, FRT is flexible due to using all available information. Routing tables can be constructed without being agnostic to the current ring state. This becomes clear when moving from fully-meshed to a multi-hop network: whereas this would mean a fundamental change to the Chord protocol, it is natural for FRT-based algorithms to do this step. The original paper [NS11] mentions more advantages, which we will not repeat here.

\(^3\)This discussion omits the case of unavailable nodes. When a node is detected as unavailable by its neighbors, it will be filtered from their routing table as well. This does not contradict FRT's property of avoiding to lose information, as a dead node is not useful for routing.
2.3. FRT-Chord

2.2.3. Examples of FRT

[NS11] introduces two instances of the FRT concept: FRT-Chord and GFRT-Chord. Both are based on a Chord-like ring topology. In contrast to Chord, nodes using FRT-Chord and GFRT-Chord do not compute the ranges where fingers should point, but instead learn routing table entries when communicating with other nodes or by using active learning lookups (issuing lookups for keys chosen from a probability distribution). Consequently, FRT-Chord and GFRT-Chord do not depend on a uniform distribution of nodes over the ring, as all routing table entries can be used and no restrictions apply for the entries. The distance between consecutive routing table entries is expected to grow exponentially if many nodes exist in the ring, but the base of the distance increase is not fixed. This is in contrast to Chord, where the distances between two consecutive fingers is increased by at least a factor of 2.

GFRT-Chord extends FRT-Chord by grouping nodes: a node uses static information to determine fingers pointing to nodes within the same group, e.g. nodes within the same data center. We will discuss the approaches at length in this section, as they are the base for flexible routing tables in Scalaris and thus the main subject of this thesis.

2.3. FRT-Chord

FRT-Chord uses Chord’s identifier space \( I = [0, 2^m - 1] \), with keys and nodes being mapped into \( I \) using an appropriate base hash function. Lookups are routed clockwise around the ring in the same way as by Chord’s `find_successor` algorithm. In this section, we will consider the process of creating a routing table at a node \( s \). The general aim of FRT-Chord is to provide routing with logarithmic path lengths.

FRT-Chord uses flat routing tables. Except for markers to distinguish non-filterable sticky nodes (typically predecessor and successor nodes) from normal nodes, the routing table does not distinguish between node types. A routing table \( E \) is a set of entries \( \{e_i\}_{i=1,...,|E|} \), where [NS11] assumes that each entry \( e_i \) is a pair of a node identifier \( e_i.id \in I \) and a pair \( e_i.addr. e_i.addr \) consists of the entries IP address and port. Similar to the original paper, we will use \( e_i \) and \( e_i.id \) interchangeably. \( E \) is further arranged in ascending order of entry distance to \( s \), so for indices \( i, j = 1, \ldots, |E|, i < j \Rightarrow d(s, e_i) < d(s, e_j) \) holds. The distance function \( d \) is defined as for Chord Eq.(1.1). Note that because of the routing table’s order \( i \neq j \Rightarrow d(s, e_i) \neq d(s, e_j) \) holds, and consequently no two entries point to the same node. Also due to the ordering of the table, \( e_1 \) is the successor of \( s \), while \( e_{|E|} \) is its predecessor. The successor and predecessor are
2. Flexible Routing Tables

![Diagram](image)

(a) Before $n_1$ learned $n_4$.  
(b) After $n_1$ learned $n_4$.

Figure 2.2.: Entry learning in FRT-Chord is lazy: when a new node is encountered, it is immediately added to the routing table.

non-filterable sticky nodes (or also called *sticky entries*). FRT-Chord’s *guarantee of reachability* is based on the sticky nodes. As they are non-filterable, routing should always be possible to succeed at least by using the sticky nodes. This is similar to Chord’s routing guarantees, which are based on maintaining a node’s successor lists. The maintenance of sticky entries is out of scope in [NS11]. The authors assume that the sticky entries are well-maintained. One solution for maintaining the sticky entries is to use Chord’s periodic stabilization routine.

**Entry Learning**

[NS11] proposes three ways how a node could learn its routing table entries. The initial set of entries could be learned by transferring the routing table of $s$’s successor to $s$. This is sensible as the successor was active for a longer time and thus has got a routing table which includes some entries. Additional nodes can be learned by $s$ on communication with other nodes. When $s$ receives messages which includes node information, $s$ is able to add the node information to its routing table. This can be the case when forwarding lookup requests, for example. The last method proposed in [NS11] is to learn new routing table entries by doing active learning lookups. Active learning lookups are lookups for keys generated by a random number generator in a similar fashion as done by Symphony [MBR03].

These three methods of entry learning are general enough to be used in other FRT schemes as well. FRT itself does not restrict the means to learn new nodes to these three procedures, so »that opportunities to refine a routing table will not be wasted« [NS11, p. 75]. With respect to [Gho06, p. 6], this approach to topology maintenance can be named *lazy*, as the topology is not maintained using a strict topology-generating algorithm.
Figure 2.3.: The cumulative distribution function for FRT-Chord’s active learning lookups grows steeply for small values of \( x \in I = [1, 2^m] \). This example depicts the cumulative distribution function node at node \( s \) with identifier 0 within a ring of size 1024, where the number of nodes equals the ring size. Here, the probability to pick an identifier \( x \) with \( d(s, x) \leq 64 \) is larger than to pick any \( x \) with \( d(s, x) > 64 \).

Implementing routing table transfer on node join and adding nodes to a routing table on communication is straightforward. To implement active learning lookups, a random number generator based on a probability distribution \( p(x) \) is needed. Nagao and Shudo [NS11] propose a cumulative distribution function \( F \) (see Equation (2.1)).

\[
F(x) = \begin{cases} 
\frac{\ln d(s, x)}{\ln d(s, e_1)} & d(s, e_1) < x < d(s, e_{|E|}) \\
\frac{\ln d(s, x)}{\ln d(s, e_{|E|})} & \text{otherwise} 
\end{cases}
\]

The probability to pick \( x \) from all possible keys when using \( F(x) \) is »in inverse proportion to the distance from \( s « [NS11, p. 75]. The distribution is more likely to produce an \( x_1 \) near \( s \) than an \( x_2 \) far from \( s \). See Figure 2.3 which depicts a nodes \( F(x) \) in a ring where the ring size equals the number of nodes. The sequence of keys is generated with a random number generator \( g \):

\[
g = s + d(s, e_1) \left( \frac{d(s, e_{|E|})}{d(s, e_1)} \right)^R
\]

\( R \) is a uniform probability variable producing values in \([0, 1]\). The generator \( g \) is used when »the need arises«. The paper does not state when this happens, but it is reasonable to assume that a node which encounters long lookup paths could use the random number generator in Equation (2.3) to enhance its routing table.
2. Flexible Routing Tables

To restrict the size of a routing table, a bound $L$ is chosen which can be picked dynamically according to properties of the node such as its lifetime or performance. The choice of $L$ must be sensible to enable efficient routing, as we will see in the next subsection.

Entry Filtering and Best Routing Tables

As mentioned above, FRT uses a total order $\leq_{\text{ID}}$ to continuously enhance routing tables. The aim of the order is to converge routing tables towards the best possible routing tables. Which routing tables are better than others depends on the topology which should be generated. In the case of FRT-Chord, we consider an underlying Chord ring topology. In such a setting, best routing tables are expected to forward lookups using short paths and consequently, that a single hop in the path reduces the overlay distance to the endpoint significantly.

Consider the routing table $E$ of node $s$ and a routing table entry $e = s.forward(t)$ to which a lookup for key $t$ is forwarded. [NS11] defines the reduction ratio of a forwarding query: $d(e,t)/d(s,t)$. The reduction ratio is the remaining distance fraction after forwarding the request one step. Thus, the smaller the reduction the better the forwarding operation. The authors investigate the worst-case reduction for each routing table entry $e_i$. They state that »the reduction ratio of a forwarding node $e_i$ takes the worst-case value when the key identifier $t$ equals $e_i + 1«$. This non-intuitive conclusion follows from a simple observation: Given two routing table entries $e_i$ and $e_{i+1}$, only lookups for keys $k \in (e_i, e_{i+1}]$ will be forwarded to node $e_i$. We know from the definition of the routing table that $d(s,e_i) < d(s,e_{i+1})$, hence for $k$: $d(s,e_i) < d(s,k) < d(s,e_{i+1})$.

**Lemma 1** The worst-case reduction ratio for $e_i$ is encountered when the looked up key equals $e_{i+1}$

We provide a proof for Lemma 1 in Appendix A.1. Nagao and Shudo define the worst-case reduction ratio $r_i(E)$ for entry $e_i$.

**Definition 1** The worst-case reduction ratio for a routing table entry $e_i$ is defined as

$$r_i(E) = \frac{d(e_i,e_{i+1})}{d(s,e_{i+1})}$$

The worst-case reduction ratio is the base to define the term best routing table.

**Definition 2** (Definition 1 [NS11]) In FRT-Chord, the best routing table $\tilde{E} = \{\tilde{e}_i\}$ minimizes $\max_i\{r_i(E)\}$. 

26
The best routing table is specific to one node. Nodes $n_1, n_2$ with $n_1 \neq n_2$ will have different best routing tables, as their node identifiers differ. Nagao and Shudo proved further that the minimum worst-case reduction ratio of a node $s$ (and thus the worst-case reduction ratio of its best routing table) can be computed from its predecessor and successor using the following Lemma 2.

**Lemma 2 (Lemma 1 [NS11])**

$$r_i(\tilde{E}) = 1 - \left( \frac{d(s, \tilde{e}_1)}{d(s, \tilde{e}_{|\tilde{E}|})} \right)^{\frac{1}{|\tilde{E}|-1}} \quad (2.2)$$

The authors proof their first central Theorem 1 to show that path lengths in FRT-Chord are within $O(\log N)$ w.h.p., given that each node has the best routing table. To proof this, the authors assume that a routing table can contain at least $1 + 2\log N$ entries. Our evaluation in Section 5 will show that less routing table entries are sufficient for FRT-Chord to perform as well as Chord.

**Theorem 1 (Theorem 1 [NS11])** *With high probability (or under standard hardness assumptions), assuming that all nodes have the best routing table with $O(\log N)$ entries in an $N$-node network, path lengths are $O(\log N)$.*

In the proof of the theorem, inequality 2.3 is used, which holds with high probability. We mention this inequality as our implementation of FRT-Chord will make use of this bound for an optimization.

$$r_i(\tilde{E}) < 1 - \left( \frac{1}{N} \right)^{\frac{2}{|\tilde{E}|-1}}, \text{ with } i = 1, \ldots, |\tilde{E}| - 1 \quad (2.3)$$

From Equation 2.3, we know that a routing table $E$ cannot be a best routing table $\tilde{E}$ if the inequality does not hold.

**The total order $\leq_{ID}$** The best routing tables are able to route a lookup in $O(\log N)$ by minimizing the worst-case reduction ratios. Before we can explain how to refine routing tables to best routing tables, we have to introduce FRT-Chord’s total order $\leq_{ID}$. Nagao and Shudo define that order using $r_i(E)$ and state that »the order $\leq_{ID}$ represents an indicator of closeness to the best routing table«.
2. Flexible Routing Tables

**Definition 3** (Definition 2 [NS11]) Let \( \{ r_i(E) \} \) be the list arranged in descending order of \( \{ r_i(E) \} \).

\[ E \leq_{ID} F \iff \{ r_i(E) \} \leq_{dic} \{ r_i(F) \} . \]

In this definition, \( \leq_{dic} \) is a lexicographical order, namely

\[ \{ a_i \} <_{dic} \{ b_i \} \iff a_k < b_k \ (k = \min\{i | a_i \neq b_i \}) \]

\[ \{ a_i \} =_{dic} \{ b_i \} \iff a_i = b_i \]

\[ \{ a_i \} \leq_{dic} \iff (\{ a_i \} <_{dic} \{ b_i \}) \cup (\{ a_i \} =_{dic} \{ b_i \}) \]

The authors show that the order \( \leq_{ID} \) allows to evaluate if a routing table \( \tilde{E} \) is better than any other routing table \( E \).

**Theorem 2** (Theorem 2 [NS11]) Let \( E \) be a candidate for a routing table and \( \tilde{E} \) be a best routing table, \( \tilde{E} \leq_{ID} E \).

By Definition 3 and Theorem 2, we can deduce that \( \leq_{ID} \) indeed defines an »indicator of closeness« between a routing table and a best routing table.

**Entry Filtering** When the routing table \( E \)'s size exceeds the maximum size \( L \), entries have to be filtered. As the entry learning procedure adds one entry at a time, we will consider the case where at most one entry has to be filtered after a preceding learning operation. As demonstrated by Figure 2.4, FRT-Chord chooses the routing table \( E' \) from a set of possible tables \( \{ \{ e_i \}_{i \neq j} \}_{j=1,...,|E|} \), where each table is missing one entry that exists in \( E \). \( E' \) is picked such that \( E' \leq_{ID} E_i \), for all candidate routing tables \( E_i \).

In contrast to the prior description of FRT, FRT-Chord does not consider all possible routing tables when filtering, but only those which do not filter sticky entries. The routing correctness of FRT-Chord depends on sticky entries, hence they are not to be removed. Sticky entries are changed by ring maintenance operations, i.e. when sticky entries fail or nodes between \( s \) and its adjacent nodes join the ring. We will not consider sticky entries maintenance here.

Entry filtering is a three-steps process:

1. Substitute entries in the routing table \( E \) into a collection \( C \).
2. Remove sticky entries from the collection \( C \).
3. Select an entry from \( C \) to be removed from \( E \).  

28
2.3. FRT-Chord

Figure 2.4.: On entry filtering, a routing table is chosen from a set of candidate tables, where each candidate table removes one entry from the original entry. In this example, the candidate which removed node $n_3$ is supposed to be the optimal routing table of the three candidate tables and thus picked on filtering.

The first two steps of this procedure are self-explanatory. The filtering step 3 is the important part of the algorithm and has to comply with the total order $\leq_{ID}$. We will now discuss how Nagao and Shudo have defined Step 3 to comply with the total order $\leq_{ID}$. Step 3 is defined such that the resulting routing tables converge towards best routing tables.

Consider a routing table $E$ and function $S^E_i$ called canonical spacing, which is defined in Definition 4.

**Definition 4** (Def. 3 [NS11])

$$S^E_i = \log d(s,e_{i+1}) - \log d(s,e_i), \text{ with } i = 1,\ldots,|E| - 1$$ (2.4)

The canonical spacing of two routing table entries is the distance between the entries on a log-scale:

$$S^E_i = \log \frac{d(s,e_{i+1})}{d(s,e_i)} = \log d(s,e_{i+1}) - \log d(s,e_i)$$

With $S^E_i$ defined as above, we can define a property (Equation 19 in [NS11]) that a routing table entry $e_{i\ast}$ must fulfill in order to be filtered:

$$S^E_{i\ast - 1} + S^E_{i\ast} \leq S^E_{i\ast - 1} + S^E_i, \text{ with } i = 2,\ldots,|E| - 1$$ (2.5)

**Definition 5** The canonical neighbor spacing CNS$_i(E)$ of a routing table entry $e_i$ is defined as $S^E_{i - 1} + S^E_i$. 

29
2. Flexible Routing Tables

The canonical neighborhood spacing allows to define a dual problem to converge routing tables to best routing tables. Instead of minimizing the maximum of \( r_i(E) \), FRT-Chord maximizes the minimum of the canonical neighborhood spacing. We thus adapt the definition of best routing tables to this dual problem.

**Definition 6** In FRT-Chord, a routing table is also a best routing table if its canonical neighborhood spacing maximizes \( \min_i \{ \text{CNS}_i(E) \} \).

