HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Modular Implementation and Evaluation of a Location-aware Peer-to-Peer Overlay

Bachelor Thesis

by

Olga Batiukova

born in
Kertsch

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

February 2017

Supervisor:
Jun.-Prof. Dr.-Ing. Kalman Graffi

Abstract

The main goal of this bachelor thesis is to develop an implementation of a local-based protocol *Geodemlia* and to ensure that this implementation works properly. The overlay is considered to be implemented in modules, so that they can be extended or removed without high expense. To generate the real-world conditions, the *PeerfactSim.KOM* simulator was used.

In order to manage this task, a detailed description of the theoretical functionality of the protocol is introduced. An overview of the practical implementation in Java is also given in this bachelor thesis.

To ensure the correctness of the protocol's implementation, the overlay is tested and the test results are evaluated.

Acknowledgments

First of all, I would like to thank Jun.-Prof. Dr.-Ing. Kalman Graffi for an excellent course in *Computer Networking*, which he held in winter semester 2015/16. This course has awoken my interst in Computer Networks and sparked an idea of writing my bachelor thesis in this field. I also would like to thank him for giving me the opportunity to write this bachelor thesis under his supervision.

Special thanks to my advisor, Mr. Tobias Amft, for making a hard process of writing a bachelor thesis into an interesting journey full of new knowledge and experience: he was like a white rabbit, whom I had to follow; and falling down the rabbit-hole, it became "[...] curiouser and curiouser!" [Car]. Thank you, for being patient, for answering all the numerous questions I had and for giving me precious pieces of advice.

I am also very grateful to my beloved friend, Martin Böckmann, for supporting me during these three months. Thank you for believing in me and for not letting me give up!

Additionally, I wish to express my gratitude to "the most awesome" friend, Alexander Diez, for the constructive critisim and helpful pieces of advice he gave me.

И в заключение, мне бы хотелось поблагодарить мою семью: моих маму и папу, тётю и бабушку. Спасибо вам за ваше бесконечное терпение, вашу теплоту и трепетную заботу, с которой вы ко мне относитесь, а также за всю ту поддержку, которую вы мне дарите! Вы научили меня всему, что я умею, и потому только блыгодаря вам я смогла написать эту работу. Спасибо! Я вас очень люблю!

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In Section 1.1, the motivation for this bachelor thesis is given. Section 1.2 gives a brief overview of the structure of this thesis.

1.1 Motivation

Information has always played an important part in our society. And as the life is becoming increasingly faster, live communication and efficient information exchange are becoming more important every day. Writing letters have already sunk into oblivion, as computer networks have opened a new page in the history of an information exchange. They make the whole process easier, faster and a lot more efficient.

Speaking about computer networks, it is necessary to mention that there are two types of computer network organisation: *client-server* architecture and *Peer-to-Peer* (*P2P*) architecture.

Client-server models are centralized. It means that all the data is stored on the powerful computers called servers which makes them responsible for managing the whole infomation and resources [JFK12], [Wikb]. These computers provide information for other less powerful computers called clients (desktops, laptops, smartphones, and so on). In order to get some information from the Internet, a client have to send a response message to a server which in its turn will proccess the response, find the appropriate information and send it back to the client.

Peer-to-Peer networks are, on the contrary, mostly or completely decentralized. All computers are connected with each other in the way, so that they can access the information and share the resources stored on multiple hosts (*peers*) [JFK12], [Wikd]. Each and every peer is a client and a server at the same time which makes it possible for the computers to directly exchange the information with each other even if one of the peers, holding some piece of the information needed, is offline at the moment.

Though *client-server* type of computer network organisation makes it easy to store and back up information, *P2P* networks gain in popularity in the last years. Lots of traffic-intensive applications are based on *P2P* architecture, including file sharing (e.g., BitTorrent) and Internet Telephony (e.g., Skype) [JFK12]. The main reason for such popularity is that *client-server* computer networks are prone to congestions. If a server crashes, the whole network will go down, as it happened, for instance, in the USA on October, 21 2016. On that day much of the American Internet was brought down as a result of an DDOS attack on the Dyn servers [Woo]. The *P2P* networks are, on the contrary, more reliable. If you are, for example, downloading a file and the peer distributing it suddenly goes offline, there are other peers distributing the same file.

The *P2P* overlays make use of the numerous protocols and routing algorithms, such as *Chord*, *Napster*, *Kademlia* or *Gnutella*.

In recent years, mobile communications are gaining in popularity. Cell phones have changed the lifestyle of a whole generation. Nowadays, it is almost impossible to imagine our life without Facebook, Twitter or other social networks. Almost everyone wants to share the information with their friends and relatives as fast as possible, which requires an immidiate Internet access and appropriate location-based technologies on the smartphones. And as current devices are equipped with GPS, position-aware applications are conquering the market with the speed of light [LYH13]. This allows users to share not only small pieces of information, but also large size files, such as high resolution pictures and short videos [GRS$^+$13]. Such popularity results in a variety of approaches for location-aware search, such as the hierarchical tree-based concept or the approach of space filling curves [KW06a], [KW06b], [GSR$^+$12]. The main disadvantages of these approaches are either load-balancing and scalability problems or the impossibility to preserve the directionality and locality [GSR$^+$12]. In order to overcome these difficulties, a robust P2P overlay called *Geodemlia* was developed.

The goal of this bachelor thesis is to implement this protocol and to test its functionality. While writing this thesis, the *PeerfactSim.KOM* simulator was used, which made it possible to generate, test and analyse the overlay.

1.2 Structure of the Thesis

In Chapter 1 the motivation for this thesis, as well as its structure are presented.

Chapter 2 includes the description of the *PeerfactSim.KOM* simulator used for the simulations. The main properties of such protocols as *Chord* and *Kademlia* are also shown in this chapter. Considerations about basic modules used for applications and overlays are presented at the end of this chapter.

In Chapter 3 theoretical concepts of the *Geodemlia* protocol are given. The main operations in *Geodemlia* are also presented in this chapter.

The implementation of the *Geodemlia* protocol in Java is given in Chapter 4.

Chapter 5 presents the data to be analysed. The evaluation of this data is also given in this chapter.

Finally, a short summary, as well as the ideas for future work are presented in Chapter 6.

Chapter 2

PeerfactSim.KOM And Basic P2P Routing Protocols

The first step towards an implementation of the proposed protocol is to understand, what makes *Geodemlia* differ from other existing protocols. It is also necessary to understand the functionality of the simulator, the protocol will be developed under. For this reason, this chapter provides a brief overview of the functionality of basic P2P routing protocols, such as *Chord* and *Kademlia*. The main principles and properties of the *PeerfactSim.KOM* simulator, used to generate the overlays, are also described in this chapter. Conclusively, the main basic modules, that can be found in each overlay, are presented.

2.1 PeerfactSim.KOM

PeerfactSim.KOM has been developed in Java by the working group of *Prof. Dr.-Ing. Ralf Steinmetz* at the *TU Darmstadt* at the *Multimedia Communications Lab* (KOM) [Pee] in order to test and analyse the dependencies in large scale P2P networks. This simulator is not a static project and can be further extended by other working groups, as it has been done, for instance, at the *University of Düsseldorf* (HHU) by the working group of *Jun.-Prof. Dr.-Ing. Kalman Graffi* or at the *University of Paderborn* (UPB) by *Thim Strothmann* and *Matthias Feldotto* [Pee].

As it is always very difficult to foresee the future, the main challenge for the developers of *PeerfactSim.KOM* was to create a simulator that could work independently from the network architecture and could fit the wide spectrum of the use cases [Ste11]. The simulator consists of several layers, each of which can be set up by the user. This layer structure is presented in Figure 2.1.