CNS computes the exponent \( \beta \) of the expression \( \alpha^\beta = \frac{d(s,e_{i+1})}{d(s,e_{i-1})} \). In a best routing table, \( \alpha \) and \( \beta \) will be constant for all routing table entries. Applying canonical neighborhood spacing to Chord allows us to compare FRT-Chord and Chord. If we consider three subsequent routing table entries \( e_{i-1}, e_i, e_{i+1} \) in Chord, the distance ratio \( \frac{d(s,e_{i+1})}{d(s,e_{i-1})} \) is expected to be 4 if the ring is filled completely with nodes. This is the case as the distance ratios \( \frac{d(s,e_{i+1})}{d(s,e_i)} \) and \( \frac{d(s,e_i)}{d(s,e_{i-1})} \) both equal 2.\(^4\) The canonical neighborhood spacing (or \( \beta \)) is thus 2. Computing the expected canonical neighborhood spacing is more complicated for FRT-Chord. First, we derive an equivalent representation of CNS:

\[
\text{CNS}_i(E) = - \log_2 (-r_{i-1}(E) + 1) - \log_2 (-r_{i+1}(E) + 1)
\]

By using Lemma 2, we can compute the value of the canonical neighborhood spacing of a best routing table \( \tilde{E} \).

\[
\text{CNS}_i(\tilde{E}) = \frac{2}{|\tilde{E}| - 1} \log_2 \frac{d(s, \tilde{e}_{|\tilde{E}|})}{d(s, \tilde{e}_2)} \quad (2.6)
\]

Consider a Chord ring \( I = [0, 127] \) with a total of 128 nodes in the ring. Using Equation 2.6, we can compute the canonical neighborhood spacing of a best routing table at node 0. We allow a maximum size of 18 routing table entries, including one successor and one predecessor node. For node 0, \( e_1 = 1 \) (as the node with identifier 1 is its successor), and \( e_{|E|} = 2^m - 1 = 127 \). Thus

\[
\text{CNS}_0(\tilde{E}) = \frac{2}{17} \log_2 127 \approx 0.78.
\]

Consequently for this FRT-Chord example, the canonical neighborhood spacing is not a natural number and is less than 1. In our evaluation of routing table entries in Section 5.1, we expect to see canonical neighborhood spacings similar to the one computed here when evaluating FRT-Chord with 16 non-sticky and two sticky entries on a ring of size 128.

\(^4\)Fingers halve distances. Consequently, the distance ratio of two fingers is expected to be 2.
Refining Routing Tables with Canonical Neighborhood Spacing From all removal candidates, we choose that entry $e_i*$ from $E$ whose adjacent routing table entries have a minimal canonical neighborhood spacing.\footnote{Note that $i$ does not equal the entry for the successor or predecessor. In general, $i$ can never equal the index of a sticky node. When the successor and predecessor lists are longer than one entry, $i$ must be adapted appropriately.} This is reasonable as we try to route asymptotically in $O(\log N)$. Only nodes splitting a large distance into two smaller distances of similar size are useful for the routing table. In essence, this results in a Chord-like topology. Chord achieves this by looking up keys with exponentially increasing distances.

Using Equation (2.5) to search for the entry to be filtered is efficient: only an ascendingly ordered list $M = \{S_E^{i-1} + S_E^i\}_{i=2,\ldots,|E|-1}$ has to be maintained and the index of $\min(M)$ determines the entry to be filtered. Only a constant number of $M$'s entries changes on entry learning or when a node is filtered. In order to show that Eq. (2.5) can be used to refine a routing table, Nagao and Shudo proof Theorem 3.

**Theorem 3** (Theorem 3 [NS11]) In FRT-Chord , let $E \setminus \{e_i*\}$ be a routing table filtered by removing $e_i*$. For any $e_i, i = 2,\ldots,|E| - 1,$

$$E \setminus \{e_i*\} \preceq_{ID} E \setminus \{e_i\}.$$

**Convergent Routing Tables**

Routing tables do not degrade from entry learning and filtering. Nagao and Shudo show this with Theorem 4.

**Theorem 4** (Theorem 4 [NS11]) In FRT-Chord, let $(E \cup \{e_{learn}\}) \setminus e_{filter}$ be a routing table after an entry learning process and a succeeding entry filtering process, where $e_{learn} \notin E$ is a learned entry and $e_{filter} \in (E \cup \{e_{learn}\})$ is a removed entry in these processes.

$$(E \cup \{e_{learn}\}) \setminus e_{filter} \preceq_{ID} E$$

The continuous refinement process of FRT-Chord allows to filter entries which have just been learned. [NS11] calls such tables *convergent routing tables*. Convergent routing tables will not be refined to best routing tables until newly learned nodes ‘survive’ the filtering process. Because of this, it is not clear from the proofs on best routing tables if routing in FRT-Chord
Flexible Routing Tables

Algorithm 7 FRT-Chord Entry Filtering

1. Substitute entries in the routing table $E$ into a collection $C$.
2. Remove sticky entries from the collection $C$.
3. Select the entry $e_i$ from $C$ that minimizes $S_{i-1}^E + S_i^E$.

is as efficient as promised, as tables can be stuck in states worse than best routing tables. FRT-Chord performs well when only convergent routing tables exist in the system, as Nagao and Shudo proof with the following Theorem 5.

Theorem 5 (Theorem 5 [NS11]) Assuming that all nodes have convergent routing tables with $O(\log N)$ entries in an $N$-node network, path lengths are $O(\log N)$ with high probability.

Similarly to Theorem 1, an inequality (Equality (29) in [NS11]) is used in the proof for Theorem 5, which will be used in our implementation. Consider Eq. (2.7).

$$r_j(E) < 1 - \left(\frac{1}{N}\right)^{\frac{4}{|i|-2}}, \text{with } j \in J$$

(2.7)

$J$ is the set of entry indices $i$ for routing table $E$ such that a node $n$ exists between $e_i$ and $e_{i+1}$, but $n$ is not contained in the routing table. As $E$ is assumed to be a convergent routing table, this implies that when such a node is learned it will be filtered again.

The three-steps algorithm for entry filtering can now be refined into its final form in Algorithm 7. As we will see in the following section, by changing the step where entries are substituted into $C$, other considerations than node identifiers can be used in FRT-Chord-like algorithms as well.

2.4. GFRT-Chord

We will see in this section that FRT simplifies extending an existing routing scheme with additional properties. In traditional DHT implementations, this is not a simple task, as routing tables are restricted by required node identifier combinations. In Chord, for example, only nodes with exponentially increasing distance to $s$ are considered for a routing table. When other properties such as latency requirements should be included in routing table construction, the additional properties are subordinate to the identifier combination requirement.
2.4. GFRT-Chord

In FRT, requirements can be mixed arbitrarily by adapting Algorithm 7. We will see how this can be done by discussing GFRT-Chord from [NS11]. In GFRT-Chord, routing table refinement does not only consider node identifiers, but also node grouping. Routing within a group of nodes should be fast and thus nodes from the same group should try to keep other nodes from the same group within their routing table. With this, GFRT-Chord can increase path locality. Quoting [Gho06, p. 10],

*Path locality means that queries for items which are available within an organization should not be routed to nodes outside the organization.*

The term *organization* is broad and can mean any partition of the set of nodes into subsets. It is reasonable to assume the partitioning to generate sets of nodes with small round-trip latency between each other. An example for this is a network of data centers, where nodes within one facility exhibit low latency when communicating with other nodes from the same facility.

Lookup latency can be reduced by minimizing the amount of inter-group hops, namely by avoiding to forward a request outside of a node group when the request will possibly end in the original group again after some time. We will use the term *group* for any set of nodes with a common set or properties, e.g. a group $G = \{n_i\}$ of nodes where the latency between $n_i, n_j \in G$ is below a constant bound.

GFRT-Chord differs to FRT-Chord only in the structure of its routing table and the entry filtering procedure. Guarantee of reachability and entry learning is implemented in the same way as for FRT-Chord.

In addition to the identifier and address pair already existing for a node in a FRT-Chord routing table, GFRT-Chord adds an additional field to an entry $e$, called $e.group$. This field indicates the group to which $e$ belongs and can be configured manually or automatically.

**Entry Filtering**  
GFRT-Chord aims to minimize inter-group hops as well as path lengths. To achieve this, a node $s$ tries to keep entries $e_i$ with $e_i.group = s.group$ in its routing table when entry filtering happens. GFRT-Chord also marks additional nodes as sticky entries, namely nodes from a *group successor list* and a *group predecessor*. We will now discuss variables build from routing table $E$ at node $s$ used in the entry filtering procedure. For this, Consider Figure 2.5, which shows a linear view on a node’s routing table. As in FRT-Chord, only non-sticky nodes can be filtered. Additionally, GFRT-Chord introduces a variable $e_{\alpha}$, which marks the first routing table entry belonging to $s.group$. $e_{\beta}$ accordingly marks the last routing table entry belonging to $s.group$. $e_{\alpha}$ and $e_{\beta}$ are considered to be sticky nodes. GFRT-Chord
partitions the routing table entries into two sets. $E_G$ is the set of all nodes from $s.group$, whereas $E_{\overline{G}}$ is the set of entries from other groups.

From all non-sticky nodes, only entries in $E_G$ should be considered for filtering. There could be cases where all entries $e \in E_{\overline{G}}$ are in front of $e_\alpha$. If that is the case, all nodes are considered for filtering, as GFRT-Chord could otherwise not guarantee that larger distances in the ring are well covered. This is due to the placement of the nodes from $s.group$. The distances between those entries do not necessarily increase exponentially. To mitigate this problem, the entry filtering of GFRT-Chord considers all non-sticky nodes for filtering if no entry from another group exists with a distance larger than $d(s,e_\alpha)$. Consequently, step 1 distinguishes the two cases. We summarize the variables used in the entry filtering process in Definition 7.

Definition 7 For the routing table $E$ at node $s$, the following variables are defined [NS11, p. 77].

- $E_G = \{ e \in E | e.group = s.group \}$, $E_{\overline{G}} = \{ e \in E | e.group \neq s.group \}$
- $e_\alpha$ such that $\forall e_i \in E_G: d(s,e_\alpha) \leq d(s,e_i)$
- $e_\beta$ such that $\forall e_i \in E_G: d(s,e_\beta) \geq d(s,e_i)$
- $E_{near} = \{ e \in E | d(s,e) < d(s,e_\alpha) \}$
- $E_{far} = \{ e \in E | d(s,e_\alpha) < d(s,e) < d(s,e_\beta) \}$
- $E_{leap} = E_{far} \cap E_{\overline{G}}$

GFRT-Chord refines Algorithm 7 using the variables from Definition 7 into Algorithm 8. The algorithm hints on the extensibility of FRT schemes: only step 1 differs to the original implementation in FRT-Chord. The actual filtering in step 3 remains unchanged. Algorithm 8 ensures that a routing table will never grow over the maximum size $L$ by filtering nodes from $s.group$ if no entry from another group is in the routing table.

---

6We deliberately use the term should, as nodes from the same group must be filtered if the routing table grows to large and no non-group node can be filtered. This is indicated by step 1 in Algorithm 8.
Algorithm 8 GFRT-Chord Entry Filtering

1. Substitute $E_{\bar{G}}$ into collection $C$ if $E_{\text{leap}} \neq \emptyset$, otherwise substitute $E$ into collection $C$.

2. Remove sticky entries from collection $C$.

3. Select the entry $e_{i_0}$ from $C$ that minimizes $S_{i_0}^E + S_{i_0}^E$.

Although GFRT-Chord prefers nodes from the same group over nodes from other groups, with high probability, only a small number of additional hops will be added to the path length. Nagao and Shudo proof this in Theorem 6.

**Theorem 6** (Theorem 10 in [NS11]) With high probability, assuming that all nodes have a convergent routing table with $O(\log N)$ entries in an $N$-node network, path lengths are $O(|G| + \log N)$ in a GFRT-Chord system composed of groups $G = \{G_i\}$.

Naturally, the number of groups $|G|$ should not exceed $O(\log N)$, or otherwise routing can take more than $O(\log N)$ hops.

2.5. Summary

FRT is a mechanism to generate routing tables lazily by learning and filtering entries as needed. It is a general concept, providing three basic primitives as a framework to built routing table algorithms. FRT-Chord and GFRT-Chord are instances of FRT. FRT-Chord is built on top of a Chord ring and aims to generate routing tables which minimize lookup path lengths. GFRT-Chord extends FRT-Chord by adding node group considerations to the filtering of routing table entries. In GFRT-Chord, a routing table entry is extended with a flag, which marks whether it belongs to the source node’s group.
3. Scalaris

In recent years, many new databases have been implemented which deviate from the relational database paradigm. Some applications do not need the consistency guarantees traditional databases provide but instead require more performance. Other applications produce data which can be modeled easier by using different data models than relations between tuples. The change in requirements over the last years gave rise to a number of new database implementation commonly summarized under the term *NoSQL*. ¹

One instance of this class of databases are the aforementioned *key-value stores*. Key-value stores are databases in which data is modeled as key-value pairs. Similar to a hash table, key-value stores provide simple methods to access data items. The database is backed by a data structure, which can, but does not need to be, a hash table. As noted before, we will implement FRT-Chord and GFRT-Chord in the distributed key-value store Scalaris [SSR08]. Scalaris is implemented using the Erlang programming language. The source code in the remainder of this thesis will be written using Erlang. We will not give an introduction to this language here, but instead refer readers to the Erlang website ² and [AWV93].

![Figure 3.1.: Layers in a Scalaris Node](image)

---

¹NoSQL is often seen as a misnomer and a product of marketing, as such databases do not necessarily avoid using SQL. [http://nosql-database.org/](http://nosql-database.org/) (accessed on 19. February 2013) defines the term as «non-relational, distributed, open-source and horizontally scalable» databases, which does not exclude using a SQL-dialect to access data stored in the database.

²[http://www.erlang.org/doc/getting_started/intro.html](http://www.erlang.org/doc/getting_started/intro.html), visited on 21. February 2013, contains a quick tutorial into the basics of the language. Reading sections 1 to 3 is sufficient to understand the source code of this thesis.
3. Scalaris

Scalaris is backed by a DHT and provides transactions over multiple keys, striving for scalability, performance and consistency. As shown in Figure 3.1, the implementation is split into three layers, each adding additional services to the layer below. The uppermost level transaction provides transactional access to items in the database and is the main entry point for users when working with Scalaris. Below the transaction layer lies the replication layer, which replicates key-value pairs over a set of nodes. The last layer implements the peer-to-peer functionality of Scalaris and is used to provide basic facilities for node communication and lookups. This peer-to-peer layer also contains Scalaris’ DHT implementation. Scalaris offers different DHTs, including Chord and Chord#. The implementation of the routing algorithms is placed within the peer-to-peer layer. Our implementation will be placed in that layer as well, and consequently we will describe it in more detail in Section 3.1.

Replication in Scalaris is symmetric: a value associated with key \( k \) is replicated to other keys \( k \oplus c \) shifted clockwise by an index \( c \). \( c \) is determined by the replication degree: if the replication degree is four, replicas of the item under key \( k \) are found at keys \( k \oplus 2^{m-2} \), \( k \oplus 2^{m-1} \) and \( k \oplus 2^{m-1} \oplus 2^{m-2} \).

The consistency of data stored under the replicated keys is implemented by a distributed consensus algorithm, which allows reading or writing the item even when a minority of the replicating nodes to fail.\(^3\)

3.1. Scalaris Peer-to-Peer Layer Software Architecture

In this section, we will discuss the technical aspects of Scalaris’ peer-to-peer layer. We explained earlier that Scalaris is organized hierarchically and that the base layer implements a DHT. We will focus our explanation on parts of the layer needed for the upcoming sections.

Scalaris is built on the Chord-like ring topology. The topology on top of that ring structure can be exchanged flexibly with arbitrary routing table implementations. The routing table implementation build on top of the ring can work differently than Chord. Standard Scalaris ships with a Chord implementation and a commercially available version of Scalaris also supports Chord# [Sch10].\(^4\) As our evaluation in Chapter 5 will focus on comparing FRT-Chord and GFRT-Chord with Chord, we do not consider Chord# further.

\(^3\)So with the default replication degree of four, one replicating node is allowed to fail.
3.1. Scalaris Peer-to-Peer Layer Software Architecture

The peer-to-peer layer consists of a set of interacting processes, of which three are relevant for this thesis: dht_node, rm_loop and rt_loop. A user-facing API api_raw_dht provides simplified access to the basic DHT operations, while hiding the layer’s implementation details.

In the remainder of this chapter we will first discuss the three processes mentioned above in more detail and then conclude the chapter with an example where we show how keys are looked up with the peer-to-peer layer. The example will use source code written in the Erlang programming language.

3.1.1. Message Hub: dht_node

The dht_node is the main message hub within a Scalaris node and acts as the local part of a DHT. It provides access to locally stored key-value pairs, and forwards requests for remote keys towards the responsible Scalaris nodes. Communication of Scalaris nodes happens between their respective dht_node: when a Scalaris node A wants to communicate with another Scalaris node B, A sends the dht_node at B a message. B’s dht_node must then either handle the message itself or forward the message to the corresponding process within the Scalaris node. Figure 3.2 depicts this simplified structure.

dht_nodes do not maintain the DHT structure themselves, but use the routing table provided to them for routing. We will discuss later how the rm_loop and rt_loop processes work together to maintain the DHT topology and inform the dht_node about changes to the topology. Section 3.1.4 will explain how a dht_node’s routing table is updated.
3. Scalaris

```erlang
rml_loop:subscribe(self(), rt_loop,
    fun rml_loop:subscribe_dneighbor_change_slide_filter/3,
    fun rt_loop:rml_send_update/4, inf)
```

Listing 3.1: The process `rml_loop` maintains the ring topology. Other processes can subscribe to receive information on changes to the topology. The process `rt_loop` subscribes to `rml_loop` in order to get updates about the Scalaris node’s neighborhood. `rml_loop` calls the function `rt_loop:rml_send_update/4` to inform `rt_loop` about changes.

### 3.1.2. Accessing the Peer-to-Peer Layer

Users can access the peer-to-peer layer in two ways: either by calling functions exported from the module `api_dht_raw`, or by sending messages to processes within the layer. Calls to the API functions are to be preferred when the internals of the peer-to-peer layer are not relevant. Higher-layer modules in Scalaris often avoid the indirectness of the API and send messages directly to avoid unnecessary function overhead.

The module `api_dht_raw` offers a simple interface for lookups in the DHT. It provides functions to retrieve the value stored with a key, generalized lookups for sending arbitrary messages to the lookup’s endpoint and also functions to execute ranged reads. Ranged reads are used to read all key-value pairs within an interval of keys [Sch10].

### 3.1.3. Scalaris Ring Maintenance

Nodes maintain the underlying ring topology using the process `rml_loop`. This process is responsible for maintaining a node’s view on its neighborhood and to inform local processes about changes to the neighborhood. A node’s neighborhood is the set of its predecessors, successors and the node itself. Currently two neighborhood maintenance implementations exist in Scalaris, of which a user can choose one at build time. The first is an implementation of Chord’s periodic stabilization routine called `rml_chord`. The second is an implementation of T-Man [JB05] called `rml_tman`.

Processes can subscribe to neighborhood updates by calling `rml_loop:subscribe/5`. This is depicted in the example in Listing 3.1. Upon a change to the neighborhood, `rml_loop` calls the callback function `rt_loop:rml_send_update/4`. This callback is provided with the old and new neighborhood, so that the routing table maintenance can change the routing table.
appropriately. The next section will explain in more detail how a change to the neighborhood will affect the routing table maintenance.

### 3.1.4. Routing Table Construction

The routing table is maintained by the `rt_loop`, using a helper module which implements the routing table behavior `rt_beh`. The module which implements the behavior does the actual routing table creation. The maintenance loop `rt_loop` generalizes mechanisms such as subscribing to `rm_loop`, but delegates the routing table creation to the helper module.

The currently active routing table implementation is configured using the Erlang macro `?RT`, whose value is the name of the module which should be used to create the routing tables. We will implement two modules providing the functions required by `rt_beh`, `rt_frtchord` for FRT-Chord and `rt_gfrtchord` for GFRT-Chord. We will describe the two modules in Section 4. Now we are going to discuss the main functions which are to be defined in such modules. The implementation Chapter 4 will refer to this functions later on.

### 3.1.5. The Routing Table Behavior

The routing table behavior specifies a small list of required functions which an implementation must export. Of those, we will look at a subset to implement the tasks of initializing routing tables, reacting to topology changes and computing the next hop of lookups. For a full list of needed functions, see the reference manual [SKS13].

The routing table behavior defines two different types of routing tables: external and internal routing tables. The external routing table is another representation of the internal routing table with a different structure. It is exported to the `dht_node`, which will pass that table to the routing algorithm. This is done to speed up computations for routing and avoids querying the `rt_loop` process on each routing operation. We will call the internal routing table routing table from now on, except where it is not clear from the context which type of table is meant.

---

5In Erlang, behaviors define interfaces for modules similar to interfaces in object oriented programming languages.
3. Scalaris

**Initialization** rt_loop calls ?RT:init/1 to initialize the routing table. The external routing table is initialized using ?RT:empty_ext/1. The functions are called with the current neighborhood and can thus initialize the routing tables with some knowledge of the current state of nearby nodes.

rt_loop keeps the routing table produced by ?RT:init/1 in its state and passes it to functions of the module ?RT as necessary. As explained before, the external routing table is passed to the dht_node. This does not happen automatically upon calling ?RT:empty_ext/1 but has to be done within the functions for the routing table maintenance, as explained in the following paragraph.

**Routing Table Maintenance** The routing table maintenance is split into a number of functions. ?RT:init_stabilize/1 is called periodically to start a stabilization round of the routing table. In rt_chord, this function starts the periodic stabilization round described in Section 1.3.2.

A routing table implementation has to react to changes to the neighborhood. When the set of neighbors or the identifier of the node changes, a call to ?RT:update/3 is issued, which is responsible for updating the routing table. This function call is triggered by the ring maintenance rm_loop as explained with the subscription example above in Section 3.1.4.

A routing table can also use failure detectors to detect dead nodes in the routing table. When a node is detected as dead, ?RT:filter_dead_node/2 is called which should remove the node from the routing table.

Changes to the routing table require the external routing table to be adapted and exported to the dht_node. This is done by the function ?RT:check/4, which checks if the routing table has changed and sends an updated external routing table to dht_node if that is the case.

Additionally, a routing table implementation can send and receive messages not handled by rt_loop. Messages sent to rt_loop which are not handled by that process are forwarded to the function ?RT:handle_custom_message/2. This mechanism can be used to implement communication between routing tables on different nodes. Functions in rt_beh implementations can send messages to the rt_loop process on other nodes, which then forwards the message to the respective handler in its rt_beh implementation.

**Routing** As described before, the dht_node is responsible for forwarding a lookup request to the next hop. The dht_node does not implement a method to calculate the next hop, as the algorithm depends on the routing table. Thus, rt_beh requires that an implementation must
Figure 3.3.: Lookup in a multi-node ring. The first dht_node and its successor are not responsible for the requested key. Thus, the request is forwarded to another node. That node’s successor is responsible for the key and receives a lookup_fin message. The message lookup_hops to the Monitor of dht_node3 is used to accumulate statistics on the lookup.

Amending Lookup Message Payloads A routing table implementation can add additional information to lookup requests, enabling to append data without interfering with the base routing algorithm. This is useful in cases where an implementation needs to observe single routing steps, e.g. for amending the routing table at the current hop with data from prior hops. rt_beh offers two functions to achieve this: rt_beh:wrap_message/3 and rt_beh:unwrap_message/2. On each forwarding hop, the function wrap_message/3 of the routing table implementation is called and the current state of the dht_node, as well as the lookup path length and message are passed in to the message. The function is expected to return a message, which is then forwarded along the lookup path. unwrap_message/2 is expected to strip the additional payload from the message upon entering the endpoint of the lookup. This makes the mechanism of amending information transparent to other parts of Scalaris.

The default routing table implementations in Scalaris implement the described functions as identity functions. FRT-Chord and GFRT-Chord in Scalaris use more implementations of the functions. We will discuss them in Section 4.

3.1.6. Key Lookups in Scalaris

In this section, we will describe key retrieval in Scalaris using an example.
3. Scalaris

Consider that a user of the peer-to-peer layer wants to retrieve a key-value pair \( \{ K, V \} \) where the key is a hashed string produced by \(?RT:hash_key(String)\). We assume that this key-value pair has been stored some time ago and the node which stores it is available. To retrieve the key, the user issues the call \( \text{dht}_\text{api}_\text{raw}:\text{unreliable}_\text{get}_\text{key}(K) \). This function uses \( \text{dht}_\text{api}_\text{raw}:\text{unreliable}_\text{lookup}/2 \) to send the node’s \( \text{dht}_\text{node} \) (see Figure 3.3) the message \( \text{lookup}_\text{aux} \) (described in Section 3.1.1). The \( \text{dht}_\text{node} \) handles a \( \text{lookup}_\text{aux} \) message by calling \( \text{dht}_\text{node}_\text{lookup}:\text{lookup}_\text{aux}/4 \). This function does the actual request routing: if the requested key is between the current node and its successor, the message tag is changed from \( \text{lookup}_\text{aux} \) to \( \text{lookup}_\text{fin} \) (see the message sent from \( \text{dht}_\text{node}_2 \) to \( \text{dht}_\text{node}_3 \) in Figure 3.3), and forwarded to its successor. If the key is not between those two nodes, the routing table is queried. For this, \( \text{dht}_\text{node}_\text{lookup}:\text{lookup}_\text{aux}/4 \) calls \(?RT:next_hop/2\). In case of Chord-like algorithms, \(?RT:next_hop/2\) implements the \text{find_successor} method: it forwards a request to the farthest known node whose identifier is smaller than the requested key.

On receiving \( \text{lookup}_\text{fin} \), the \( \text{dht}_\text{node} \) validates that is indeed responsible for the key. If that is not the case, it restarts the routing. Otherwise it inspects the payload. The payload generated by \( \text{api}_\text{dht}_\text{raw}:\text{unreliable}_\text{get}_\text{key}/1 \) is \( \{ \text{get_key}, \text{From}, K \} \), where \text{From} is a globally unique identifier for the process which requested the key. Such an identifier is a combination of an IP address and an Erlang process identifier. The \( \text{dht}_\text{node} \) can now lookup the key in its local database and send the key-value pair to the provided identifier \text{From}. This concludes the lookup.

A lookup request does not necessarily end with a reply to the requesting process. The payload determines if a reply is necessary. In the case of key retrieval, it is obvious that a reply has to be done. Other examples such as gossiping protocols might not need the reply and thus it depends on the implementation of a protocol if a reply is issued. When no reply is issued, the requesting process has no mechanism to check that the lookup succeeded.
4. Implementation of Flexible Routing Tables

We will now discuss the implementation of FRT-Chord and GFRT-Chord. The implementation consists of two modules implementing the `rt_beh` described in Chapter 3.1. The `rt_loop` process uses one of the two modules for the routing table maintenance, as was also explained in Chapter 3.1.

Flexible routing tables can share code and type definitions. As the actual implementation of FRT-Chord and GFRT-Chord only differs in annotation of routing table entries – GFRT-Chord adds a flag to routing table entries, indicating if the entry belongs to the same group as the source node – we provide a common header `rt_frt_common.hrl` file which exports functions and types common to both FRT-Chord and GFRT-Chord. We will thus first discuss the shared routing table structure as well as the generalized entry learning and filtering algorithms. Later, we will see how one can build upon the general algorithms to implement more specialized routing table construction algorithms.

```
rt_beh
   includes
      rt_frt_common
      rt_frtchord
      rt_gfrtchord
```

Figure 4.1.: FRT-Chord and GFRT-Chord share common code in the header file `rt_frt_common.hrl`. This file exports the functions required by the routing table behavior `rt_beh`. 

45
4. Implementation of Flexible Routing Tables

As explained before, Scalaris distinguishes between internal and external routing tables. External routing tables are cached in `dht_nodes`, allowing the `dht_node` to call `?RT:next_hop/2` without delegating the computation to the `rt_loop` process. The internal routing table is kept in the state of the `rt_loop` process and represents the actual state of the routing table. The external routing table is a mere mapping of the internal routing table and updated when needed by calling `?RT:check/4`.

The structure of the internal FRT routing table is defined with the Erlang record `#rt_t` (also referred to as `rt_t()`) in Listing 4.1. It consists of the source node’s identifier `source` and a tree of routing table entries called `nodes`. The tree `nodes` maps ring identifiers to routing table entries.

An external routing table is a tree mapping an entry’s node identifier to its address. The source node is filtered from the external routing table, as it is not needed for routing lookup requests. Listing 4.3 displays the algorithm to convert an internal routing table to an external routing table.

**Routing Table Entry** In Section 2.3, we described a routing table entry in FRT-Chord as a pair consisting of the entry’s ring identifier and its address. Listing 4.2 is the actual representation

---

```erlang
%% A routing table consists of the source node id and a set of entries
-record(rt_t, { source = undefined :: key_t() | undefined, nodes = gb_trees:empty() :: gb_tree() })
-type(rt_t() :: #rt_t{}).
```

**Listing 4.1: Internal routing table structure**

```erlang
-record(rt_entry, { node :: node:node_type(), type :: entry_type(), %% normal, sticky or source? adjacent_fingers = {undefined, undefined} :: {key_t() | undefined, key_t() | undefined}, custom = undefined :: custom_info() })
-type(rt_entry() :: #rt_entry{}).
```

**Listing 4.2: The structure of a routing table entry.**

**4.1. Routing Table Structure**

As explained before, Scalaris distinguishes between internal and external routing tables. External routing tables are cached in `dht_nodes`, allowing the `dht_node` to call `?RT:next_hop/2` without delegating the computation to the `rt_loop` process. The internal routing table is kept in the state of the `rt_loop` process and represents the actual state of the routing table. The external routing table is a mere mapping of the internal routing table and updated when needed by calling `?RT:check/4`.

The structure of the internal FRT routing table is defined with the Erlang record `#rt_t` (also referred to as `rt_t()`) in Listing 4.1. It consists of the source node’s identifier `source` and a tree of routing table entries called `nodes`. The tree `nodes` maps ring identifiers to routing table entries.

An external routing table is a tree mapping an entry’s node identifier to its address. The source node is filtered from the external routing table, as it is not needed for routing lookup requests. Listing 4.3 displays the algorithm to convert an internal routing table to an external routing table.

---

```erlang
%% A routing table consists of the source node id and a set of entries
-record(rt_t, { source = undefined :: key_t() | undefined, nodes = gb_trees:empty() :: gb_tree() })
-type(rt_t() :: #rt_t{}).
```

**Listing 4.1: Internal routing table structure**

```erlang
-record(rt_entry, { node :: node:node_type(), type :: entry_type(), %% normal, sticky or source? adjacent_fingers = {undefined, undefined} :: {key_t() | undefined, key_t() | undefined}, custom = undefined :: custom_info() })
-type(rt_entry() :: #rt_entry{}).
```

**Listing 4.2: The structure of a routing table entry.**

---

**4.1. Routing Table Structure**

As explained before, Scalaris distinguishes between internal and external routing tables. External routing tables are cached in `dht_nodes`, allowing the `dht_node` to call `?RT:next_hop/2` without delegating the computation to the `rt_loop` process. The internal routing table is kept in the state of the `rt_loop` process and represents the actual state of the routing table. The external routing table is a mere mapping of the internal routing table and updated when needed by calling `?RT:check/4`.

The structure of the internal FRT routing table is defined with the Erlang record `#rt_t` (also referred to as `rt_t()`) in Listing 4.1. It consists of the source node’s identifier `source` and a tree of routing table entries called `nodes`. The tree `nodes` maps ring identifiers to routing table entries.

An external routing table is a tree mapping an entry’s node identifier to its address. The source node is filtered from the external routing table, as it is not needed for routing lookup requests. Listing 4.3 displays the algorithm to convert an internal routing table to an external routing table.

---

**Routing Table Entry** In Section 2.3, we described a routing table entry in FRT-Chord as a pair consisting of the entry’s ring identifier and its address. Listing 4.2 is the actual representation
4.1. Routing Table Structure

Listing 4.3: Converting the internal routing table to an external routing table for the dht_node.

The algorithm iterates over all non-source nodes and, for each such node, adds a mapping to a key-value storing datastructure.

in our implementation, which we will refer to as rt_entry(), and which consists of additional information. The entry node contains the aforementioned tuple of identifier and address. To mark a node as being the source node, a sticky node or a normal node, the entry type is used. Sticky nodes are neighbors of the source node. Only normal nodes will are considered in the entry filtering algorithm later. The pair adjacent_fingers is used for topology maintenance and contains the ring identifiers of the routing table entries adjacent to the entry. For example, if the nodes with identifiers 0, 1000, and 2000 are contained in the routing table and there are no other nodes between them, the adjacent fingers of the entry with the identifier 1000 is the tuple \( \{0, 2000\} \). We use this structure to build the list of canonical neighborhood spacings (see Section 2.3) lazily: the original paper on FRT-Chord [NS11] proposed to keep the list of canonical neighborhood spacings persistently with the routing table. For ease of implementation, we decided against that and compute this list on each entry filtering operation. This is reasonable, as we can maintain the adjacent fingers easily and can compute the list of canonical neighborhood spacings comfortably with them. Listing 4.4 displays the algorithm used to compute the list. The function call spacing(Node, RT) computes \( S^E_i \) for an entry \( i \) and takes constant time. Computing predecessor_node is more costly, as it retrieves the predecessor’s routing table entry from the routing table, which is a \( O(\log N) \) operation. This is fast enough for routing tables, as the size of a routing table will be bound by a constant \( L \), and so this is a constant time operation as well.

The custom field in a routing table entry is an untyped entry which can be used to specialize routing table entries. FRT-Chord does not make any special use of this entry, as the filtering mechanism only depends on the node identifiers. GFRT-Chord, on the other hand, will use
4. Implementation of Flexible Routing Tables

Listing 4.4: Calculating the list of canonical neighborhood spacings is fast. The algorithm iterates over all routing table entries and generates a list entry for each one. A list entry is generated by adding the distance to the node’s predecessor to the distance of the node’s successor.

```
%% Generate a list consisting of SE_i + SE_{i+1}. Each entry i is
%% built from one routing table entry e_i

[ spacing(Entry, RT)  %% compute SE_i
  + spacing(predecessor_node(RT,Entry), RT)  %% compute SE_{i+1}
  || Entry <- AllEntries  %% for all e_i
]
```

Figure 4.2.: Our FRT implementation uses a number of messages to refine routing tables. Nodes communicate with each other and exchange information to enhance their respective routing tables. Here, dashed arrows indicate replies to preceding request messages. The message `rt_learn_node` can be either a request or an answer: When send from the node to itself, it is a request to learn a node; when send from another node, it is a reply to a `rt_get_node` request.

this entry to mark entries as being in the source node’s group. The custom field is set by a call to the function `rt_entry_info/4`, which receives the identifier-address pair, type and adjacent neighbors of the node to be added. Implementations which include the header file `rt_frt_common.hrl` have to implement this function as part of the contract.