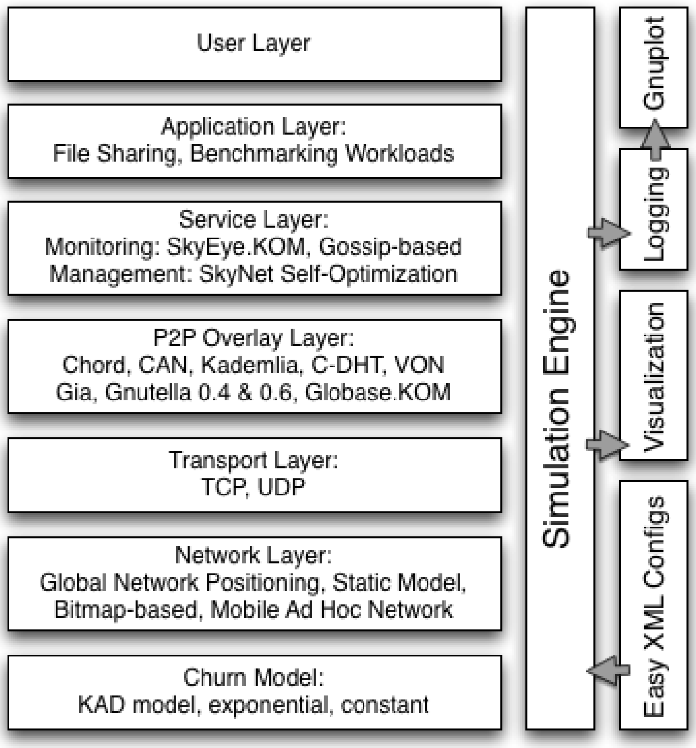

User Layer
Application Layer:
File Sharing, Benchmarking Workloads
Service Layer:
Monitoring: SkyEye.KOM, Gossip-based
Management: SkyNet Self-Optimization
P2P Overlay Layer:
Chord, CAN, Kademlia, C-DHT, VON
Gia, Gnutella 0.4 & 0.6, Globase.KOM
Transport Layer:
TCP, UDP
Network Layer:
Global Network Positioning, Static Model,
Bitmap-based, Mobile Ad Hoc Network
Churn Model:
KAD model, exponential, constant
Simulation Engine
Gnuplot
Logging
Visualization
Easy XML Configs

Figure 2.1: Structure of PeerfactSim.KOM [Pee].

The overlay can be started and manipulated through two types of files: an *XML*-file, as well as a *DAT*-file. In the *XML*-file the basic structure of the overlay is defined, as well as parameters, such as churn rate or the number of peers per region, can be set. The *DAT*-file includes commands that come from the user and can be directly called out from the protocol code.

As mentioned in the paper written by *Matthias Feldotto* and *Kalman Graffi*, this simulator is event-driven. This means that all events "[...] follow a timeline which assures sequential processing" [FG13]. In this case, events are represented as operations that can be generated either by the simulator itself or through the *DAT*-file. For each event it is possible to set the start time and the delay, which enables a much easier management of the operations. It is also possible to use and analyse different protocols at the same time, as "[...] the operations on various layers are decoupled" [FG13].

2.2 Chord

Chord is one of the basic high-structured protocols for P2P networks. It was developed at the *MIT Laboratory of Computer Science* and introduced in 2001 by *Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek* and *Hari Balakrishan* [Wika].

This protocol is one of the basic distributed hash tables (DHT), that is why it provides just one operation: mapping a key to a node. In order to map the keys to the nodes, a consistent hashing is used [SMK$^+$01]. For load-balancing and scalability purposes, each peer stores information about just a fixed number of peers.

In this overlay the nodes are arranged on the flat address space so that they create a circle. Instead of IP addresses, m-bit identifiers, created by using the *SHA-1* hash function, are used. There are at most 2^m nodes with the identifiers ranging from 0 to $2^m - 1$. The value of m should be, though, large enough in order to avoid collisions. Each node has only the information about its successor and its predecessor and is responsible for an area between its ID and the ID of its predecessor [TU]. A successor is a node that is next in the circle in a clockwise-direction. Under predecessor a node that lies next in the counter-clockwise direction is meant.

Each node has *short cuts* (*fingers*) to other nodes further in the ring that are stored in the *finger tables*. Such tables have up to m entries where the successors$((n+2^{i-1})\ mod\ 2^m)$ are stored. The first entry in a *finger table* is always an immediate successor. If a key k must be found, a node passes a response message to the nearest successor or predecessor of this key in its *finger table* until the key is found. Such routing algorithm helps to avoid the linear search time as the search is done in $\mathcal{O}(logN)$ steps. An example of the nodes organisation and finger tables in *Chord* is shown in Figure 2.2.

2.3 Kademlia

Kademlia is a robust P2P distributed hash table that minimizes the number of messages sent between nodes and avoids timeout delays [MM]. This protocol was developed by *Petar Maymounkov* and *David Mazières* in 2002. Due to its decentralized structure, *Kademlia* is much more resistant against the DoS attacks than many other protocols, since the overlay recovers itself around the nodes that had been flooded [Wikc].

Equally to *Chord*, *Kademlia* makes use of the consistent hashing using the *SHA-1* function to map a key to a value. Each node has a 160-bit node ID that is used to locate values. The ID number must be

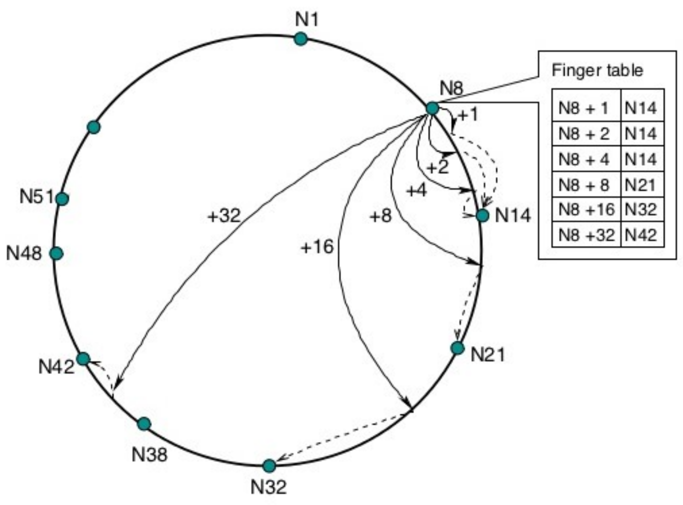


Figure 2.2: An example of a finger table and organisation of nodes in Chord (Adapted from [Wika]).

unique for every node and can be generated by hashing the IP address of a node or just by generating a random number [TU]. Using the node ID, it is possible to locate some value to this node. In contrast to other protocols, *Kademlia* uses the same routing algorithm from start to the end, in order to locate a node near a particular ID [MM]. Each peer stores the data whose IDs are next to its own ID. For the distance calculation the *XOR* metric is used. The *XOR* function is symmetric, which means that a distance from node A to node B is the same as the distance from node B to node A. Such property allows the system to learn useful routing information from the received messages, so that the peers participating in the overlay receive the queries from exactly the same distribution of nodes in their routing tables [MM].

The routing table in *Kademlia* is represented as a binary tree whose leaves are nodes in the overlay. The position of a node in the routing table is determined by the shortest unique prefix of its ID. In Figure 2.3, the position of a node 0011, as well as the division into three successively lower subtrees (circled areas) are represented. The first subtree is as large as the half of the binary tree, the second one is as large as the half of the remaining tree and so on. Such structure ensures that every node knows of at least one node in each of its subtrees and can locate further nodes in the routing table [MM]. This allows a peer to search for a node in $\mathcal{O}(log N)$ steps.

2.4 Basic Overlay Modules

In Section 2.2 and Section 2.3 various approaches used in different overlay designs were introduced. Before discussing in more detail the main functionality principles and proposing a practical

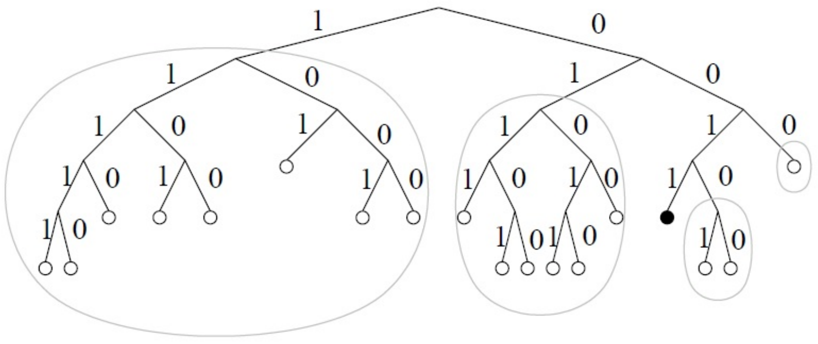


Figure 2.3: Position of the node 0011 in the routing table (Adapted from [MM]).

implementation of Geodemlia, observations concerning the commonalities of all overlays should be pointed out. Despite of the different routing table structures, every protocol follows some common rules.

In the structure of each and every protocol, some small similar modules can be found. As the proposed protocol has to be implemented in a modular way, the main types of these modules should be defined.

Some modules ensure the stabilization processes in the overlays (*Updates* = *UP*). Under stabilization, operations such as periodic updates, or join and leave procedures are meant. The *Routing algorithm* module (*RA*) implements the routing of the information. The *Routing table* module (*RT*) sorts the node's contacts according to some fixed algorithm and stores these contacts in the routing table. Some of these modules are passive as they do nothing else than just providing some place for information storage. The example of such modules is the *RT* module: the contacts are sorted and stored on a rule-based basis and no tasks are scheduled here. *RA* and *UP* belong to the active modules. They send requests and receive response messages and can be used by other modules.