Each routing table entry is associated with a failure detector, which checks if the corresponding node is alive. The routing table receives a message when it decides that a node has failed. `rt_loop` then calls the function `filter_dead_node/2`, which removes the node and returns a new routing table. No additional checks to ensure entry validity are used.
4.2. Generalized Entry Learning and Filtering

We will now discuss the shared code for entry learning and filtering. Further, we will explain which messages are sent between Scalaris nodes due to flexible routing table construction. In the following segment, we will use the term routing table and \texttt{rt\_loop} interchangeably. Hence, if we write that the routing table sends a message, we mean that the process maintaining the routing table called a function which caused a message to be sent to another process. If we write that the routing table accepts a message, we mean that the \texttt{rt\_loop} process forwards a message to our implementation of \texttt{rt\_beh}.

4.2.1. Communication between Routing Tables

The flexible routing tables use a simple protocol to exchange routing table entries. With this protocol we implement the three operations for entry learning described in Section 2.3. Figure 4.2 depicts the messages which are send between routing tables. On initialization, a node requests the routing table of its successor using the message \texttt{get\_rt}. The successor answers the request with the message \texttt{get\_rt\_reply} and its routing table. A routing table can also query another routing table for its identifier-address pair using the message \texttt{rt\_get\_node}, which is used when routing table \texttt{A} knows the IP address of routing table \texttt{B} but also needs \texttt{B}'s ring identifier. This is useful for active learning lookups: when the lookup carries the message \texttt{rt\_get\_node} to the endpoint \texttt{E} of the request, \texttt{E} sends the message \texttt{rt\_learn\_node} and its identifier-address pair to the requestor, which can then add \texttt{E} to its routing table. We will discuss this in more detail in Section 4.2.2, where we will introduce entry learning in more detail.

As discussed before, routing tables periodically perform active learning lookups. Each interval, a routing table sends itself the message \texttt{trigger\_random\_lookup}, triggering a key lookup and scheduling the next trigger message.

Routing tables also send itself messages (or rather: the \texttt{dht\_node} sends a message to the routing table) on forwarding lookup requests. The message \texttt{rt\_learn\_node} in Figure 4.2 which loops back to the node carries the identifier-address pair of the node which requested the lookup. By this, the routing table is refined with every lookup which passes through the node.
4. Implementation of Flexible Routing Tables

```
init(Neighbors) ->
  % if activated, schedule the first active learning lookup
  case config:read(rt_frtchord_al) of
    true -> comm:send_local(self(), {trigger_random_lookup});
    false -> ok
  end,
  % ask the successor node for its routing table
  Msg = {send_to_group_member, routing_table, {get_rt, comm:this()}},
  comm:send(node:pidX(nodelist:succ(Neighbors)), Msg),
  % initial table, consisting of the source node and its neighborhood
  update_entries(Neighbors, add_source_entry(nodelist:node(Neighbors),
    #rt_t{}}).
Listing 4.5: Initialization of a routing table.
```

4.2.2. Initialization and Entry Learning

Entry learning follows the description from [NS11] closely. We will go through the three mechanisms step by step.

**Learning on Routing Table Initialization and Merging the Successor’s Routing Table** In Section 3.1.4, we explained that a routing table must implement and export a function `init/1`, which accepts the node’s neighborhood and constructs an initial routing table. Figure 4.5 depicts the initialization function for the generalized FRT implementation. The function schedules the first active learning lookup, requests the successor’s routing table, and constructs an initial routing table consisting of the nodes from the `Neighborhood`.

The node’s successor answers requests for its routing table with the message `get_rt_reply`, as discussed in Section 4.1. The routing table’s `handle_custom_message/2` handler accepts this message and merges the received table. Listing 4.5 depicts this clause of the message handler. We do not require that the received routing table came from the successor node. Any routing table can be merged, as entry filtering will ensure that only useful entries survive the merge.

Merging tables is straightforward: entries from the received routing table are added consecutively and, after adding a node, entry filtering is performed when the routing table is full. We assume that each entry can be added as a normal, filterable entry. This is reasonable, as the ring maintenance process `rm_loop` will later inform the routing table about changes to the neighborhood and we can then transform normal entries to sticky entries if needed.
4.2. Generalized Entry Learning and Filtering

handle_custom_message({get_rt_reply, RT}, State) ->
  %% Merge the routing tables. Note: We do not care if this
  %% message is not from our current successor. We only
  %% have to ensure that the merged entries are valid.
  LocalRT = rt_loop:get_rt(State),
  NewRT = case LocalRT /=/= RT of
    true ->
      % Iterate over the routing table which was received
      % - add each entry from the other RT that does not exist yet
      % - entries are added as normal entries
      util:gb_trees_foldl(fun(Key, Entry, Acc) ->
        case entry_exists(Key, Acc) of
          true -> Acc;
          false -> add_normal_entry(rt_entry_node(Entry), Acc)
        end
      end,
      LocalRT, get_rt_tree(RT));
    false -> LocalRT
  end,
  rt_loop:set_rt(State, NewRT);

Listing 4.6: Merging the successor’s routing table.

Active Learning Lookups  The periodical active learning lookups are triggered by the
routing table, which sends itself delayed trigger_random_lookup messages. Listing 4.7
depicts the message handler for those messages. The message handler sends a delayed
trigger_random_lookup message to itself and performs the lookup. For the lookup, Key is
generated from the random number generator discussed in Section 2.3. K is looked up by a
call to api_dht_raw:unreliable_lookup(Key). The lookup message carries the message
rt_get_node to the endpoint of the lookup, which replies with a rt_learn_node message
and, as described above, its identifier-address pair. The requesting routing table can then add
this node to its routing table.

Learning on Forwarding  The original proposal for FRT [NS11] includes entry learning
on node communication as one mechanism to refine a routing table. The authors did not
specify further if entry learning should happen for each message sent between nodes, or if
restrictions would be useful. We chose to learn new entries only during lookup routing, using the
functions rt_beh:wrap_message/3 and rt_beh:unwrap_message/2, which we introduced
4. Implementation of Flexible Routing Tables

```plaintext
handle_custom_message([{trigger_random_lookup}], State) ->
  %% schedule the next random lookup by sending a message to itself
  Interval = config:read(rt_frtchord_al_interval),
  msg_delay:send_local(Interval, self(), {trigger_random_lookup}),

  %% compute the random key
  RT = rt_loop:get_rt(State),
  SourceNode = ?RT:get_source_node(RT),
  SourceNodeId = entry_nodeid(SourceNode),
  {PredId, SuccId} = adjacent_fingers(SourceNode),
  Key = get_random_key_from_generator(SourceNodeId, PredId, SuccId),

  %% execute the lookup
  api_dht_raw:unreliable_lookup(Key, {?send_to_group_member, routing_table,
                                      {rt_get_node, comm:this()}}, State).
```

Listing 4.7: Active learning lookups

in Section 3.1.5. Routing tables will learn of the lookup’s issuer on forwarding, and the issuer
will receive a reply from one of the nodes in cases where a node decides that the issuer’s routing
table should know about another node. This is less intrusive into Scalaris’ existing source
code than trying to learn from every communication path between two nodes. Learning new
entries on forwarding keeps changes in Scalaris local to the routing table maintenance but is
good enough to allow efficient routing table refinement. Furthermore, communication between
nodes outside of lookup requests already either depends on routing tables, so a direct link exists
between the nodes in the topology, or makes use of ring maintenance, which is reflected in the
routing tables as well.

Routing tables try to learn of a node each time they forward a lookup request. The issuer
of the request thus appends its node information to the lookup request, so that intermediate
routing tables can learn of it. This is done in the call to the first clause of \texttt{wrap_message/3}
(Listing 4.8), where the hops parameter is set to 0. The issuer of the lookup wraps the lookup
request message in a tagged tuple and adds its identifier-address description to it. The identifier-
address pair for the node is extracted from the state carried by the \texttt{dht_node}. Intermediate
nodes will call the second clause of \texttt{wrap_message/3}, extracting the issuer’s identifier-address
pair from the message and adding it to their routing table. They add the issuer to their routing
table by sending the node information with the message \texttt{rt_learn_node} to the routing table
process.
4.2. Generalized Entry Learning and Filtering

```prolog
%% The issuer wraps the lookup request with additional information
wrap_message(Msg, State, 0) ->
    {"wrapped", dht_node_state:get(State, node), Msg};

%% Subsequent nodes learn the issuer from the lookup request
wrap_message({'$wrapped', Issuer, _} = Msg, _State, _Hops) ->
    % learn a node when forwarding its request
    learn_on_forward(Issuer),
    Msg.

%% To learn a new node, a message is sent to the routing table
learn_on_forward(Issuer) ->
    comm:send_local(self(), {?send_to_group_member, routing_table,
        {rt_learn_node, Issuer}}).
```

Listing 4.8: Learning a node on request forwarding.

```prolog
%% The endpoint gets the original lookup content and sends its id to the originator
unwrap_message({'$wrapped', Issuer, UnwrappedMessage}, State) ->
    comm:send(node:pidX(Issuer),
        {?send_to_group_member, routing_table,
            {rt_learn_node, dht_node_state:get(State, node)}},
        UnwrappedMessage.
```

Listing 4.9: Unwrapping lookup requests.

As the message payload of the lookup is forwarded within the endpoint node, the additional wrapping around the message has to be deleted before the message is forwarded. This is done with the `unwrap/2` (Listing 4.9) function, which returns the unwrapped message. It also sends a message to the issuer, so that this node may learn of the endpoint, in order to refine the routing table. This is a crude approach for routing table refinement and we will discuss how to avoid sending unnecessary messages in the remaining part of this section.

**Optimized Issuer’s Routing Table Refinement**

In the last paragraph, we introduced a mechanism to refine the routing table of a lookup issuer by sending it the identifier-address pair of the endpoint. So for each lookup request, one extra message is sent to the issuer of the lookup. Here, we will explain under which circumstances
4. Implementation of Flexible Routing Tables

Figure 4.3.: To optimize refinement of a lookup issuer’s routing table and to reduce the number of messages sent on lookups, we consider lookup paths longer than two hops. In this picture, we display three nodes $s, n_0$ and $n_1$, which are the prefix of the lookup path $(s, n_0, n_1, \ldots)$.

such a reply can be avoided, and which intermediate node should be learned of by the issuer in cases where the lookup request path was suboptimal. The motivation for the optimization is based on two observations: Firstly, the simple approach discussed before sends a reply for each lookup, even if the routing table of the issuer is convergent. Secondly, the routing table of a node should be refined by first adding nodes nearby to the node. This is due to the fact that the source node only forwards to one of its known nodes, thus the quality of the last hop in the lookup path is not as important for the source node as the quality of the first hop. We will optimize these two points by using FRT-Chord’s reduction ratio.

Our optimization considers two distinct cases of the lookup path’s length: if the path is one or two hops long, we do not send any message to the issuer. This is reasonable, as the issuer already knows the endpoint (one hop lookup path), or the second node in the lookup path knows the endpoint. In the latter case, the function next_hop/2 would return the second node for keys in the range of the endpoint, so the endpoint does not refine the routing table considerably. Conversely, if the path is longer than two hops, the first node of the path decides if the issuer should learn the second node of the path. In that case, consider a setting of three nodes, $s, n_1$ and $n_2$ (see Figure 4.3), where the node $s$ requests a key and forwards the request to its routing table entry $n_1$. $n_1$ forwards the request to its entry $n_2$. When the request reaches node $n_1$, this node can determine the reduction ratio $\frac{d(n_1, n_2)}{d(s, n_2)}$. From Section 2.3, we know that for the
4.2. Generalized Entry Learning and Filtering

reduction ratio \( r_i(\tilde{E}) \) of best routing tables \( \tilde{E} \) (Eq. 2.3):

\[
r_i(\tilde{E}) < 1 - \left( \frac{1}{N} \right)^{\frac{2}{|\tilde{E}| - 1}}.
\]

Hence, we can consider a forwarding operation as ‘not good enough’, if the reduction ratio is bigger than \( 1 - \left( \frac{1}{N} \right)^{\frac{2}{|\tilde{E}| - 1}} \) and hence the first hop was not far enough towards the endpoint of the lookup. The bound above obviously applies only when best routing tables are possible. This is not necessarily the case, so we have to accommodate convergent routing tables as well. For this, we can substitute the bound above with the bound given for convergent routing tables (Eq. 2.7):

\[
r_j(E) < 1 - \left( \frac{1}{N} \right)^{\frac{4}{4 |E| - 2}}, \forall j \in J.
\]

Eq. 2.7 only applies for \( j \in J \).\(^1\) This is sufficient for our optimization: for entries \( i \notin J \), there is no node between \( i \) and \( i + 1 \), thus the lookup finishes after another step and we will not send a message to the issuer.

Both bounds make use of the number of nodes \( N \). In Scalaris (and any distributed system of considerable size), we can only estimate \( N \). This is good enough for our purposes and we will explain later how Scalaris estimates the number of nodes. We cache ring size estimate in the internal and external routing table and update it periodically.

We expect Eq. 2.3 to cause more messages being sent than with Eq. 2.7, as

\[
1 - \left( \frac{1}{N} \right)^{\frac{2}{|\tilde{E}| - 1}} < 1 - \left( \frac{1}{N} \right)^{\frac{4}{4 |E| - 2}}.
\]

Hence, lookup paths which do not obey the bound Eq.2.3 might still obey Eq. 2.7.

**Implementing the Optimization**  As the optimization does not need to consider the last hop, we reduce the unwrap_message/2 function to the one given in Listing 4.10.

Replies to the issuer of the lookup are no longer sent by the endpoint, but by an intermediate hop. This is implemented in wrap_message. We need to add the key as an additional parameter to wrap_message. The important parts, excluding glue code, of our implementation are given

\(^1\)Note again that \( J \) is the set of routing table entry indices \( i \) such that a node exists between \( e_i \) and \( e_{i+1} \).
4. Implementation of Flexible Routing Tables

unwrap_message({'$wrapped', _Issuer, UnwrappedMessage}, _State) -> UnwrappedMessage.

Listing 4.10: Unwrapping is not needed anymore with our optimization.

MyId = dht_node_state:get(State, node_id),
SenderId = node:id(Issuer),
NextHop = next_hop_(State, Key),
SendMsg = case external_rt_get_ring_size(RoutingTable) of
    unknown -> true;
  RingSize ->
    %% Compute the reduction ratio of the hop
    FirstDist = get_range(SenderId, MyId), % d(s, n1)
    TotalDist = get_range(SenderId, node:id(NextHop)), % d(s, n2)
    %% Is the reduction ratio worse than bound?
    (1 - FirstDist / TotalDist) > reduction_ratio()
end

Listing 4.11: Optimized routing table refinement algorithm

in Listing 4.11. The issuer of the lookup wraps the message in the same way as before. The node of the first hop has to decide if the lookup message did not travel a long enough distance. The code in Listing 4.11 is therefore only run if the request travelled one hop and travels at least two more hops. As discussed before, this is only done when the first hop is not the predecessor of the node responsible for the key. If it is the predecessor, nothing has to be done, as the reduction ratio is not useful for that case.

Relying on Figure 4.3, consider that \( n_1 \) is not the predecessor of the key and thus forwards a lookup_aux message to \( n_2 \). \( n_1 \) has to decide if the first hop from \( s \) to \( n_1 \) was long enough. Hence it must calculate the identifier-address pair \( \text{NextHop} = \text{next_hop_}(\text{State}, \text{Key}) \) and the distances \( \text{FirstDist} (= d(s, n_1)) \) and \( \text{TotalDist} (= d(s, n_2)) \). With that, we obtain the hop’s reduction ratio by computing \( 1 - \frac{d(s, n_1)}{d(s, n_2)} \), as \( 1 = \frac{d(s, n_1)}{d(s, n_2)} + \frac{d(n_1, n_2)}{d(s, n_2)} \).

As stated above, we have two options to check if the reduction ratio was good enough: by comparing it to the reduction ratio of best routing tables or by comparing it to the reduction ratio of convergent routing tables. In our implementation, the choice is defined by a configuration parameter, which is retrieved with :config:read(rt_frt_reduction_ratio_strategy). We wrap this in the function reduction_ratio/0.

A message is send to the issuer of the lookup only when the actual reduction ratio is worse than
4.2. Generalized Entry Learning and Filtering

```prolog
1  best_rt_reduction_ratio(RingSize) ->
2     1 - math:pow(1 / RingSize, 2 / (maximum_entries() - 1)).
3  convergent_rt_reduction_ratio(RingSize) ->
4     1 - math:pow(1 / RingSize, 4 / (maximum_entries() - 2)).
```

Listing 4.12: The reduction ratio bounds are computed as described in Section 2.3

the chosen bound. The decision is stored in the boolean variable SendMsg. If SendMsg is true, the message \{rt_learn_node, NextHop\} will be sent to the routing table of the issuing node. As in the naïve implementation of wrap_message, nodes forwarding a request will always learn the issuing node. Thus after deciding if the issuer should be informed about the subsequent hop, nodes call learn_on_forward(Issuer), which causes the message rt_learn_node to be sent to the routing table process of the node.

For wrap_message, the clause to handle requests which traveled more than one hop remains to be discussed. Here, the only action of the node is to learn the issuer of the lookup. The message itself remains untouched.

The implementation of the reduction ratio bounds is given in Listing 4.12. The functions best_rt_reduction_ratio/1 and convergent_rt_reduction_ratio/1 both require the ring size estimate as an argument. In the next paragraph, we will explain how Scalaris computes the estimate.

Retrieving a Ring Size Estimate  To estimate the ring size, Scalaris makes use of a gossip protocol based on the work of Jelasity, Montresor and Babaoglu [JMB05]. Periodically, a node \(a\) chooses another node \(b\) at random and sends \(b\) its state. \(b\) replies with its state and incorporates \(a\)’s state. Upon receiving \(b\)’s answer, \(a\) updates its local state as well. The gossip algorithm is based on rounds, where a new round is started periodically by the node responsible for the key \(?RT\):hash_key("0"). On each new round, local state is flushed.

In Scalaris, the process gossip implements this protocol and accumulates averages on load statistics and the size of the ring.

For the purposes of our optimization, we consider the ring size estimate size_kr. This estimate is based on the average number of identifiers for which nodes are responsible (their respective key range). Given an average key range length of \(a\), we can estimate size_kr directly:

\[
\text{size}_\text{kr} = \frac{N_{\text{max}}}{a}, N_{\text{max}} := |I|
\]
4. Implementation of Flexible Routing Tables

This estimate is reasonable under the assumption that all nodes have key ranges of the same length. The average key range estimate \( a \) is aggregated by the gossip algorithm. Given two nodes \( n_1, n_2 \) and their local key range averages \( a_1 \) and \( a_2 \), node \( n_1 \) updates its local average to a new average \( a'_1 \) upon receiving the state of \( n_2 \) by calculating 
\[
  a'_1 = \frac{a_1 + a_2}{2}.
\]
The initial value for the average key range is unknown, until the ring maintenance process informs gossip about the key range of the node.

A Note on the Optimization and GFRT-Chord

The reduction ratio bounds above apply only to FRT-Chord. The formal analysis of GFRT-Chord in [NS11] is only partially based on reduction ratios. Informally, the lookup performance proof ([NS11, Theorem 10]) is split into cases of hops leaving a group and cases of hops remaining within the group. For the latter, the reduction ratio can be bound with
\[
  r_j(E) < 1 - \left( \frac{1}{N} \right)^{\frac{4}{|\pi_j| - 2}}.
\] (4.1)

In the case where the lookup request leaves a group, the reduction ratio can be worse than Eq. 4.1.

In our implementation, we will use the reduction ratio bound for convergent routing tables from FRT-Chord and justify our decision with the following inequality.
\[
  1 - \left( \frac{1}{N} \right)^{\frac{4}{|\pi_j| - 2}} < 1 - \left( \frac{1}{N} \right)^{\frac{4}{|\pi_j| - 2}} \quad \text{FRT-Chord’s bound}
\]

From this, we know that the bound of FRT-Chord might overestimate the reduction ratio and hence, the optimized algorithm might not send learning messages when that should be done. Also, the optimization might underestimate the reduction ratio when a hop leaves a group, thus the optimization might send too many messages. The latter does not contradict the optimization: even though too many message are sent, the number of messages will still be at most the number of lookups issued. Hence, the algorithm does not perform worse than the naïve implementation. In the case where the bound overestimates the reduction ratio, we expect the optimization to perform well enough for fast routing table refinement. Our evaluation in Section 5.2 shows that this approach outperforms the naïve approach.
4.3. FRT-Chord and GFRT-Chord Implementation

Listing 4.13: Entry filtering computes the ordered set \( \{S_{E_{i-1}} + S^E_i\}_i \) for all allowed entries \( i \).

4.2.3. Entry Filtering

After discussing our entry learning implementation, only entry filtering remains to be explained. To filter one entry from a routing table, the function `entry_filtering/1`, which computes the list of filterable nodes with a call to `allowed_nodes/1` and then delegates filtering to `entry_filtering/2` (Listing 4.13), `allowed_nodes/1` is the function where the FRT-Chord and GFRT-Chord implementation will differ, as we will see in the upcoming sections. The function `entry_filtering/2` is split into two clauses. If the list of filterable nodes is empty, no node can be filtered. It is up to the user of the function to decide if this should be handled as an error. When at least one entry is filterable, the algorithm described in Section 2.3 is applied (see Listing 4.13): we compute the list of canonical neighborhood spacings `Spacings`, order it in ascending order and remove the routing table entry `FilteredNode` with the smallest corresponding canonical neighborhood spacing.

4.3. FRT-Chord and GFRT-Chord Implementation

The common FRT code above provides a framework which can be extended to filter a subset of nodes from the routing table and not from all non-sticky nodes. Note that it is an implementor’s responsibility to ensure that entry filtering is capable of filtering. Hence, the implementer has to ensure that, when the routing table is full, a routing table entry can actually be deleted by
4. Implementation of Flexible Routing Tables

4.3.1. FRT-Chord

The common FRT code implements most of FRT-Chord. Hence, we do not need to add additional information to a routing table entry. FRT-Chord’s implementation of `rt_entry_info/4` returns `undefined` (Listing 4.14) for all possible arguments. FRT-Chord allows all normal nodes to be filtered without any further restriction. FRT-Chord’s `allowed_nodes/1` implementation in Listing 4.15 thus only filters non-normal nodes.

4.3.2. GFRT-Chord

As described in Section 2.4, GFRT-Chord adds a group flag to a routing table entry, indicating if that node belongs to the same group as the source node. This is reflected in `rt_entry_info/4` (Listing 4.16) of GFRT-Chord. The function `is_from_other_group/1` can be used to check if a given routing table entry belongs to the same group as the source node.

For simplicity’s sake, we assume that nodes belong to the same group if they share the same IP address. We set a routing table entry’s `group` field to `same_dc` if it belongs to the same group as the source node. This is a simplified assumption and can be enhanced further. We will discuss
4.3. FRT-Chord and GFRT-Chord Implementation

```erlang
rt_entry_info(Node, _Type, _PredId, _SuccId) ->
  #rt_entry_info{group =
    case comm:get_ip(node:pidX(Node)) == comm:get_ip(comm:this())
    of
      true -> same_dc;
      false -> other_dc
    end}.
```

Listing 4.16: Extra routing table entry information in GFRT-Chord. A routing table entry contains a field group to indicate whether it belongs to the same group as the source node.

In Chapter 6 how one could make use of the network coordinate system Vivaldi [DCKM04] to use latency considerations in grouping nodes.

GFRT-Chord's allowed_nodes/1 function is more complex than the FRT-Chord variant. Not all variables introduced in Definition 7 are needed to implement it.

In our implementation, we distinguish between the case where we allow all nodes to be filtered and the case where only non-group entries are filterable. When $E_G = \emptyset$, we know that only non-group entries exist in the routing table and thus allow all entries to be considered for filtering. When there are both group and non-group entries, we verify that $E_{\text{leap}}$ is non-empty. So if $E_{\text{leap}} = E_{\text{far}} \cap E_G = \{n \in E_G | d(s, e_\alpha) \leq d(s, n)\} \neq \emptyset$, only non-group entries are allowed to be filtered. We filter sticky nodes from $E_{\text{leap}}$, as Scalaris uses predecessor and successor lists. According to the definition of GFRT-Chord in [NS11], the immediate successor and predecessor of a source node will never be contained in $E_{\text{leap}}$. We extended this to all sticky nodes in the routing table. This is a sensible decision, as the sticky nodes are not filterable in any case. By not allowing any sticky entries to end up in $E_{\text{leap}}$, we ensure that $E_{\text{leap}}$ can never consist of sticky entries only. This concludes the discussion of our implementation.
4. Implementation of Flexible Routing Tables

\[
\text{allowed_nodes(RT) ->}
\]
\[
\text{Source = get_source_node(RT), SourceId = rt_entry_id(Source),}
\]
\[
\text{Nodes = rt_get_nodes(RT),}
\]
\[
\% Compute } E_G \text{ and } E_G
\]
\[
\{E_NG, E_G\} = \text{lists:partition(fun is_from_other_group/1, Nodes),}
\]
\[
\% If } E_G = \emptyset, \text{ we know that we can allow all nodes to be filtered.}
\]
\[
\% Otherwise, check if } E_{\text{leap}} \neq \emptyset.
\]
\[
\{\text{OnlyNonGroupMembers, } \{E_a, E_b\}\} = \text{case } E_G \text{ of}
\]
\[
\{\} \to \{\text{true, ignore, ignore}\};
\]
\[
\{\text{First} | \_\} \to
\]
\[
\text{Predecessor = predecessor_node(RT, Source),}
\]
\[
\text{FirstDist = get_range(SourceId, rt_entry_id(First)),}
\]
\[
\% Compute } e_\alpha, e_\beta \text{ and the respective distances}
\]
\[
\{\{E_{\alpha}, E_{\alpha}\text{Dist}\}, \{E_{\beta}, E_{\beta}\text{Dist}\}\} = \text{lists:foldl1(
\text{fun (Node, } \{\{\text{MinNode, MinDist}, \{\text{MaxNode, MaxDist}\}\}\} ->
\text{NodeDist = get_range(SourceId, rt_entry_id(Node)),}
\text{NewE_{\alpha} = case } \text{erlang:min(MinDist, NodeDist) of}
\text{MinDist -> } \{\text{MinNode, MinDist}\};
\_ -> \{\text{Node, NodeDist}\}
\text{end,}
\text{NewE_{\beta} = case } \text{erlang:max(MaxDist, NodeDist) of}
\text{MinDist -> } \{\text{MaxNode, MaxDist}\};
\_ -> \{\text{Node, NodeDist}\}
\text{end,}
\text{\{NewE_{\alpha}, NewE_{\beta}\}
\text{end, } \{\{\text{First, FirstDist}, \{\text{First, FirstDist}\}\}, E_G,}
\]
\[
\% Is there any non-group entry } n \text{ such that } d(s,e_\alpha) \leq d(s,n) \text{ and}
\% n \neq s.\text{pred? The following line basically computes } E_{\text{leap}} \text{ and checks}
\% if that set is empty.}
\]
\[
\text{\{lists:any(fun(P) \text{ when P := Predecessor -> false;}
\text{\text{(N)}} -> \text{get_range(SourceId,}
\text{rt_entry_id(N)) } \geq \text{ E_{\alpha}\text{Dist}},
\text{not is_sticky(N),}
\text{not is_source(N) \text{ end, E_NG},}
\text{\{E_{\alpha}, E_{\beta}\}\}}
\text{end,}
\]
\[
\text{if OnlyNonGroupMembers -> } \{\text{N || N <- E_NG, not is_sticky(N) , not is_source(N)}\}
\]
\[
\text{; not OnlyNonGroupMembers andalso E_a /= ignore andalso E_b /= ignore ->}
\]
\[
\{\text{N || N <- Nodes, not is_sticky(N) , not is_source(N)}
\text{, N /= E_a , N /= E_b}\]
\]
\text{end.}

Listing 4.17: Filterable nodes in GFRT-Chord
5. Evaluation

In this section, we will compare Scalaris’ Chord implementation and our implementation of FRT-Chord and GFRT-Chord. We will focus on a number of key aspects to evaluate the benefits and disadvantages of our implementation in comparison to Chord. Firstly, we will discuss the difference in finger placement of the algorithms. To understand the generated topologies, we ran simulations with different Chord rings and observed the routing table of a single node within the ring. Here, we will also discuss if FRT-Chord and GFRT-Chord are capable of reaching good routing tables. Secondly, we try to grasp the convergence speed of FRT-Chord routing tables in local simulations. In this section, we will also discuss the merits of our optimization as introduced in Section 4.2.2. Thirdly, we will investigate the performance in terms of average hops and latency. For this, we ran experiments on PlanetLab to test Scalaris under more realistic circumstances.\(^1\) Fourthly, we used a model of churn as described in [RGRK04] to compare the behavior of the algorithms under frequent node joins and leaves. Lastly, we will compare the algorithms’ maintenance overhead by evaluating another PlanetLab experiment.

**A Note on Nomenclature**  In this section, we will run FRT-Chord and GFRT-Chord with differing routing table sizes. We will use FRT-Chord \(i\) to refer to a FRT-Chord routing table of maximum size \(i\), where the maximum size does not include sticky nodes. So, FRT-Chord 8 refers to an FRT-Chord routing table with a maximum size of 8 non-sticky entries. If, in the example of FRT-Chord 8, the system uses one predecessor and one successor per node, routing tables will contain up to 10 entries. We will define the number of successors and predecessors per in each experiment. Scalaris’ default setting allows up to 9 successor and predecessor entries.

---

5. Evaluation

5.1. Finger Placement

To investigate the topologies generated by the different implementations, we simulated a stable Scalaris system and examined the behavior of a single node within the system. The simulated system consisted of a number of nodes which joined the ring successfully. After the nodes are stabilized, we joined a new node and observed the routing table the node generated. This special node was given enough time to choose its preferred routing table entries from all nodes in the ring. In the case of Chord, a node received its preferred entries after running the periodic stabilization routine once. For FRT-Chord and GFRT-Chord, the observed node receives the routing table of its successor and learns all other nodes of the system in random order.

5.1.1. Structure of the Experiment

The experiment is split into two sets, where the node with identifier 0 is picked as the observed node. In the first set of experiments, presented in Section 5.1.2, we simulated a fully-filled ring with nodes \{0, \ldots, 127\}. In the second set, presented in Section 5.1.3, the ring consists of nodes \{0, 1, 11, 21, \ldots, 121\} to simulate a sparsely-filled ring. We will refer to the first set of experiments as experiments on a densely-filled ring and to the latter as experiments on a sparsely-filled ring. Node 0 joins the ring after all other nodes have been joined successfully.

Since we ran this simulation locally, the behavior of FRT-Chord and GFRT-Chord does not differ. This is due to our choice of grouping nodes: as our current implementation uses the IP address of nodes for grouping, all nodes will belong to the same group, and thus GFRT-Chord exhibits the same filtering behavior as FRT-Chord. Hence, we will not discuss GFRT-Chord in this section.

FRT-Chord and Chord handle successor and predecessor nodes differently. Chord includes only the immediate successor node in the routing table, whereas FRT-Chord routing tables can contain more than one predecessor and successor. We will consider this in the experiments by running one instance of FRT-Chord with routing table sizes as large as Chord’s. As we argued earlier that FRT-Chord adapts dynamically to changes in the number of nodes, we also ran experiments with larger routing tables. We will present the results of running the experiments on Chord, FRT-Chord 6 and FRT-Chord 16. Each experiment uses one successor and one predecessor per node.
5.1. Finger Placement

**Expectations** Several expectations motivate the experiments. For the experiments on densely-filled rings, we expect FRT-Chord 6 to build routing tables similar to Chord, and that Chord builds the best routing table under the definition of best routing tables in Section 2. We also expect FRT-Chord 16 to exhibit canonical neighborhood spacings similar to the precomputed value from Section 2.3.

Contrasting these expectations, we expect Chord to build suboptimal routing tables in sparsely-filled rings. FRT-Chord is expected to build better routing tables, namely, that the canonical neighborhood spacings are closer to the expected value. In the case of Chord, the expected canonical neighborhood spacing is 2. For FRT-Chord with the same routing table size, the value is close to 2 as well. Also, FRT-Chord should use all routing table entries if enough nodes exist in the system. FRT-Chord should additionally exhibit exponentially increasing distances between nodes where possible, and fully meshed routing tables when a surplus of entries is available.

The experiments do not reflect a real world setting: in large systems, a node will take a long time to learn all other nodes – if that happens at all. Also, in our simulation, the ring is stable, with no nodes failing.