Such a module-based structure of the overlay has a lot of advantages. If an overlay or an application is implemented in small modules, the developers can remove, exchange or extend them with an astonishing simplicity. These modules can be easily stopped and started again in order to analyse the reaction of the overlay on such events.

Chapter 3

Geodemlia: Main Functionality Principles

After providing in Chapter 2 some pieces of information about such widely used protocols as *Kademlia* and *Chord* and presenting the *PeerfactSim.KOM* simulator, the next logical step towards the practical implementation of *Geodemlia* is to explain the basic principles and the main operations in this protocol. For this purpose, a brief overview of the main idea and the basic operations in this location-based routing protocol are introduced in this chapter.

3.1 Main Concept of the Protocol

In comparison to *Chord* or *Kademlia*, *Geodemlia* is supposed to provide an effective location-based search using a search tag. This protocol allows to maintain locality and directionality, which means that neighbored peers store the neighbored location-based information and that the mapping of this information onto peers preserves the orientation in the multidimensional space [GSR+12], [KSS09]. The design of the protocol was inspired by the *Kademlia* protocol that was presented in Section 2.3. Such design makes *Geodemlia* protocol robust, which means that the performance of the overlay does not drop below a fixed threshold even if the churn rate is very high. The position-aware information gets periodically replicated onto k nodes that are nearest to the location that this information is associated with [GSR+12].

The main idea of *Geodemlia* is that each and every point has its own geographical coordinates $l_p = (\phi, \psi)$, where $\phi \in [-180°, 180°]$ and $\psi \in [-90°, 90°]$ [GSR+12]. Under this assumption and because most localization techniques use spherical coordinates, the peers are located on the sphere representing the earth. Each peer determines its own coordinates (longitude and latitude) using GPS, IP location service [Geo] or other techniques. As in *Chord* and *Kademlia*, each and every node in *Geodemlia* has not only an IP address but also an identifier $i \in [0, 2^{160} - 1]$. However this identifier does not prove to be an efficient tool, as it consumes memory space and does not demonstrate any advantages.

The distance $d(l_1, l_2)$ between two nodes with coordinates $l_1 = (\phi_1, \psi_1)$ and $l_1 = (\phi_2, \psi_2)$ can be computed using the *Haversine formula* [Hij16], [CN13], as shown below:

$$d(l_1, l_2) = d(\phi_1, \psi_1, \phi_2, \psi_2) = 2r \cdot \arcsin(\sqrt{a(\phi_1, \phi_2) + b(\phi_1, \phi_2) \cdot a(\psi_1, \psi_2)}), \tag{3.1}$$

where

$$a(\phi_1, \phi_2) = \sin^2(\frac{\phi_2 - \phi_1}{2}) \text{ [GSR}^+\text{12]} \tag{3.2}$$

and

$$b(\phi_1, \phi_2) = \cos(\phi_1)\cos(\phi_2) \text{ [GSR}^+\text{12]}. \tag{3.3}$$

In *Geodemlia* it is possible to store data objects and tags that are associated with fixed coordinates. The search can be carried out in a circular area within some fixed radius, but other forms of the search area are also allowed.

Geodemlia provides three main operations: *FIND_NODES(l_s, k, b)*, *AREA_SEARCH(l_s, r, s, b)* and *STORE(o, l_o)*. The structure of the routing table in *Geodemlia*, as well as these three operations are described in the next sections.

3.2 Structure of the Routing Table

One of the crucial differences between *Geodemlia* and *Chord* is the organisation of the routing table. As mentioned in the paper written by *Christian Gross* and *Dominik Stingl*, each peer divides the geographical space in n directions $j \in [0, n-1]$ based on the bearing angle $\theta \in [-\pi, \pi]$ clockwise from north to south [GSR$^+$12], as shown in Figure 3.1. The number of directions is usually set to 4, but any other number is also allowed.

For determining in which direction $j \in J$ a given peer lies, a bearing angle θ using the formula that is shown below must be calculated:

$$\theta = \arctan 2(c, d), \tag{3.4}$$

where

$$c = \sin(\phi_q - \phi_p) \cdot \cos(\psi_q) \text{ [GSR}^{+}\text{12]} \tag{3.5}$$

and

$$d = \cos(\psi_p) \cdot \sin(\psi_q) - \sin(\psi_p) \cdot \cos(\psi_q) \cdot \cos(\phi_q - \phi_p) \text{ [GSR}^{+}\text{12]} \tag{3.6}$$

with p and q being two peers.

Having the bearing angle, the direction j can be computed using the formula below [GSR^{+}12]:

$$j = \lfloor \frac{[(\theta + 2\pi) \ mod \ 2\pi] \cdot n}{2\pi} \rfloor. \tag{3.7}$$

Each direction is divided into buckets K_i^j in which, same as in *Kademlia*, the information about a fixed number of peers is stored [MM]. This information includes the geographical coordinates of the peer, the peer ID and the transport layer information, such as an IP address and a port. The size of the geographical area covered by a bucket grows exponentially with the distance of the bucket. As a result, fine-grained information about the nodes in the neighborhood and only scarse pieces of information about the nodes, that are further away, will be stored. Such routing table organisation ensures the probability of peer p to be a part of the peer q's routing table to be inversely proportional to the distance between these two nodes, so the routing converges within a logarithmic amount of steps [AAG^{+}].

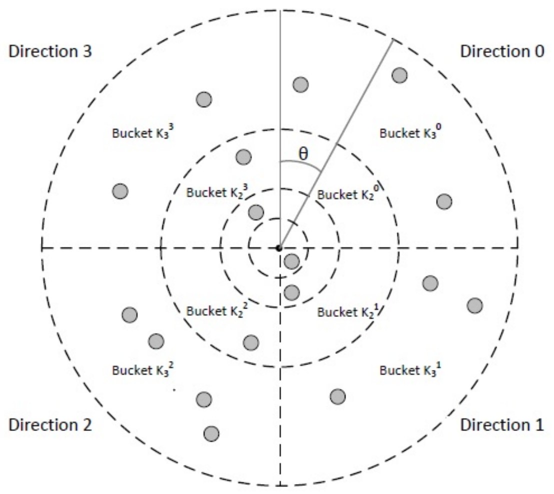


Figure 3.1: An example of how a routing table with 4 possible directions in Geodemlia can look like [GSR^{+}12].

3.3 FIND_NODES(l_s, k, b)

The *FIND_NODES(l_s, k, b)* request is the basic operation in *Geodemlia*, because other operations, such as *STORE(o, l_o)* and *AREA_SEARCH(l_s, r, s, b)*, as well as the updating mechanism, internally utilize the functionality of this method. This query includes the number of peers k that have to be returned, the location l_s, and the bloom filter b computed out of the IDs of the already found peers. If this request has been received, k nearest peers to the given location will be returned.

This operation is performed in the following way. If a peer p wants to find k nearest nodes, it first initializes a list C, where contacts to be queried will be stored, and checks its routing table in order to find k closest nodes to the location l_s. The found contacts are stored in the list C and the bloom filter b is computed from the IDs of these peers. After that, a fixed number of nodes from the list C will be contacted. The peers receiving this request do the same thing as the first node and search in its routing table for k nearest peers that still were not included in the bloom filter. Then the list of the newly found nodes is sent back to the querying node which will be merged with the contacts from the list C. From this list, a new bloom filter is calculated and sent as a part of a new *FIND_NODES(l_s, k, b)* request to the still not contacted peers. This procedure will be repeated as many times as necessary until no further nodes closer to the location l_s can be found.

3.4 STORE(o, l_o)

The *STORE(o, l_o)* request has the following parameters: o - object to be stored and the location l_o. If a node wants to store an object o in *Geodemlia*, it first must find k closest nodes with respect to the given location l_o. Afterwards, every peer from the set of k peers receives the command to store the object o on them.

3.5 AREA_SEARCH(l_s, r, s, b)

In contrast to *Kademlia*, *Geodemlia* provides a method for searching for the position-aware information within some fixed radius. The *AREA_SEARCH(l_s, r, s, b)* request includes a search location l_s, a search tag s, radius r within which the search must be proceeded, and the bloom filter b. During the *AREA_SEARCH(l_s, r, s, b)* process, two sets of peers must be found. The first set includes the nodes within the search radius r. The second set includes the peers that lie outside the query area but remain nearest to it. As a result, if the query area is too small, it is still possible to find the data objects.