**Aim of the Experiments** Our aim is to show that our implementation confirms the simulated results in [NS11]. There, the authors show by simulation that a node’s routing table will exhibit exponentially increasing distances between routing table entries over time. In Section 5.2, we will enhance the experiment further to determine the speed of convergence for a node’s routing table.

**A Note on the Figures** The figures in this subsection contain a data point for node 1. The canonical neighborhood spacing for this node is less than 0. For FRT-Chord 16 in a densely-filled ring:

\[ CNS_1(E) = \frac{2}{17} \log_2 \frac{2}{127} \approx -0.70. \]

We will display the entry for node 1 with a CNS of 0, although it is technically less than zero. This does not make a difference for FRT-Chord, as entry filtering does not consider node 1 in the experiments. Node 1 is a sticky node and thus not considered for filtering.
5. Evaluation

5.1.2. Experiments on Densely-Filled Rings

Chord is expected to behave optimally within a ring of maximal size. A Chord node’s routing table is filled completely and the routing table entries grow exponentially. Figure 5.1 depicts the canonical neighborhood spacings (CNS) of Chord and FRT-Chord after one run of the experiment. In said figure, FRT-Chord is present twice, once with a maximum of 8 entries (6 non-sticky and 2 sticky nodes, named FRT-Chord 6) to ease comparison with Chord, and once with up to 18 entries (16 non-sticky and 2 sticky nodes, named FRT-Chord 16) to show that FRT-Chord uses all available routing table entries. In Section 2.3, we precomputed the expected canonical neighborhood spacing for FRT-Chord 16 with two sticky nodes. In this experiment, we expect FRT-Chord 16 to exhibit the canonical neighborhood spacing computed in Section 2.3.

Results As can be seen in the upper graph of Figure 5.1, Chord’s CNS are are of equal value, except for the successor and predecessor fingers 0 and 127. Chord exhibits the best possible routing table. In the same graph, we display the result for FRT-Chord 6. The figure demonstrates that FRT-Chord 6 uses similar nodes for its entries as Chord. Some fingers are placed differently. This difference originates from the order of learned nodes. Depending on the order, FRT-Chord and GFRT-Chord might filter nodes which would be useful to construct an optimal routing table. Consider node 0’s routing table as depicted in Figure 5.1 and that it learns node 64. As its routing table is full, entry filtering must be done. The CNS of this situation is depicted in Figure 5.2. As nodes 1 and 127 are the successors, the CNS of 64 is minimal and 64 will be filtered. FRT-Chord 6 will not learn node 64 in this situation until other nodes have been exchanged.

We can thus conclude that FRT-Chord 6 produces good routing tables in this experiment, but not the optimal ones. If the nodes were learned in the order in which Chord computes its finger, the resulting routing tables for the different algorithms would be the same.

The lower graph in Figure 5.1 depicts the CNS of Chord and FRT-Chord 16. This simulates the situation when the maximum size of the routing tables are chosen to scale up to a large number of nodes, but only a small number of nodes exist in the system. When the system is planned to scale to an identifier space $I = [0, 2^m - 1]$, at least $m$ entries per routing table must be allowed. As can be seen in the Figure, FRT-Chord 16 makes use of all available routing table entries.

\[ \text{The canonical neighborhood spacing} \] is a function defined in Definition 5 in Section 2.3. It computes the exponent $\beta$ of $\alpha^\beta = d(s_i, s_{i+1}) / d(x, s_{i+1})$.

\[ \text{The precomputed value was approximately 0.78.} \]
5.1. Finger Placement

Figure 5.1.: Chord and FRT-Chord 6 produce similar routing tables in this experiment. FRT-Chord 6 diverges from Chord as the routing table depends on the order in which nodes are learned. FRT-Chord 16 fills all available routing table. Consequently, its CNS are smaller than Chord’s. FRT-Chord 16’s CNS is near $\approx 0.78$, which was the value computed in Section 2.3.

Figure 5.2.: When a fully-filled routing table of FRT-Chord 6 is filled with the nodes given here, learning node 64 results in a subsequent filter operation of the same node. Nodes 0 and 127 can not be filtered as they are sticky nodes, and consequently 64 has the minimal canonical neighborhood spacing of the nodes.
5. Evaluation

Figure 5.3.: Results of simulating a sparsely-filled ring of 13 nodes. Both FRT-Chord 6 and FRT-Chord 16 use all available routing table entries. For FRT-Chord 16, this would result in a fully-meshed network in this example.

resulting in a smaller CNS. The CNS of the entries are close to the precomputed value 0.78. The predecessor and successor nodes show a larger deviation from the expected value, which was to be expected.

5.1.3. Experiments on Sparsely-Filled Rings

To contrast the discussion above with simulations aiming for a more realistic setting, we now consider a sparsely-filled ring with nodes \{0, 1, 11, \ldots, 121\}, where node 0 is observed again.

Figure 5.3 depicts FRT-Chord 6, FRT-Chord 16 and Chord again. It is easy to see that both FRT-Chord 6 and FRT-Chord 16 use all available routing table entries. Chord’s routing table is a subset of the corresponding routing table of FRT-Chord 6. When more routing table entries are allowed than nodes exists in the system, as depicted in the lower graph of Figure 5.3, FRT-Chord constructs a fully-meshed network. A node with routing table size larger than the number of nodes can and will contain all nodes in the system, provided that it encounters information about each other node.
5.1. Finger Placement

5.1.4. Finger Placement Histograms

FRT-Chord produces best routing tables only over time. To extend our prior experiment on
finger placement, we ran FRT-Chord multiple times for different routing table sizes and let
the routing tables learn all nodes from a fully-filled ring with nodes 1 to 127 in randomly
chosen order. The resulting routing tables are then merged with prior runs to create a histogram
mapping node identifiers to relative number of occurrences. We will refer to this histogram
as the average routing table. It is obvious that 0’s successors and predecessors are the same
each run, thus they will have a relative frequency of 1. The relative frequency of other nodes is
determined by the filtering mechanism of FRT-Chord.

Structure of the Experiment

FRT-Chord can only produce best routing tables when the nodes are learned in a specific order,
or if nodes are learned repeatedly to disrupt local optima. To investigate this behavior, we ran
the experiment twice. In the first set of experiments, node 0 learns each other node exactly once.
In the second set of experiments, node 0 learns each other node 100 times. The second set of
experiments should produce best routing tables, whereas the first set of experiments should
produce routing tables with exponential distances between entries, but which are unlikely to be
the best routing tables. To distinguish between good and best average routing table, we will
now introduce the notion of optimal nodes.

Optimal Nodes In contrast to Chord, the exponential base of distance growth is not static
in FRT-Chord. Given a best routing table \( \tilde{E} \), we know from Lemma 2 that for the worst-case
reduction ratio

\[
  r_i(\tilde{E}) = 1 - \left( \frac{d(s, \tilde{e}_1)}{\tilde{d}(s, \tilde{e}_{|E|})} \right)^{\frac{1}{|E|-1}}.
\]

Assuming that FRT-Chord reaches a best routing table after learning all nodes, we should be
able to observe exponentially increasing distances between routing table entries, where the
distances grow by the factor \( m_i(E) \):

\[
  m_i(E) = \frac{1}{1 - r_i(E)}.
\]

The factor \( m_i(E) \) is derived from the routing tables reduction ratio:
5. Evaluation

\[ 1 = \frac{d(s, e_i)}{d(s, e_{i+1})} + \frac{r_i(E)}{d(e_i, e_{i+1})} \implies 1 - r_i(E) = \frac{d(s, e_i)}{d(s, e_{i+1})}. \]

For node 0, the successor and predecessor nodes are the nodes 1 and 127. The worst-case reduction ratio for the best routing table \( \tilde{E} \) at node 0 is consequently \( r_i(\tilde{E}) = 0.4994 \ldots \). Accordingly, we can compute the distance growth between entries in \( \tilde{E} \):

\[ m_i(\tilde{E}) = 1.9977 \ldots. \]

It follows that a best routing table in FRT-Chord may consist of other entries than for Chord, as Chord’s distance growth factor \( m_i(E) \geq 2 \). The optimal node for the i-th finger in a best FRT-Chord routing table is the node responsible for the identifier \( s \oplus \lceil m_i(\tilde{E}) \rceil \). In this experiment where we let node 0 learn all nodes repeatedly in random order, the average routing table should include those optimal nodes more frequently than other nodes.

**Expectations**

We expect FRT-Chord to produce good routing tables, namely that the first set of experiments produces average routing table which differ only slightly from the average routing table of the second set of experiments. The second set of experiments should accordingly produce average routing tables with entries placed around optimal nodes. Also, as noted above, optimal nodes should be more frequent than non-optimal nodes in the average routing table produced in the second set of experiments.

**Results**

Figure 5.4 depicts our results for the first set of experiments, where routing tables are allowed to contain one predecessor and one successor. The histograms show that fingers near optimal nodes are more frequent than nodes farther away. Consider the second graph in Figure 5.4, displaying the experiment for FRT-Chord 6. An optimal entry for this routing table size would be node \( 0 \oplus \lceil m_i(\tilde{E}) \rceil = 64 \). FRT-Chord often places entries near this node, but we can not guarantee that the node will be contained in the actual routing table. For the purpose of hop-minimal routing, this does not hurt, as the distance between entries grows exponentially.
Figure 5.4.: The figure depicts the histogram of average FRT-Chord routing tables if the observed node learns each other node exactly once. The average routing table is computed from the results of multiple runs. Nodes nearby optimal nodes are more likely to appear than nodes far-off optimal nodes.

The results of the second set of experiments are shown in Figure 5.5. The figure shows that, when a node learns many other nodes multiple times in random order, optimal nodes are more likely to end in the routing table than non-optimal nodes. Whereas the optimal node 64 was chosen less often by FRT-Chord 6 than node 65, it is chosen often in Figure 5.5.

### 5.1.5. Conclusion

Based on the results of our experiments, we conclude that FRT-Chord should exhibit similar average hops and latency as Chord in cases were the routing tables are of the same size and the ring is densely-filled. This is due to FRT-Chord producing similar tables with exponentially increasing distances between nodes. If the ring is only sparsely or non-uniformly filled, FRT-Chord’s routing tables react dynamically to the system shape. Thus, in such cases FRT-Chord should be able to outperform Chord. This should especially be the case when FRT-Chord can make use of more routing table entries than Chord. In those cases, FRT-Chord will fill as many entries as possible, and consequently reduce the average number of hops for lookups. We will present detailed practical results when we introduce the results of our PlanetLab experiments in Section 5.3.
5. Evaluation

Figure 5.5.: The figure also depicts the histogram of average FRT-Chord routing tables. In contrast to Figure 5.4, the observed node learns each other node multiple times. Then, optimal nodes are even more likely.

5.2. Convergence Speed

In this section, we will investigate the average time a newly joined node needs until its routing table is stabilized. This allows us to analyze if the initial routing table merge on node join is useful, and if our optimization from Section 4.2.2 is an improvement over the naïve approach to routing table refinement.

We split our experiments into two larger sets. In the first set of experiments, we ran our implementation with a fixed active learning lookup interval of 5 seconds. Each round, \( N = 50 \) nodes join the ring. For each of the joined nodes, we measure the time until a routing table state is reached which is not changed within the following 20 seconds. We repeat this experiment under three settings:

(a) With our refinement optimization and with successor routing table merge (SRTM)

(b) With our refinement optimization and without SRTM

(c) Without our refinement optimization and with SRTM

(d) Without our refinement optimization and without SRTM
5.2. Convergence Speed

The second set of experiments focuses on convergence speed under different active learning lookup intervals. To increase the time until the routing tables are stabilized, we deactivated SRTM. Each round, \( N = 10 \) nodes enter the ring. We measured the time until the variance of routing table entries’ canonical spacing \( S_{E}^{j} \) fell below multiple bounds for 20 seconds.

**Variance as an Indicator of Convergence**  Our main indicator to determine the state of routing table convergence is the canonical spacings’ variance. For a fully-filled routing table, the variance of the canonical spacings should be small. This is motivated by Equation (2). Equation 2 showed that the worst-case reduction ratio \( r_{i}(\tilde{E}) \) is constant for a best routing table \( \tilde{E} \). Given this, we can show that the canonical spacings are also constant for a best routing table.

**Lemma 3** Given a best routing table \( \tilde{E} \), the canonical spacings for all entries \( e_{i} \) are constant.

**Proof.** Given in Appendix A.1

We argue in a similar fashion for convergent routing tables: the canonical spacings are not constant, but bound depending on the total number of nodes in the ring. From Equation 2.7, it follows that \( S_{j}^{E} < \frac{4}{|E| - 2} \log \frac{1}{N} \) for a routing table entry \( j \in J \) and \( N \) nodes. We conclude that the canonical spacings should converge towards a constant below the bound. The value of the constant depends on the current number of nodes in the ring and changes over time. As the canonical spacings of a good routing table are similarly large, it follows that the variance of such a table is small. This implies that the variance is an indicator of a routing table’s convergence state.

### 5.2.1. Convergence Speed for Fixed Active Learning Lookups Interval

We will analyze how FRT-Chord’s convergence speed is affected by different parts of our implementation. This helps to understand how good the initial routing tables are when SRTM is enabled, and reveals routing table refinement improvements by our optimization.

**Expectation** We expect all four subexperiments of this section to show decreasing canonical spacings’ variance over time. SRTM should give a good routing table to start with, so experiments with enabled SRTM should outperform the experiments without SRTM. Also, our optimization is expected to be better or at least not worse than the naïve approach to routing table refinement, so an enabled optimization should outperform experiments where it is not
5. Evaluation

Figure 5.6.: Cumulative histograms of the convergence experiments. Convergence is fastest when both SRTM and our optimization are activated. SRTM is more important than our optimization for fast convergence. The optimization outperforms the naïve approach to routing table refinement, while reducing the number of messages sent for maintenance.

enabled. We thus expect convergence in Experiment a to be faster than in Experiment c, faster in Experiment c than in Experiment b, and slowest in Experiment d.

We do not expect the experiment to reflect a real world environment closely. As we do not interact with Scalaris, only Scalaris maintenance base load generates traffic. Thus, the routing table convergence might be faster in a system under real load.

Results Figure 5.6 illustrates cumulative histograms for the convergence time of the experiments. As expected, Experiment (a) shows the steepest increase of the histogram and thus routing tables with both SRTM and our optimization enabled stabilize fastest. The impact of our optimization is hardly noticeable when SRTM is enabled, as can be seen when comparing the results for Experiment (c) with Experiment (a). Both experiments exhibit a median convergence time of almost 12 seconds, as shown in Table 5.1. This is a good result, as it shows that the optimization, while reducing the number of messages sent for maintenance, does not slow down convergence. The impact of our optimization is more obvious when SRTM is disabled. The cumulative histogram of Experiment (b) grows steeper than that of Experiment (d), as was expected. We can thus conclude that our optimization outperforms the naïve approach to routing table refinement while also reducing the number of messages sent.
5.2. Convergence Speed

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Median Convergence Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>11.7663</td>
</tr>
<tr>
<td>b</td>
<td>23.5127</td>
</tr>
<tr>
<td>c</td>
<td>11.8048</td>
</tr>
<tr>
<td>d</td>
<td>26.8485</td>
</tr>
</tbody>
</table>

Table 5.1.: The table simplifies the result depicted in Figure 5.6. Experiment (a) outperforms all other experiments. Using our optimization decreases the convergence time slightly when SRTM is enabled and speeds up convergence by 12.4% when SRTM is disabled.

The histograms show that the initial routing tables with activated SRTM are stable for more than 25% of the nodes. Thus, more than 25% of the nodes finish their stabilization in under one second. Without SRTM, this is only the case for 10% of the nodes. The experiments’ probability density functions given in Figure 5.7 exhibit an interesting behavior: Convergence times peak within periodic intervals. SRTM displaces the peaks. Without SRTM, the peaks occur every 5 seconds. This is also the default interval for active learning lookups. If the interval is chosen to be smaller, the peaks will be placed more densely.

We conclude that SRTM and the optimization speed up routing table convergence. The actual improvement will differ in a real world environment, but we can expect that additional traffic through the system decreases convergence time even more.

5.2.2. Convergence Speed for Different Active Learning Lookup Intervals

We will proceed with our discussion of routing table convergence speed by presenting the active learning lookup interval’s impact. In this experiment, we created a node of 127 nodes and let 10 additional nodes join. For each joining node, we recorded the time until several variance bounds of the routing tables canonical spacings’ where reached. A reached bound must not be broken within subsequent 20 seconds. We ran the experiment for both the naïve and our optimized issuer’s routing table refinement, which we presented in Section 4.2.2.