The *AREA_SEARCH(l_s, r, s, b)* operation is performed as follows. If a peer p wants to issue this request, it uses the *FIND_NODES(l_s, k, b)* method. After the k nearest nodes were found, the peer p initializes two lists: a list C, where the peers to be contacted will be stored, and a list $\overline{C}$, where the peers will be stored, that have not yet been contacted. The found k peers are stored in list C, from which the bloom filter is calculated. Afterwards, all peers in the list C are contacted. After receiving *AREA_SEARCH(l_s, r, s, b)* request, the nodes check first whether they are inside the query area or not. If so, the stored objects will be included in the response message, as well as the additional k peers that are closest to the search area. After receiving the reply message, the peer p first checks, whether the newly found nodes have already been included to the list $\overline{C}$. The nodes, that match the nodes in the list $\overline{C}$, will be removed, otherwise, they will be added to the list C. The newly found objects will be stored in a result list. After that, the peer, that sent the response, is removed from the list C and added to the list of already contacted nodes. After that, a new bloom filter is calculated. Afterwards, the further peers in the list C are contacted until the list C is empty [GSR+12].

3.6 Process of Entering And Leaving the Overlay

In order to join an overlay, a peer p has to define its geographical coordinates and then contact a bootstrap node, it knows about. Afterwards, the bootstrap node is added to the routing table of the peer p. After that, the peer p sends the *FIND_NODES(l_s, k, b)* request to it. In this way, the peer p fills its routing table. Before a newly found contact is added to the routing table, the following steps should be done: (1) peer p should check whether a bucket, to which a newly found contact belongs, has less then k peers or not. (2) If there are less than k peers in the bucket, then a new contact is added to the end of the bucket. If the bucket is already full, then the least recently contacted peer should be found and contacted via a ping message. (3) If this peer responds to the query, then it is removed from the bucket and added to the end of the same bucket, and a newly found peer is added to the cache. (4) If this peer fails to reply, then it is removed from the bucket and a newly found peer is added to the end of the same bucket. This mechanism allows *Geodemlia* to update the routing information in such a way that the least recently contacted peers are at the beginning and the newly found peers are at the end of the bucket.

While leaving the overlay, it is not necessary to notify other nodes about this event, as the absence of this node will soon be discovered by the neighboring peers.

3.7 Updating the Routing Table

A peer p updates its routing table whenever it finds a new peer as a result of the requests, or by periodically querying for a random location. In this case, the steps from (1) to (4) from the previous section are carried out.

3.8 Modular Structure of Geodemlia

In the sections above we described the core procedures carried out by *Geodemlia*. Given this knowledge and the information about the modules presented in Section 2.4 of Chapter 2, the next step towards an implemention of the protocol is by defining first theoretically, which operation belongs to which module. This will help by planning the structure of the implementation.

Firstly, the active operations should be introduced. The process of entering the overlay, as well as the process of updating the routing table are active operations and they are a part of the *UP* module. Operations, such as *FIND_NODES*(l_s, k, b) and *AREA_SEARCH*(l_s, r, s, b), are also active, but they don't have any influence on the periodical updates of the routing table. These operations provide some services, that can be called by other modules. That is why they belong to the *RA* module.

The routing table also represents a module, but a passive one, as it just stores the contacts following some routing principle.

The *STORE*(o, l_o) operation and cache, in which the objects and the newly found nodes can be stored, also belong to passive modules.

Chapter 4 will show, whether an attempt to follow this modular division was successful or whether it was needed to change the notion of the modular overlay structure.

Chapter 4

Geodemlia: Implementation in Java

In the previous chapter, the theoretical understanding of main functionality principles of *Geodemlia*, as well as the modular structure of this protocol, were described. Using this knowledge, the next step can be done by introducing the main modules used for implementing the *Geodemlia* protocol.

4.1 Geodemlia: Basic Modules

For each overlay the ability to ensure the communication between hosts, to react on the connectivity changes and to handle the events coming from the inside and outside is very important. For the implemetation of *Geodemlia*, three main basic modules can be defined: *TransceiverModule*, *NetConnectivityModule* and *TaskSchedulerModule*.

The *NetConnectivityModule* provides methods to react on the churn property of the overlay. It is shown in Listing 4.1 and Listing 4.2 how method references are used by this module in order to specify the methods called, if a node goes on- or offline.

```java
/**Create new modules: transceiverModule for sending and receiving the messages,
 *                     netConnectivityModule for reacting on the connectivity changes,
**/                   TaskSchedulerModule for scheduling the events.

this.transceiverModule = new TransceiverModule(host, this.port);
this.netConnectivityModule = transceiverModule.getNetConnectivityModule();
this.taskSchedulerModule = new TaskSchedulerModule(host, this.netConnectivityModule);
this.netConnectivityModule.addNetConnectivityListener(this::online,this::offline);
```

Listing 4.1: Definition of the methods called if the connectivity of a node changes.

```java
// Called if a node goes online
private void online(){

        // If a node goes offline, it will be removed from the bootstrap manager
        GeoBootstrapManager.registerNode(this);
        GeoAnalyzer.registerNode(this.contact.getNodeTransInfo());
}

// Called if a node goes offline
private void offline(){

        // If a node goes offline, it will be removed from the bootstrap manager
        GeoBootstrapManager.removeNode(this);
        GeoAnalyzer.unregisterNode(this.contact.getNodeTransInfo());
 }
```

Listing 4.2: Methods called if the connectivity of a node changes.

The *TransceiverModule* ensures an efficient communication between peers as it is responsible for sending and receiving the messages. This module reacts on connectivity changes by stopping all actions, if a node goes offline. It can also be deactivated, if needed. Same as the *NetConnectivityModule*, the *TransceiverModule* makes use of Java 8. In Listing 4.3 is shown, how this module uses method references to define which method should be called after receiving a message and specifies the type of the messages the overlay is listening to.

```java
// Define the type of the messages to be listened to and the methods that should be called,
// if some particular message has been received
this.transceiverModule.addMessageListener(this::receiveFindKNodesRequest,
                                        FindKNodesRequest.class);
this.transceiverModule.addMessageListener(this::receiveFindKNodesReply,
                                        FindKNodesReply.class);
this.transceiverModule.addMessageListener(this::receiveAreaSearchRequest,
                                        AreaSearchRequest.class);
this.transceiverModule.addMessageListener(this::receiveAreaSearchReply,
                                        AreaSearchReply.class);
this.transceiverModule.addMessageListener(this::receivePingMessage, GeoPing.class);
this.transceiverModule.addMessageListener(this::receiveStoreRequest, StoreRequest.class);
```

Listing 4.3: Definition of the main message types the overlay is listening to and the functions called after receiving these messages.

This module provides the following methods: *sendUdp()*, *sendAndWaitUdp()*, *sendReplyUdp()*. *sendUdp()* enables sending a message to a specified receiver. *sendAndWaitUdp()* can be used to send a message and to listen to a response message. It is also possible to set the methods that will be called, if a message has been successfully sent or if the process failed, as well as the number of retransmissions and the delay interval. *sendReplyUdp()* ensures the sending of a reply message. The usage of these methods is represented in Section 4.2.2.

The *TaskSchedulerModule* provides methods to schedule the tasks. Equally to the *TransceiverModule*, this module stops all operations, if a node goes offline.

4.2 Other Modules Used in the Implementation of Geodemlia

In this section, the main classes that were developed are described, as well as the structure of some methods is presented. Selected classes are described in details. Over the rest of the classes just a brief overview is given. In Figure 4.1 the structure of this implementation is presented.

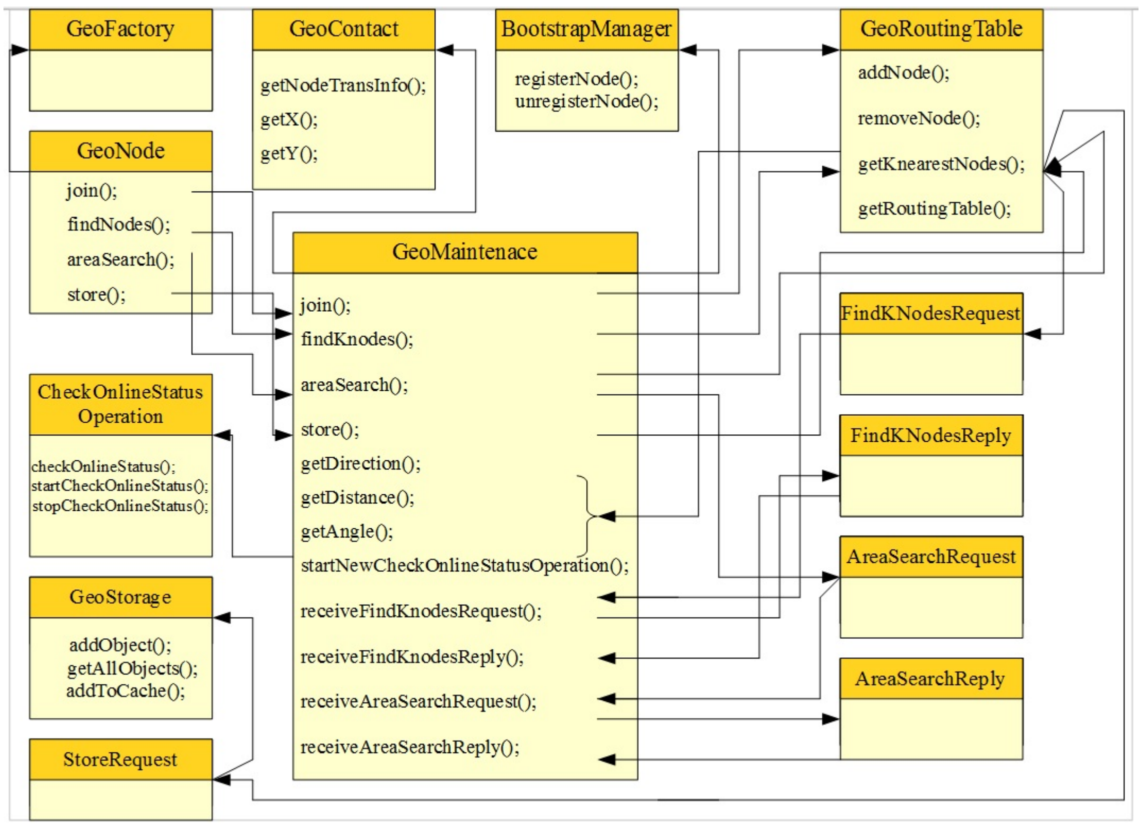


Figure 4.1: Geodemlia: the structure of the implementation.

Every overlay or application needs a tool that installs the applications on the hosts so that it can run. The **GeoFactory** fulfills this task by creating an instance of the overlay on a given node.

The **BootstrapManager** represents some kind of a container where all possible bootstrap nodes can be added, stored or removed. The *registerNode()*-method is used to add a node to the **BootstrapManager**. To remove a node from the **BootstrapManager** use the *unregisterNode()*.

The **GeoNode** class creates an instance of a node and serves as an interface between the *XML*-file and the inner structure of the overlay.

The **GeoContact** stores the *TransLayer* information, such as IP address and ID of the node, as well as the geographical coordinates of the node. In this implementation, the IDs are not used because they did not prove to be an efficient tool, as it has already been mentioned in Chapter 3. To get access to the IP address and port use *getNodeTransInfo()*-method. The *getX()* and *getY()*-methods are used to extract the node's coordinates.

The **FindKNodesRequest** class creates a request message used by the *findKnearestNodes()*-method. The **FindKNodesReply** class creates a reply message used by the *findKnearestNodes()*-method.

The **AreaSearchRequest** class creates a request message used by the *areaSearch()*-method. The **AreaSearchReply** class creates a reply message used by the *areaSearch()*-method.

The **StoreRequest** class creates a request message used by the *store()*-method.

In order to represent a storage, where the objects as well as the newly found nodes could be stored, the **GeoStorage** class was created.

The **GeoRoutingTable** class represents a container for storing the contacts of the nodes. For its implementation an *ArrayList* was used. Listing 4.4 presents the usage of the *ArrayList*. The outer list represents the number of directions, the middle list represents the number of buckets, and the inner list represents the position of a contact in some particular bucket. *ArrayList* was chosen for this implementation as it preserves the order of insertion of the elements. This plays an important role for the process of updating the routing table described in Section 3.6 and Section 3.7. The nodes, that take up the first positions, are the oldest nodes, that joined the overlay. The nodes at the end of the bucket are the nodes, that newly joined the network. This property of the *ArrayList* saves us some space, because no time stamps for every node must be created.

```
private ArrayList<ArrayList<ArrayList<GeoContact>>> routingTable;
private int numberOfDirections;
private int numberOfBuckets;

public GeoRoutingTable(GeoContact contact, int numberOfBuckets, int numberOfDirections){
        this.contact = contact;
        this.numberOfBuckets = numberOfBuckets;
        this.numberOfDirections = numberOfDirections;

        routingTable = new ArrayList<>();
        prepareRoutingTable();
}

```

```
/**
 * Initialise the routing table
 */
private void prepareRoutingTable(){
        for(int i = 0; i < this.numberOfDirections; i++){
            routingTable.add(new ArrayList<ArrayList<GeoContact>>());

            for(int j = 0; j < this.numberOfBuckets; j++){
                routingTable.get(i).add(new ArrayList<GeoContact>());
            }
        }
}
```

Listing 4.4: Geodemlia: ArrayList used as a routing table.

The main methods of this class are: *addNode()* which adds a contact to the routing table, *removeNode()* which removes a node from the routing table, *getRoutingTable()* which returns the routing table and *findKNearestNodes()* which returns the k nearest nodes to a given location and is described below in Section 4.2.2.

For maintaining the periodic operations responsible for updating the routing table the **CheckOnlineStatusOperation** class was developed. The main methods of this class are: *startCheckOnlineStatus()* which allows the procedure to be called from the outside, *checkOnlineStatus()* which is called, if the operation starts, and *stopCheckOnlineStatus()* which stops an operation.

The **GeoMaintenance** class is the most important active class in this implementation as it maintains the most import functions of the overlay: *join()*, *findKnodes()*, *areaSearch()*, *store()*.

4.2.1 findKNearestNodes()-method

This method can be either called from the outside or utilized internally as a part of the *join()*-, *areaSearch()*- or *store()*- procedures. As it has already been said in Section 4.2, this method returns a list of the k nodes closest to a specific location. In case of joining the overlay, these coordinates are the location of the node joining the network. The implementation of this method is shown in Listing 4.5.

```
/**
 * Return the list of k nearest nodes
 *
 * @param node
 * @param x
 * @param y
 * @param k
 * @return
 */
```

```java
public ArrayList<GeoContact> getKNearestNodes(GeoContact node,
                                              GeoMaintenance maintenance,
                                              long x,
                                              long y,
                                              int k){

       GeoAnalyzer.incFindNodesOperations();
       GeoAnalyzer.incOperations();

       ArrayList<GeoContact> kNearestNodes = new ArrayList<>();
       HashMap<Number, ArrayList<GeoContact>> distanceList = new LinkedHashMap<>();

       // Compute the direction and the bucket, in which this node could belong,
       // in order to  find the nearest nodes
       double d = maintenance.getDistance(node, x, y);
       int bucket = maintenance.getBucket(maintenance.maxX, maintenance.maxY, d);
       int dir = maintenance.getDirection(node, x, y);

       // If a node is inside the routing table area
       if(bucket != -1){
          // Iterate over all nodes in a bucket and calculate the distances
          // to (x, y)-coordinates
          for(int m = 0; m < routingTable.get(dir).get(bucket).size(); m++){
                ArrayList<GeoContact> nodes = new ArrayList<GeoContact>();
             double dd = maintenance.getDistance(this.routingTable.get(dir).get(bucket).
                                                                   get(m), x, y);

             if(distanceList.containsKey(dd) == true)
                distanceList.get(dd).add(routingTable.get(dir).get(bucket).get(m));
             else{
                nodes.add(routingTable.get(dir).get(bucket).get(m));
                distanceList.put(dd, nodes);
             }
         }
      }
     // Sort the distances
     List list = new ArrayList(distanceList.keySet());
     Collections.sort(list);

     // Find the first k nodes with the smallest distances
     for(int i = 0; i < list.size(); i++){
         for(int j = 0; j < distanceList.get(list.get(i)).size(); j++){
             if(kNearestNodes.size() < k){
                kNearestNodes.add(distanceList.get(list.get(i)).get(j));
             }
         }
     }
     return kNearestNodes;
}
```

Listing 4.5: Geodemlia: findKNearestNodes()-method.

First of all, the direction in which the given coordinates are located should be determined, because just the nodes from the same direction can be considered to be the closest ones. After that, the right bucket is calculated. Thereafter, the distances between this location and all nodes from the calculated bucket are computed. The results of this calculation are stored in the *LinkedHashMap*, where the keys are the distances and the values are the lists of the nodes. In this implementation, the *LinkedHashMap* is used instead of the *HashMap* in order to ensure the output to be deterministic. It is very important as we have to run the simulation a number of times and ensure that the data remains the same.

To find the *k* nearest nodes, the distances first need to be sorted in the ascending order. As it is impossible to sort the keys of the *HashMap*, a *List* was created and filled with the key set of the *LinkedHashMap*.After that, the *sort()*-method which can be found in the *Collections* package was used. Thereafter, the iteratation over the first *k* distances in the sorted list is done and the nodes associated to these keys are added to the *kNearestNodes*. The *kNearestNodes* list to be returned is of type *ArrayList* because the order of the insertion must be preserved to make the output deterministic, as in case of the *LinkedHashMap* described above.