**Expectation** Longer active learning lookup intervals should result in longer intervals until certain convergence bounds are reached, as the system traffic is lower and nodes are less eager to discover new nodes. Our second expectation is that both the naïve and the optimized issuer’s routing table refinement result in similar convergence periods.
5. Evaluation

Figure 5.7.: Histograms of the convergence experiments. The histogram shows only the frequency of occurrences were the routing table did not converge within 2 seconds of its creation. Without SRTM, the frequency of convergence times peaks at certain intervals. For Experiments (b) and (d), the interval is of length 5 seconds. This is also the default interval for active learning lookups. When the default interval length is changed, the peaks also change their position.

Results Figures 5.8 and 5.9 depict our results for the naïve and optimized issuer’s routing table refinement (IRR), respectively. Surprisingly, the active learning lookup interval did not result in a monotonically increasing function for the convergence time. As can be seen in Figure 5.8(a), sometimes a lower active learning lookup interval did not result in a lower average convergence time. We conclude that the overall traffic through a node is more important for convergence than active learning lookups.

More important than the prior observation is the difference in average convergence time between the naïve and optimized version of the IRR. Figure 5.9(a) shows that the optimized version’s average convergence time is less spread than the corresponding Figure 5.8(a) for the naïve version. The scale of the figures let the optimized version’s average convergence time appear almost constant. Similarly, the standard deviations as depicted in Figures 5.9(a) and 5.9(b) demonstrate that the optimization’s impact on IRR is more important than the choice of the active learning lookups interval. Only the naïve IRR version exhibits a noticeable convergence speed spread when the active learning lookup intervals are longer.
5.2. Convergence Speed

![Graphs showing convergence speed](image)

**Figure 5.8.:** Results with naïve issuer’s routing table refinement. Each line in the figure represents one variance bound. We depict both the average and standard deviation of the time until a bound is reached. The average time until routing tables are converged do not exhibit a strong correlation with the active learning lookups interval.
5. Evaluation

Figure 5.9.: Results with optimized issuer’s routing table refinement. Each line in the figure represents one variance bound. We depict both the average and standard deviation of the time until a bound is reached. The figure again does not indicate a strong correlation between the convergence time of routing tables and the active learning lookup intervals.
5.3. PlanetLab: Average Hops and Latency Experiments

To evaluate our implementation in a more realistic environment, we ran Scalaris with Chord, FRT-Chord and GFRT-Chord on a network of nodes on PlanetLab. In this section, we will describe the structure of the experiment, the expected outcome and the actual results.

5.3.1. Structure of the experiment

Our PlanetLab network consists of 50 physical nodes, distributed over different continents around the world. The actual number of available nodes differs over time, as some nodes are not always reachable. Of the allocated 50 nodes, about 40 were reachable during all experiments. Figure 5.10 depicts the location of the nodes and displays the average round-trip latency (RTL) between each pair of nodes. Intra-continental links have low RTL, whereas inter-continental links have medium to large RTL. Links between Europe and North America and between Asia and Australia are of medium quality (light green), whereas links between Europe and Australia as well as Europe and Asia are often of poor quality (above 350ms RTL).
5. Evaluation

Each physical node runs 10 instances of Scalaris (and thus, dht_nodes). We use this setting to simulate a data center located at each physical node. The RTL between dht_nodes within one node is very small compared to inter-node RTL. The total number of dht_nodes in each experiment lies between 300 and 400 nodes. To avoid destabilizing the system when bootstrapping the experiment, we add only 5 dht_nodes per node at once. After all nodes succeeded in adding their dht_nodes, we proceed with adding the remaining nodes.

The active learning lookup interval is set to 5 seconds. As discussed in Section 5.2.2, the choice of active learning lookup interval does not affect the convergence time noticeably. The default value is thus sufficient for this experiment.

To enable comparing Chord, FRT-Chord and GFRT-Chord, we set the maximum routing table size (excluding sticky entries) of the FRT-based algorithms to 8 entries. Thus, FRT-Chord and GFRT-Chord can scale up to 256 nodes with only non-sticky entries. This bound is intentionally lower than the actual system size, as this allows us to evaluate how FRT copes with routing tables of insufficient size. Successor and predecessor lists are chosen to be 9 nodes long. This is the default value in Scalaris.

Each experiment ran at least 5 hours. During that time, we do not interact with the system, thus only the Scalaris maintenance base load and the benchmarking process generate system load.

Benchmark Process We benchmarked the system using a single node \( n \), located in Berlin, Germany. This special node was stable over the complete experimentation intervals. The system was measured by performing lookups for random keys periodically on the observing node. Each interval, \( n \) issues a burst of 30 lookups for keys generated by calling \?RT:hash_key/1 with the string representation of an integer \( i \). In our version of Scalaris, the resulting key is the string’s md5 hash.\(^4\) After each lookup, \( i \) is increased by one, resulting in a uniform distribution of keys. We chose to use a uniform distribution instead of a more realistic workload to measure different parts of the ring equally. Each lookup is executed by calling \texttt{api_dht_raw:unreliable_lookup/2}. This function chooses one of the dht_nodes on node \( n \). The executing dht_node is not necessarily the same between subsequent lookups.

Using a uniform distribution of keys results in favoring larger keys over smaller keys. Figure 5.11 depicts the histograms when sampling one million keys using our method of hashing a counter versus sampling the keys using Erlang’s random number generator.

The figure demonstrates that choosing keys with a bit length less than 120 is very unlikely. Thus, the nodes responsible for key ranges within \([2^{120}, 2^{128}]\) will exhibit larger load. We do

\(^4\)Version as of 2013-04-18
5.3. PlanetLab: Average Hops and Latency Experiments

![Graph showing relative frequency of sampled keys.](image)

(a) Our method  (b) Erlang RNG

Figure 5.11.: Relative frequency of sampled keys. Figure 5.11(a) depicts the distribution of keys when one million keys are sampled by hashing a counter \( i \). Figure 5.11(b) depicts the distribution of keys when the keys are sampled from a uniformly distributed random number generator. The results indicate that the distribution of md5 is good enough for our purposes.

...not consider a node’s load in this experiment. Additionally, if the observed node is nearby that key range, less hops will be needed to perform the lookups. As the dht_node which performs the lookup is chosen at random from n’s 10 dht_nodes, we do not consider this to be a problem: n’s dht_nodes are distributed over the ring, resulting in some lookups to cross a wide distance.

The interval between lookup bursts ranges from 60 seconds to 100 seconds. The minimum timeout before a new burst is 60 seconds. Consequently, when all lookups issued within one burst finish in under 60 seconds, n will still wait until 60 seconds are over. This is done to keep system load generated by bursts within reasonable limits. To catch lookups with unreasonably long latency, an upper limit of 100 seconds is used. Thus, if a lookup has not finished within 100 seconds, it is declared as failed. Lookups can fail for example if a node receives a lookup but fails before the lookup is forwarded to the next hop.

We collect different statistics. Node n measures the number of hops per lookup as well as the RTL per lookup, and also the amount of failed lookups. We compute average and variances for those variables.
5. Evaluation

<table>
<thead>
<tr>
<th>Routing Algorithm</th>
<th>Hops</th>
<th>Latency [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean Hops</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>Chord (Base 2)</td>
<td>4.331</td>
<td>1.281</td>
</tr>
<tr>
<td>FRT-Chord (8 non-sticky entries)</td>
<td>3.736</td>
<td>0.990</td>
</tr>
<tr>
<td>GFRT-Chord (8 non-sticky entries)</td>
<td>3.716</td>
<td>0.898</td>
</tr>
</tbody>
</table>

Table 5.2.: This table summarizes our results of the PlanetLab experiments. Chord is outperformed by both FRT-Chord and GFRT-Chord in terms of hops and latency. Latency-wise, FRT-Chord and FRT-Chord speed up lookups by about 50 ms. This is a considerable amount, as inter-continental node-to-node RTLs are near that latency. FRT-Chord’s latency standard deviation is larger than Chord’s, which was unexpected. The deviation was caused by a number of nodes which failed during the experiment. GFRT-Chord exhibits a smaller latency standard deviation, as expected.

5.3.2. Expectations

Firstly, we expect FRT-Chord and GFRT-Chord not to be outperformed by Chord by either average hop count as well as average latency. Secondly, we expect FRT-Chord and GFRT-Chord to exhibit lower hop count and latency variances than Chord. We argue that the algorithms allow coping with suboptimal node placement in the ring. Thus, where Chord will refuse to use certain nodes, leaving routing table entries empty. FRT-Chord and GFRT-Chord will use all available routing table entries. Thirdly, we expect FRT-Chord and GFRT-Chord to exhibit lower ratio of failed lookups. This expectation is also motivated by the wider range of nodes usable by the schemes. Lastly, we expect GFRT-Chord to outperform FRT-Chord latency-wise. As we chose the grouping for GFRT-Chord based on the IP addresses of the nodes, this is a natural expectation. GFRT-Chord routing tables will filter entries with nodes on the same physical node only when the routing table is filled completely. GFRT-Chord thus prefers lower-latency routes over hop-minimal routes. We are interested to see if GFRT-Chord can fulfill this expectation, as the routing tables of nodes are largely filled with group members only. With a routing table size of 8 non-sticky entries and a total of 10 \texttt{dht_nodes} on each physical node, a routing table will contain non-group nodes only as sticky nodes. It is thus important to investigate if this preference actually slows down lookups, as more hops are needed.
5.3. PlanetLab: Average Hops and Latency Experiments

Figure 5.12.: The figure depicts the relative frequency of hops and latency resulting from our experiments on PlanetLab. The algorithms’ hop and latency distributions are spread around their mean. FRT-Chord and GFRT-Chord exhibit lower path lengths and also lower average latency. FRT-Chord’s latency is spread more than the latency of Chord and GFRT-Chord, resulting in a larger standard deviation.

5.3.3. Results

Table 5.2 depicts the aggregated results of the experiment. Figure 5.12 shows the distribution of hops and latency. We will now consider the number of hops first and latency afterwards.

**Average Hops** As shown in Table 5.2, Chord’s average hops count is near to its expected value. As noted in Section 1.3.2, Chord is expected to perform lookups with \( \frac{1}{2} \log_2 N \) hops for \( N \) nodes. Given a system size of 360 nodes, Chord should be able to route a request in 4.246 hops on average. This value is similar to the average value our experiment reached. FRT-Chord and GFRT-Chord outperform Chord in the average number of hops. The table confirms one of our expectations: FRT-Chord and GFRT-Chord are not outperformed by Chord in terms of average hops or hops’ standard deviation. Surprisingly, GFRT-Chord needed less hops than FRT-Chord on average. As the averages were generated over the complete runs, transient instabilities cause
5. Evaluation

Figure 5.13.: The figure depicts the average burst latency over time for the experiment, as well as the ratio of failed lookups. All runs were reasonably stable, and only a small amount of lookups failed. The failed lookups did not influence the average latency.

small deviations. GFRT-Chord suffered less failed up lookups than FRT-Chord and Chord. This is the cause for the smaller average hop count. GFRT-Chord’s standard deviation is lower than the deviation of FRT-Chord and Chord. This does not proof a general stability gain when using GFRT-Chord. Both Chord and FRT-Chord exhibited intervals with larger degrees of failed lookups (up to 20%). During those intervals, the hop counts standard deviation increased, affecting the overall average.

Figure 5.12(a) depicts a histogram of the hop count distribution. As expected, the average hop counts is distributed around the average value. Chord exhibits a flatter distribution than the other algorithms. The distributions should be similar to a normal distribution. This is not the case in this experiment. We argue that this is due to system instabilities and the length of the experiment. Scalaris sometimes suffered periods of failed lookups. This can influence the statistics to a larger degree. In the case of FRT-Chord, during such periods path lengths were sometimes shorter than average, thus slanting the hop count distribution.

**Average Latency**  FRT-Chord and GFRT-Chord exhibit lower average latencies than Chord. Interestingly, FRT-Chord’s latency standard deviation is considerably larger than Chord’s.
5.4. Scalaris under Churn

The graph depicting latency over time for FRT-Chord, in Figure 5.13, does not indicate that this is due to the instability which FRT-Chord encountered around 2 hours after starting the experiment. In that period, the average latency was not affected. Our expectation that Chord should not outperform FRT-Chord latency-wise is thus only confirmed for FRT-Chord’s average latency, but not the standard deviation. GFRT-Chord exhibits both a lower average latency and latency standard deviation than Chord and FRT-Chord, as expected.

5.3.4. Conclusion

We conclude that FRT-Chord and GFRT-Chord perform as well as Chord in a globally distributed system. Most of our expectations were fulfilled. FRT-Chord and GFRT-Chord exhibit a similar average number of hops per lookup. Additionally, both algorithms perform well latency-wise. The experiment did not generate sufficient results to conclude that the fraction of failed lookups is lower for FRT-Chord and GFRT-Chord. This fraction depends on the state of the distributed system and is hard to compare outside of a simulation.

5.4. Scalaris under Churn

We adapted the model proposed by Rhea et. al in [RGRK04, p. 5] to simulate the effect of churn on Scalaris. In contrast to Rhea’s approach, we omit simulating the network layer and can thus not compare the latency impact. Instead, we chose the hop count to measure churn impact. This approach leaves out effects like link congestion and other network problems. We also investigated the ratio of inconsistent lookups during churn. As in [RGRK04], a lookup is performed multiple times in parallel on different nodes. It is then defined as inconsistent if there is no majority of equal endpoints for the lookup.

Churn was simulated using a Poisson process $\mathcal{P}$. Each interval, $\mathcal{P}$ chooses a node to stop. This node is picked uniformly at random from all nodes in the system. To keep the number of nodes in the system stable, after stopping the node a new one is started immediately. $\mathcal{P}$ is based on an exponential distribution with churn rate $\lambda$. The average inter-event interval between subsequent churn operations is consequently $\mu = \frac{1}{\lambda}$. We chose $\lambda$ such that the expect median session interval $t_{med}$ of nodes in the system reflects the values chosen in [RGRK04]. A node’s session time is defined as the interval between the node join and leave. Rhea et. al show that $t_{med}$ can be computed in advance with Eq. (5.1), if the number of nodes $N$ is known.
5. Evaluation

\[ t_{\text{med}} = \frac{N \ln 2}{\lambda} = \frac{N \ln 2}{\mu} \]  

(5.1)

Given Eq. (5.1) and a median session time of one hour within a system of 1000 nodes, they show that the expected interval \( \mu \) between subsequent node kill operations is 5.2 seconds.

Due to missing system resources, we simulated a Scalaris ring of size 127 and granted FRT-Chord at most 6 non-sticky entries. We expect to see churn impact with this low number of nodes, as the median session time is proportional to the number of nodes.

Our simulation uses a single Poisson process \( \text{load\_generator} \) to generate system load. It generates events with an event rate of \( \lambda_{\text{load}} \). Each event, 8 randomly chosen nodes perform a lookup for a uniformly distributed key \( K \) from the available key range. The rate \( \lambda_{\text{load}} \) is chosen such that the generated load is expected to be 128 messages per second. Thus, \( \text{load\_generator} \) should create 16 events per second and thus \( \lambda_{\text{load}} := 16 \text{ events/s} \). Consequently, the expected time between events equals 0.0625.

Similar to our prior simulations, we did not simulate GFRT-Chord separately, as the results will be the same as for FRT-Chord.

5.4.1. Structure of the Simulation

We extended Scalaris’ supervision tree by three additional processes, which run over the whole lifetime of the Scalaris VM. The Poisson process \( P \) to generate churn is named \( \text{random\_killer} \). It orchestrates the experiment and partitions each simulation into different rounds. Each round, the median session time is halved. Rounds are separated with intervals without churn, in order to allow Scalaris to stabilize. As mentioned before, the system load for our experiment is generated by a process named \( \text{load\_generator} \). We collect hop and consistency statistics using a centralized process \( \text{collect\_churn\_stats} \).

Generating Churn  The churn generator \( \text{random\_killer} \) is a finite state machine consisting of two states \( \text{active} \) and \( \text{inactive} \), depicted in Figure 5.14. \( \text{random\_killer} \) initiates transitions by sending messages to itself.

After starting the process, it sends the message \( \text{setup\_experiment} \) delayed by one second to itself. Upon receiving \( \text{setup\_experiment} \), the message handler starts the needed number of \( \text{dht\_nodes} N \). \( \text{random\_killer} \) schedules its first \( \text{kill} \) message, which will be received after one minute. On handling the message \( \text{kill} \), a churn operation is performed. In such an
5.4. Scalaris under Churn

![Finite state machine](image)

Figure 5.14.: Finite state machine to stop `dht_nodes` randomly.

\[
\begin{align*}
\lambda_0 &= 0.00394398 \\
\lambda_1 &= \frac{\lambda_0}{2} \\
\vdots \\
\lambda_i &= \frac{\lambda_{i-1}}{2} \\
\end{align*}
\]

Figure 5.15.: The churn simulation is run in subsequent phases. Each churning phase is of length 20 minutes. The median session time of a node is determined by \( \lambda_i \), which is halved each interval. Phases are interspersed with churn-free periods. Each such period is one minute long.

operation, a node is chosen to be stopped, and the next kill message is scheduled by drawing an interval from an exponential distribution. The rate of the exponential distribution depends on the current round \( i \).

\[
\lambda_i := \begin{cases} 
0.00394398 \ldots & i = 0 \\
\frac{\lambda_{i-1}}{2} & \text{otherwise} 
\end{cases}
\] 

(5.2)

\( \lambda_0 \) was chosen such that the initial \( t_{med} = 6.2 \) h. This is the start value given in the original paper [RGRK04].