4.2.2 Join/Leave the Overlay

In this section the *join/leave* procedure in *Geodemlia* is described. The implementation of this operation is shown in Listing 4.6.

```java
/**
  * Join the network
  *
  * @param contact
  * @param node
  */
public void join(){
        // Get the random bootstrap node in order to get access to the overlay
        GeoContact bootstrapNode = GeoBootstrapManager.getRandomNode(this.node);

        // Register in a bootstrap manager
        GeoBootstrapManager.registerNode(node);

        isOnline = true;

        // Register node in the analyzer
        EduAnalyzer.registerNode(this.contact.getNodeTransInfo());

        if (bootstrapNode == null){
            firstNode = true;
        }else{
            // If this is not the first node in the overlay, then add the bootstrap node to
                the routing table
            createRoutingTable(bootstrapNode);

            // Add the bootstrap node to the contact list
            contactList = new ArrayList<>();
            contactList.add(bootstrapNode);

            // Compute the bloom filter
            bloomFilter = new LinkedHashSet<>();
            bloomFilter = computeBloomFilter(contactList);

            // Send findKnearestNodes-request to the bootstrap node
            sendRequest(this.contact, bootstrapNode, bloomFilter,
                                      this.node.getGeoContact().getX(),
                                      this.node.getGeoContact().getY(), K);
        }
    }
```

Listing 4.6: Geodemlia: join()-method.

If a node joins the *Geodemlia* overlay, it first needs to contact a bootstrap node it knows about to learn about new peers and to fill its routing table. If this node is the first node in the network, then it will be registered in the *BootstrapManager* and a *firstNode* variable is set to true. Otherwise the *BootstrapManager* generates a random bootstrap node and returns it to the node that has just joined the overlay. As it has been already mentioned in Chapter 3, if some particular node goes offline, it does not inform other nodes about its abscence. The offline node is just removed from the *BootstrapManager* as it can be seen in Figure 4.2, so that new nodes do not receive the information about an inactive bootstrap node. After joining the overlay and getting the IP address of the bootstrap node, the node contacts the bootstrap node by using the *sendAndWaitUdp()*-method described in Section 4.1 and represented in Listing 4.7. This method creates the instance of the **FindKNodesRequest** class and sends the calculated bloomfilter as well as its own geographical coordinates with this message. We also specify, what happens, if the message is successfully sent, or if the process failed.

```
/**
  * Send k nearest nodes request
  *
  * @param sender
  * @param receiver
  * @param bloomFilter
  * @param x
  * @param y
  * @param k
  */
private void sendRequest(GeoContact sender, GeoContact receiver,
                                            Set<GeoContact> bloomFilter,
                                            long x,
                                            long y,
                                            int k){

        // Set the message ID
        kNodesMessageID += 1;
        this.transceiverModule.sendAndWaitUdp(new FindKNodesRequest(contact,
                                              receiver, kNodesMessageID,
                                              bloomFilter, x, y, k),
                                              receiver.getNodeTransInfo(),
                                              3, 10*Simulator.SECOND_UNIT,
                                                this::sendSuccessful,
                                                this::sendFailed);
}
```

Listing 4.7: Geodemlia: sendAndWait()-method.

After the bootstrap node receives the request message, the **receiveFindKNodesRequest**-method is called. This method calls the *findKNearestNodes*-method of the **GeoRoutingTable** class. In case, if this bootstrap node was the first node in the overlay and the *firstNode* variable was set to true, it computes the direction to the sender of the message and defines, whether this node is inside the bucket area or not. If this node lies inside the area, its contact will be added to the routing table of the bootstrap node and the *firstNode* variable will be set to false. This ensures that the routing table of the first node in the network will not stay empty. After the *kNearestNodes* list is sent back to the sender, the **receiveFindKNodesReply** is called. In this method a new bloom filter is computed, the newly found contacts are added to the list of nodes to be contacted, the routing table is updated. After the old contact list was merged with the newly received contacts, the request messages to the nodes, that has not already been contacted, are sent. This procedure lasts until no new nodes can be found.

4.2.3 CheckOnlineStatusOperation

The methods of this class are used to update the routing table. For this purpose, a random coordinates must be generated. After that, the node performes the *findKnearestNodes()*-operation. In Listing 4.8, a snipet of the **CheckOnlineStatusOperation** class code is presented. The full code can be seen in the implementation recorded on the disc.

```
// This method is called if the operation starts
private void checkOnlineStatus(long interval){

   if(!routingTable.getRoutingTable().isEmpty()){
      update = true;

      long x = (long)(Simulator.getRandom().nextDouble()*this.node.getMaxX());
      long y = (long)(Simulator.getRandom().nextDouble()*this.node.getMaxY());

      contacts = new ArrayList<>();
      bloom = new LinkedHashSet<>();

      contacts = this.routingTable.getKNearestNodes(this.ownContact, maintenance,
                                                   x, y, (int)this.node.getK());

      this.transceiverModule.sendAndWaitUdp(new UpdateMessageRequest(this.ownContact,
                                                  n, messageID, bloom, x, y,
                                                  (int)this.node.getK()), n.getNodeTransInfo()
                                                      , 2,
                                                  2*Simulator.SECOND_UNIT,
                                                  this::receiveUpdateRequest,
                                                this::updateRequestFailed);

      this.taskSchedulerModule.schedule(() -> checkOnlineStatus(interval), interval);
   }
}
```

Listing 4.8: Periodic operations in Geodemlia (part 1).

As it is not the same to randomly generate the numbers in the real world implementations and in the *PeerfactSim.KOM* simulator, the *Simulator.getRandom()*-method was chosen in order to produce random coordinates. This ensures the generatation of the results that are repeatable for every simulation.

After *k* nodes were found, the communication to these nodes via the *sendAndWaitUdp()*-method of the **TransceiverModule** is established and the *CheckOnlineStatusMessage* is sent. In the *sendAndWaitUdp()*-method, the *checkOnlineStatusSuccess()*-method is specified, which defines what has to be done, if the receiver is still online, and the *checkOnlineStatusFailed()*, which specifies what to do, if the receiver went offline. The use of these methods can be seen in Listing 4.9.

```
// Called if reply had been received
private void checkOnlineStatusSuccess(Message msg, TransInfo senderinfo, int commID){
        GeoAnalyzer.incSuccessfulLookups();

        GeoContact contactToAdd = this.routingTable.getRoutingTable().get(direction).
                                  get(bucket).get(position);

        if(position != this.routingTable.getRoutingTable().get(direction).
                       get(bucket).size()-1){
           this.routingTable.removeNode(this.routingTable.getRoutingTable().
           get(direction).get(bucket).get(position));
        this.routingTable.addNode(direction, bucket, contactToAdd);
        }
        this.onSuccess.run();
}

// Called if no reply had been received
private void checkOnlineStatusFail(Message msg, TransInfo senderinfo, int commID){
        GeoAnalyzer.incFailedLookups();
        this.routingTable.removeNode(this.routingTable.getRoutingTable().get(direction).
                                     get(bucket).get(position));
        this.onFail.run();
}
```

Listing 4.9: Periodic operations in Geodemlia (part 2).

If the receiver is still online, than it is removed from its old position in the bucket by using the *removeNode()* from the **GeoRoutingTable** class, and added to the end of the same bucket by means of the *addNode()*-mehtod from the same class. If the receiver is offline, its contact is removed from the bucket by using the method mentioned above.

4.2.4 Practical Modular Distribution of Classes

In Section 3.8 we defined theoretically, which operation belongs to which module. Now, having seen the structure of our protocol implementation and the functionality principles of some methods, let us define practically, which class belongs to which module.

The **GeoMaintenance** class is an active class that belongs to the *UP* module, as it maintains such methods as the *join()* and *startNewCheckOnlineStatusOperation()*, which stabilize the overlay. To the same module belongs the **CheckOnlineStatusOperation** class, in which the periodical lookups are scheduled.

The *findKNearestNodes()* and *areaSearch()* methods maintain actually the routing algorithm. For this reason, they belong to the *RA* module.

The **GeoRoutingTable** class belongs to the *RT* module, as it adds, stores and removes contacts as well as searches for specific contacts in the routing table.

The **GeoStorage** class also represents a passive module used just for the storing purposes.

All other classes, such as the classes, that create an instance of a request or a reply message, belong to the auxiliary modules that are used by the active modules, but are not of a vital importance.

So as it can be seen, we have succeeded to follow the modular distribution principles presented in Chapter 3.

Chapter 5

Evaluation of the Protocol

In this chapter, the evaluation of *Geodemlia* is given. The goal of this evaluation is to test the performance and the scalability of the protocol's implementation, as well as to analyse the influence of some parameters on the performance of the protocol. In Section 5.1, the simulator setup for testing purposes is described. Section 5.2 presents the results of the protocol tests.

5.1 Simulator Setup

5.1.1 Environmental Parameters

In order to test the *Geodemlia* implementation, first the environment should be set up. For this reason, the parameters shown in Table 5.1 are used. They remain the same throughout the whole testing

Table 5.1: Environmental parameters.

Parameter	Value
Region	Germany
Size of Area	70 km x 70 km
Number of Peers	1000, 10000
Simulation Duration	180 min

process. The only parameter that varies is the number of peers: this number varies from 1000 to 10000 peers to show that the overlay functions by a sparse/dense peer distribution.

5.1.2 System Parameters

The system parameters play an important role for the performance of the overlay. These parameters are presented in Table 5.2. The parameters in bold are used for some tests as default ones. To ensure

Table 5.2: System parameters.

Parameter	Value
Number of Buckets	2, 32, 64, 160
Bucket Size	2, **5**, 10, 20
k Nearest Nodes	2, 4, **8**, 16
Number of Directions	4
Number of Parallel Lookpus	4
Radius	10 km, **20 km**, 30 km
Range of the Randomly Generated Tags	10, 100

that the overlay functions by every random peer distribution, the tests were made for 5 randomly chosen seeds. All results presented below had been computed as an average for these 5 seeds. For testing purposes, a fixed number of directions, as well as a fixed number of parallel lookups were used.

5.1.3 Scenario

As already mentioned in Chapter 2, the *DAT*-file simulates a location-based application that generates the commands, such as *join()*, *store()* or *areaSearch()*. However, in order to test a more complex scenario, the *DAT*-file contains just the method calls without any parameters. After these methods have been called, the according methods from the **GeoNode** class come into play. They generate the following scenario.

The peers join the network in the first 60 minutes. In the next hour, the nodes perform the *store()*-command by randomly generating the coordinates and numbers that serve as tags. The range of the random numbers is varied from 10 to 100 tags to test the overlay with a sparse/medium file distribution. Each node performs 100 store()-actions using the generated coordinates and tags. All the tags that are generated are stored in an *ArrayList<String> tagList*. After the *store()*-operation was performed on all nodes, each node generates 30 random coordinates and takes 30 tags randomly found in the *tagList* and starts 30 *areaSearch()*-requests. This procedure lasts for the next 60 minutes.

To simulate the dynamics of the real world, in which the nodes go not only online but also offline, the peer churn parameter is enabled in the *XML*-file. The churn generator uses the KAD churn model described by Steiner et al. [SENB07]. The churn starts after the first 75 minutes of the simulation.

5.2 Results

In this section, the results of the testing process in form of the plotted graphs are presented. For generating the graphs, the plotting tool called *Gnuplot* [gnu] was used.

5.2.1 Average Number of Operations vs. Average Number of findKNearestNodes()-operations

As already mentioned in Chaper 3 and Chapter 4, the *findKNearestNodes()*-operation is considered to be a core operation in *Geodemlia*. To prove this assumption, the number of operations, as well as the number of *findKNearestNodes()*-operations was counted. The ratio between the number of operations and the number of *findKNearestNodes()*-operations is shown in Figure 5.1 and Figure 5.2. The plotted graphs show the average number of operations for 5 different seeds with the following default parameters: number of buckets = 32, bucket size = 5, k = 8, radius = 20 km.

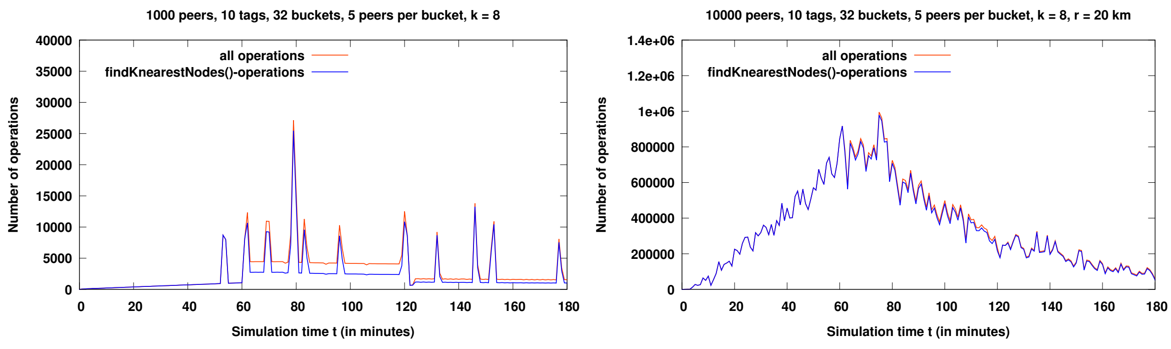
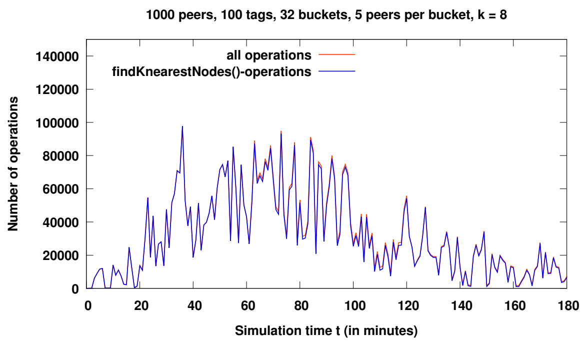


Figure 5.1: Average number of operations vs. average number of findKNearestNodes()-operations for tags = 10.

In the first two images, the number of possible tags was set to 10 and tested for 1000 and 10000 peers. In the next two plots, the number of possible tags was set to 100 and tested for the same number of peers.

Indeed, one can clearly see, that in the first 60 minutes of simulation only the *findKNearestNodes()*-operation is conducted, because it is the only action needed to join the network. But even after this step, the *findKNearestNodes()*-operation is executed almost all the time and takes almost the whole



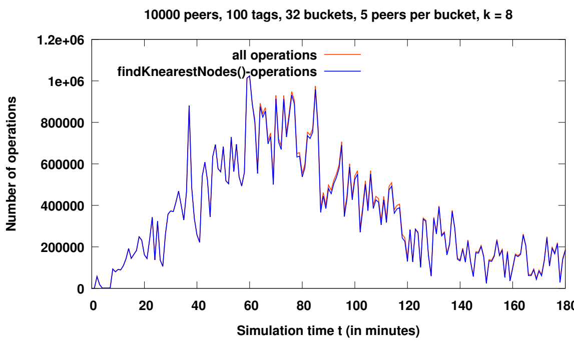


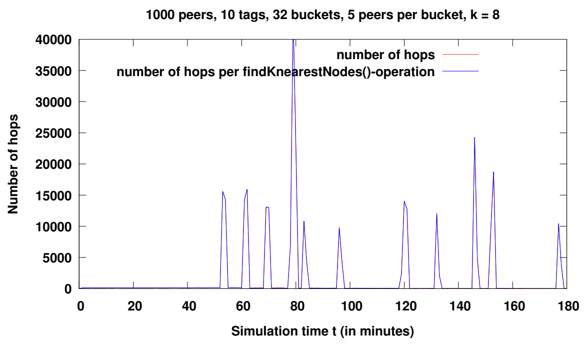
Figure 5.2: Average number of operations vs. average number of findKNearestNodes()-operations for tags = 100.

part of the executed procedures.

One can also clearly recognize, that after the 75th-80th minute, the number of *findKNearestNodes()*-operations, as well as the number of all operations decreases. This is the result of the active churn process that starts after the first 75 minutes of the simulation.

5.2.2 Average Number of Hops vs. Average Number of Hops per findKNearestNodes()-operation

The number of hops is one more proof of the assumption that the *findKNearestNodes()*-operation is the basic procedure in *Geodemlia*. In Figure 5.3 and Figure 5.4, the ratio between the number of hops and the number of hops per *findKNearestNodes()*-operation is represented. The parameters are the same as in Section 5.2.1.



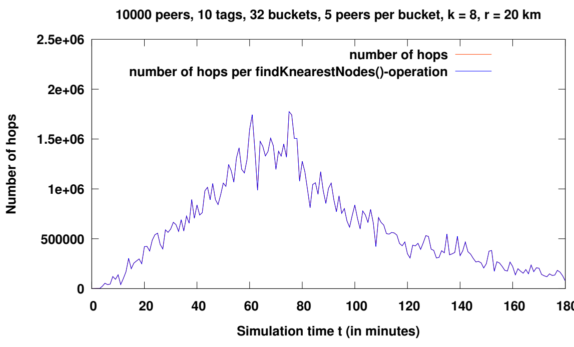
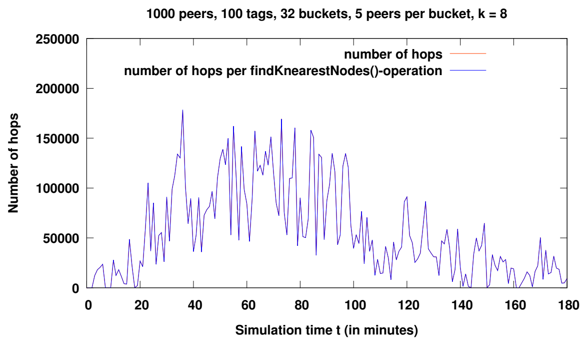


Figure 5.3: Average number of hops vs. average number of hops per findKNearestNodes()-operation for tags = 10.