After each round, `random_killer` enters a phase without churn by receiving a delayed message `deactivate`. Upon entering the state `inactive` after the \( i \)-th round, the churn rate \( \lambda_i \) is halved, generating \( \lambda_{i+1} \) for the following round. `random_killer` stays in the state `inactive` for one minute, before reentering the state `active`.

This results in a series of churn experiments interspersed with intervals without churn, as depicted in Figure 5.15. The intervals of churn are chosen to be 20 minutes long, whereas the calm intervals end after one minute.
Generating System Load  To measure both average hops and consistency of hops, we opted for a centralized load generator. The process load_generator uses an exponential distribution with fixed rate $\lambda_{load}$ as described above. load_generator is a single-state finite state machine and sends itself lookup messages. The message delays are chosen from the underlying exponential distribution. The message handler for lookup messages generates the load. It schedules the next lookup messages and issues the lookups relevant for the experiment afterwards. Listing 5.1 depicts the lookup generating part of the message handler.

The procedure to generate the lookups is designed similarly as the model in [RGRK04]. It chooses a key $K$ uniformly at random. Then, from all dht_node, 8 are chosen uniformly at random. Each of those nodes is instructed to perform a lookup for $K$. This results in a list of requests. The requests are marked with a unique identifier Ref, allowing load_generator to order incoming replies. The state of load_generator holds an entry for each lookup request, which times out after lookups_timeout() seconds.

The remarks from Section 5.3.1 on key choice also apply here: the chosen key $K$ is very likely to be large. We expect this not to skew the results as we chose 8 nodes from the set of all nodes at random. Consequently, some of the nodes are likely to be near the key whereas others are likely to be further away. The aggregated results should thus reflect both long and short paths.

Listing 5.1: Choosing 8 nodes to perform a lookup
5.4. Scalaris under Churn

The endpoints of the lookups reply directly to the central load_generator. It is then checked the reply was expected and if the lookup timed out. After waiting the maximum time until the lookups timed out, namely on receiving the message \{lookup_clear, Ref\}, load_generator computes the average hops of the received responses, and the ratio of the majority of lookups with the same endpoint against the total number of lookups. If no majority exists, this number equals zero. The collected results are then forwarded to the process collect_churn_stats.

Collecting Statistics The process collect_churn_stats collects hops and consistency statistics for lookups generated by load_generator. For each set of lookups, load_generator sends a message containing the end results to collect_churn_stats, which accumulates the values for two seconds until writing them to an output file. The output file contains average hops and the hops’ variance, the number of lookups and the average fraction of consistent lookups.

5.4.2. Expectations

We expect our simulation to show that Scalaris is coping well with churn for all algorithms described in this thesis. This includes Chord as well as FRT-Chord and GFRT-Chord. As all algorithms use the same procedures for ring maintenance, we expect to see a similar impact on all of them. In the simulation by Rhea et. al, churn did not lead to bigger inconsistency in lookups when using Chord. We also expect to this to be the case for FRT-Chord.

5.4.3. Results

As depicted in Figure 5.16, the simulation does affect neither Chord nor FRT-Chord considerably. The average number of hops is within a small margin around the expected number of hops for a system of 128 nodes. As noted before, Chord is expected to use $0.5 \log_2 N$ hops w.h.p. for lookups. In our simulation, the number of hops used for lookups in Chord should be spread around 3.5 hops, excluding the final hop. This is the case for both Chord and FRT-Chord, which leads us to the conclusion that either our simulation is not sufficient to simulate Churn, or that the median session times are too large for simulating on a single node. We propose that the simulation should be rerun on a distributed cluster in future work, simulating network congestion as well. The ratio of inconsistent lookup is also unaffected by
5. Evaluation

![Graph]

Figure 5.16.: The figure depicts the results of our churn simulation. Each data point represents the averaged number of hops needed to perform a lookup. Each round, the median session time is halved. The x-axis depicts the median session time. Neither Chord nor FRT-Chord are impacted by our churn simulation and consequently the average values are independent of the median session time.

the simulation. Figure 5.16 does not depict the ratio, as it is constant. This concurs with the results on inconsistent lookups in [RGRK04].

5.5. Maintenance Overhead

To conclude our evaluation, we also investigated Scalaris traffic usage. We ran an experiment similar to the PlanetLab experiment from Section 5.3 and measured in- and outgoing traffic rates of two nodes within the network.

Structure of the Experiment We measure the traffic over time for two nodes within a PlanetLab network. The nodes differ in their roles within the network: while both of them have one dht_node, one of the node is also hosting Scalaris’ management server. Figure 5.17 depicts the structure of the experiment. Each of the two nodes planetlab02.tkn.tu-berlin.de (TUB) and planet1.zib.de (ZIB) measures the in- and outgoing traffic in terms of bytes per second. We devise Scalaris’ internal statistic mechanisms for this: Scalaris aggregates
5.5. Maintenance Overhead

Figure 5.17.: Structure of the maintenance overhead experiment.

The performance statistics within the process monitor. This statistic includes all traffic leaving the Erlang VM over sockets used by Scalaris.\textsuperscript{5} The traffic statistics are logged over intervals of 10 seconds, and can be accessed by calling the function \texttt{statistics:getGaugeMonitorStats/4}.

In terms of number of nodes and routing table sizes, the same experiment structure as for the PlanetLab experiment in Section 5.3 applies.

The experiment ran for 4.5 hours for each routing algorithm. Similar to the prior PlanetLab experiment, each physical node ran a number of Scalaris nodes. The total number of nodes in the system was around 350. The experiment reflects only base load traffic as we did not implement additional load generators.

**Expectations** We expect the traffic usage pattern to be consistent over the different routing schemes. None of the routing schemes should generate considerably more traffic than the others. Although FRT-Chord and GFRT-Chord do not use periodic stabilization and thus send less messages for maintenance than Chord, we still expect Chord to generate only as much as traffic as the other algorithms. The algorithms share the same basic Scalaris maintenance routines, which are likely to produce more traffic than the algorithms’ maintenance protocols.

**Results** Table 5.3 combines the averaged results of the observed period. Scalaris baseline outgoing traffic on the two nodes is near 50 KByte/s, whereas incoming traffic lies between 30 and 40 KBytes/s. Figure 5.18 depicts the maintenance overhead over time for ZIB. The results for TUB were similar and we thus leave them out here. The figure does not exhibit surprises, except for the floating averages’ computation artifacts in the left beginning of each graph. The traffic usage alternates between high and low values. FRT-Chord exhibits smaller received maintenance traffic on ZIB than the other algorithms. We believe this to be the result of an instability in PlanetLab. If nodes have to resend messages often, e.g. when messages are lost in transit, outgoing traffic will be higher than incoming traffic. Table 5.3 shows that such discrepancies are given for the other algorithms as well.

\textsuperscript{5}Note that traffic within the VM is not observed.
5. Evaluation

<table>
<thead>
<tr>
<th>Site</th>
<th>Routing Algorithm</th>
<th>Send [KByte/s]</th>
<th>Receive [KByte/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Chord (Base 2)</td>
<td>52.5644</td>
<td>31.9768</td>
</tr>
<tr>
<td>TUB</td>
<td>FRT-Chord (8 non-stickies)</td>
<td>55.9533</td>
<td>33.6335</td>
</tr>
<tr>
<td></td>
<td>GFRT-Chord (8 non-stickes)</td>
<td>55.3664</td>
<td>40.0799</td>
</tr>
<tr>
<td></td>
<td>Chord (Base 2)</td>
<td>47.4744</td>
<td>40.9026</td>
</tr>
<tr>
<td>ZIB</td>
<td>FRT-Chord (8 non-stickies)</td>
<td>52.4702</td>
<td>30.6072</td>
</tr>
<tr>
<td></td>
<td>GFRT-Chord (8 non-stickies)</td>
<td>51.0098</td>
<td>43.8243</td>
</tr>
</tbody>
</table>

Table 5.3.: Aggregated results, averages of send/received traffic

The different algorithms show very similarly evolving maintenance costs. Also, our implementation does not create more overhead than Scalaris’ Chord implementation and neither does Chord generate more traffic. We argue that our expectations are thus met.
Figure 5.18.: The figure depicts maintenance overhead over time for the node ZIB. The data points are 60 s aggregates. All algorithms produce a similar amount of maintenance overhead. FRT-Chord generates less incoming maintenance traffic than the other algorithms. The peaks at the beginning of the graphs are produced by the averaging of aggregates.
6. Future Work

Our design decisions leave room for future work on FRT-based algorithms. We will give a short overview over possible improvements to our implementation. Also, we will explain how a future evaluation could investigate the merit of FRT-Chord and GFRT-Chord further.

6.1. Improving FRT-Chord and GFRT-Chord

The flexibility of FRT-Chord allows many improvements over the original algorithm. We presented one small improvement when we introduced GFRT-Chord, which adds a simple grouping mechanism to FRT-Chord.

Our first suggestion is to add a cache of nodes to FRT-Chord. To mitigate filtering of commonly used nodes, a least-recently-used cache could be added to the routing table. The effects of such a cache are hard to determine beforehand. One possible outcome of a caching scheme could be that requests to popular nodes (e.g. nodes which store a popular website) end up in many caches at once. If that is the case, routing requests to those popular nodes should be fast.

Another option to evaluate the merit of FRT within Scalaris would be to optimize on latency instead of hop count. This is complicated with FRT-Chord, as the exponential spacing between routing table entries is not necessarily given anymore. To implement this, the common filtering mechanism as presented in Section 4.2.3 must be adapted, e.g. to implement a small-world network: the filtering mechanism could keep a small number of far-reaching entries and filters other nodes with large latency.

Finally, GFRT-Chord can be improved by using a more flexible grouping scheme than by querying the IP address of nodes. The grouping should preferably allow to estimate the latency between nodes before actually routing to them. A possible solution for latency estimates are virtual latency coordinate systems, such as Vivaldi [DCKM04]. Scalaris contains a usable implementation of Vivaldi. By querying the local component of the gossip algorithm, latency estimates can be added to routing table entries.
6. Future Work

6.2. Improving our Evaluation

Due to space and time constraints, our evaluation leaves out certain experiments and simulations which are interesting when evaluating the real world effects of FRT. Probably most important is Scalaris’ behavior under churn. We introduced a simulation in Section 5.4, but did not implement a full-scale network layer simulation. We think this is important to further analyze churn impact on hop count, latency and consistency. Especially lookup inconsistencies can be problematic, as we indicated in Section 1.3.3. We believe further analysis on the phenomenon’s frequency of occurrence to be important.
7. Conclusion

In this thesis, we presented our contributions to the peer-to-peer layer of Scalaris. In Section 1, we discussed the basic knowledge needed to understand our work, namely distributed hash tables, consistent hashing and Chord. Using this knowledge, we proceeded with an introduction of the concept flexible routing tables (FRT). FRT uses all available information for constructing routing tables by adding all encountered nodes to routing tables. Routing table entries are only deleted when routing tables are full. Entries are filtered such that requirements are met, e.g. by filtering entries which do not impact routing with short path lengths. Two instances of this concept were discussed, FRT-Chord and GFRT-Chord, both of which use a Chord ring topology. FRT-Chord’s filtering mechanism is designed to emulate Chord-like finger placement, namely, with exponentially increasing distances between routing table entries. This is achieved by computing the canonical neighborhood spacing of routing table entries and subsequent filtering of the entry with the smallest such value. GFRT-Chord extends FRT-Chord by the notion of node groups. It aims to consider more than only distances when filtering routing table entries.

We outlined Scalaris’ peer-to-peer layer in Section 3.1 and proceeded with a presentation of our implementation in Section 4. The implementation uses a set of common functions, placed in the header file rt_frt_common.hrl. We optimized entry learning by using a property of the Chord ring: Upon forwarding a lookup request, a node can check if the first hop was far enough. If this is not the case, then the node can decide to send the next-hop node to the original issuer of the lookup. With this, routing table convergence can be sped up.

We evaluated our implementation in in Section 5. To compare which nodes a node is likely to put into its routing table, We discussed the finger placement of FRT-Chord and Chord. It can be shown that fingers in FRT-Chord are placed nearby fingers chosen by Chord, if the routing tables are of similar size. The actual placement differs, as FRT-Chord’s finger placement depends on the order in which entries are learned. To complement the finger placement with speed considerations, we also discussed convergence speed in FRT-Chord. We were able to show that merging the successor’s routing table on join results in good first routing tables. Also, our optimization speeds up routing table construction. In Section 5.3, we evaluated the
performance of our implementation by running it on PlanetLab. The experiments were intended to give a clearer picture of the routing algorithms’ behavior in a more realistic environment. We concluded that FRT-Chord and GFRT-Chord outperform Chord when using the same routing table size in both overlay path lengths and latency. We attempted to understand the behavior of Scalaris under churn by devising a simulation approach presented in [RGRK04]. It turned out that our simulation did not result in considerable churn impact. We concluded that this is not sufficient to argue that Chord, FRT-Chord and GFRT-Chord do not suffer under churn. Instead, future work has to be done were the layers underlying the distributed hash table are simulated in more detail. Especially network queues can have considerable impact on availability and performance. We discussed maintenance overhead in the end of Section 5.5. The maintenance overhead of our implementation is similar to prior routing table algorithms in Scalaris.

We conclude that FRT-based algorithms are a suitable replacement for more traditional routing table algorithms. FRT-Chord offers similar or even superior performance than Chord, and can be extended easily to allow more considerations than only path length optimization. As the glue-code of FRT implementations can be shared to a large degree, implementing new routing algorithms based on the concepts of guarantee of reachability, entry learning and entry filtering can be achieved in a short amount of time.
A. Appendix

A.1. Proofs

Proof of Lemma 1. Consider routing table entries $e_i, e_{i+1}$. To prove that the worst-case reduction ratio for $e_i$ is encountered when the key equals $e_{i+1}$, the following must hold for all $k \in (e_i, e_{i+1})$:

$$\frac{d(e_i, e_{i+1})}{d(s, e_{i+1})} > \frac{d(e_i, k)}{d(s, k)} \quad (A.1)$$

Assume that Eq. (A.1) does not hold. Then:

$$\frac{d(e_i, e_{i+1})}{d(s, e_{i+1})} \leq \frac{d(e_i, k)}{d(s, k)} \implies d(e_i, e_{i+1})d(s, k) \leq d(e_i, k)d(s, e_{i+1})$$

$$\implies (d(s, e_{i+1}) - d(s, e_i))d(s, k) \leq (d(s, k) - d(s, e_i))d(s, e_{i+1})$$

$$\implies d(s, k) - \frac{d(s, e_i)}{d(s, e_{i+1})} \leq d(s, k) - d(s, e_i)$$

$$\implies \frac{d(s, e_i)}{d(s, e_{i+1})} > d(s, e_i)$$

The last implication does not hold, as $d(s, e_{i+1}) > 1$. Hence, the assumption cannot be true, which completes the proof.

Proof of Theorem 3. Let $E$ be a best routing table. From Lemma 2, we know that $\forall i \in \{1, \ldots, |E|\} : r_i(E) = \text{const}$. Let $i, j \in \{1, \ldots, |E|\}$.

$$r_i(E) \text{ const} \Rightarrow r_i(E) = r_j(E) \overset{r_i(E) = 1 - 2^{-S_i^E}}{\Rightarrow} S_i^E = S_j^E \text{ const}$$

99
A. Appendix

which concludes the proof. It also follows that

$$\Rightarrow S_{i-1}^E + S_i^E = 2 S_i^E \text{ const}$$

As the canonical neighborhood spacing for a routing table entry $e_i$ of a table $E$ is given by $S_{i-1}^E + S_i^E$, we conclude that the canonical neighborhood spacings for all entries in a best routing table are also constant. \qed
Bibliography


Bibliography


[KSB⁺99] Karger, D.; Sherman, A.; Berkheimer, A.; Bogstad, B.; Dhanidina,
Bibliography


[MBR03] Manku, Gurmeet S. ; Bawa, Mayank ; Raghavan, Prabhakar: Symphony: Distributed Hashing in a Small World. In: *USENIX Symposium on Internet Technologies and Systems*, 2003. – The authors claim that bidirectional lookup can bring latency improv. up to 30 %


Bibliography