The amount of hops is so large, that it is almost impossible to recognize the hops made while per-



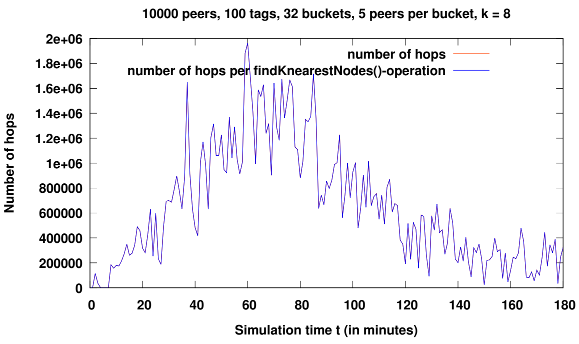


Figure 5.4: Average number of hops vs. average number of hops per findKNearestNodes()-operation for tags = 100.

forming other operations. These graphics show that the *findKNearestNodes()*-operation extremely dominates other operations in *Geodemlia*. Especially, in the first 60 minutes of joining the overlay. These plots also show the influence of the churn rate on the overlay activity, because the number af all hops and the number of hops per *findKNearestNodes()*-operation decreases after the first 75th - 80th minutes.

5.2.3 Number of Buckets

As written in a paper by *Christian Gross* and *Dominik Stingl* [GSR$^+$12], the number of buckets is originally set to 160. Such bucket partition is supposed to improve the performance of the protocol. However, by testing the overlay with 160 buckets, the unforeseen results came to light. In the next sections, the graphics reveal an unexpected influence of the number of buckets on various overlay events will be shown, as well as the according analysis will be given.

5.2.4 Number of Buckets vs. Number of Operations

In this section, the impact of the number of buckets on the number of operations is investigated. The parameters were set as follows: number of possible tags = 10, size of buckets = 5, k = 8, radius = 20 km. The results of this investigation are shown in Figure 5.5 and Figure 5.6. These tests were made for 1000 peers with 5 different seeds.

As the graphics show, an increase in the number of buckets leads to a decrease in the number of *findKNearestNodes()*-operations.

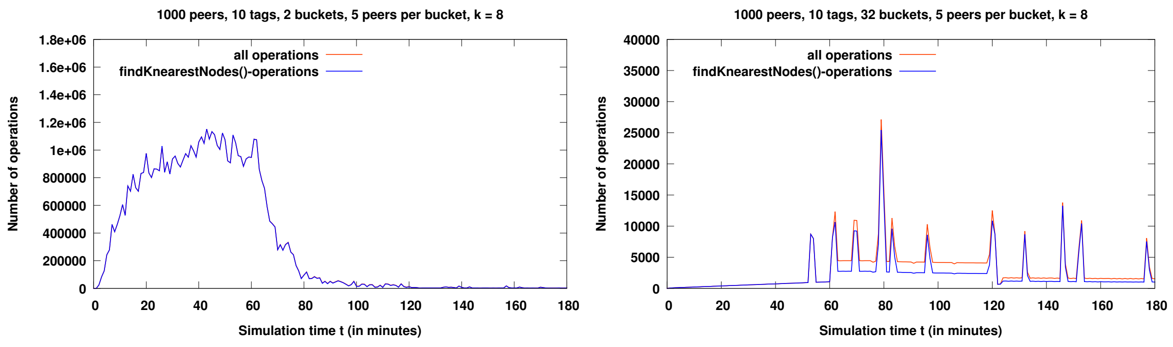


Figure 5.5: Average number of operations vs. average number of findKNearestNodes()-operations for 1000 peers with the number of buckets set to 2 and 32.

These graphics also show the impact of the active churn rate, as the number of operations decreases at about 75 - 80 minute of the simulation.

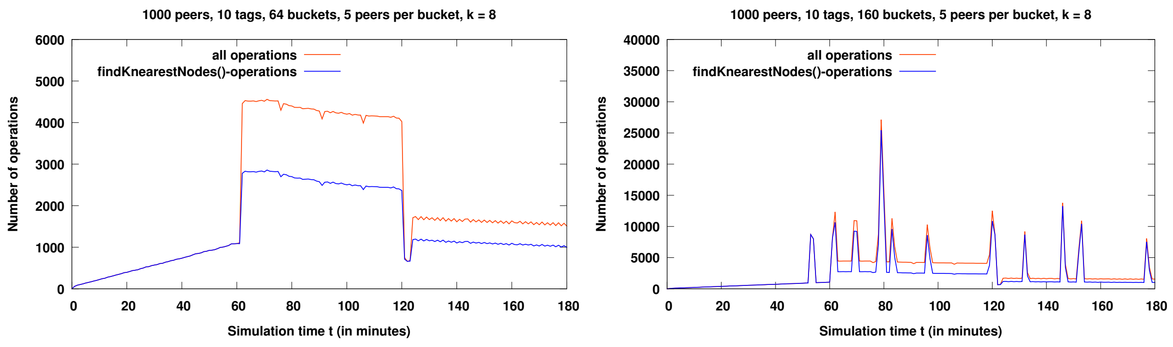


Figure 5.6: Average number of operations vs. average number of findKNearestNodes()-operations for 1000 peers with the number of buckets set to 64 and 160.

The same effect can be recognized by testing the overlay with the number of peers set to 10000. The Figures 5.7 and 5.8 demonstrate the results of this test.

There is only one explanation for this phenomenon. The larger the number of buckets, the larger is the possibility, that these buckets are empty or that they contain a very small amount of contacts. In order to imagine such situation, consider Figure 5.9.

The blue nodes belong to the routing table of the peer p while the orange marked nodes belong to the routing table of the peer q. The nodes marked with both colours belong to the routing tables of both nodes. From these graphics, it becomes clear, that if there are just a few nodes in the buckets and the peers have just one or two contacts in common or no contacts at all, than it is very difficult to find the nearest k nodes in order to perform other operations efficiently. This effect becomes even worse, if the

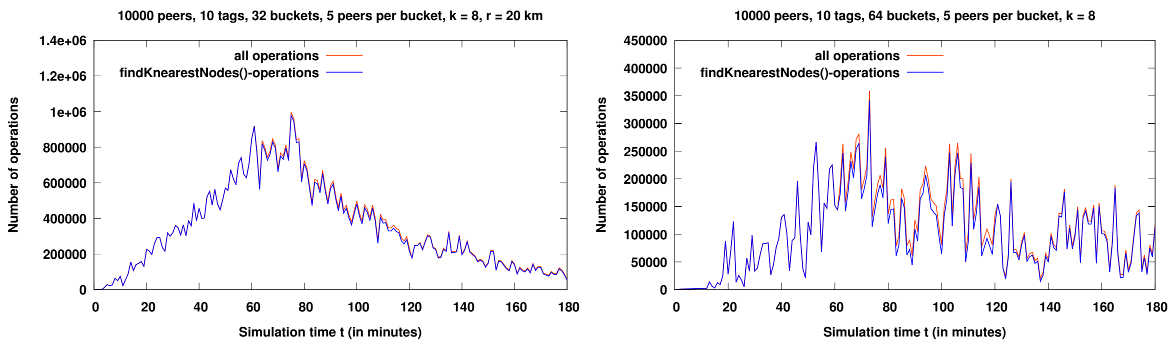


Figure 5.7: Average number of operations vs. average number of findKNearestNodes()-operations for 10000 peers with the number of buckets set to 32 and 64.

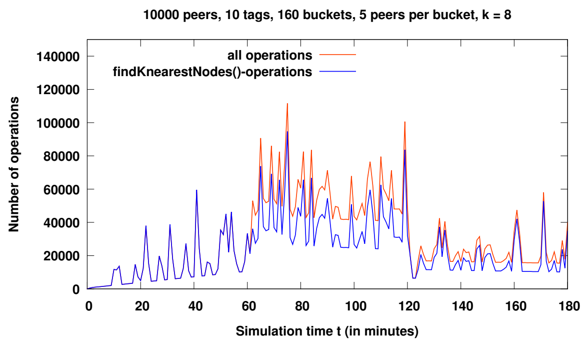


Figure 5.8: Average number of operations vs. average number of findKNearestNodes()-operations for 10000 peers with the number of buckets set to 160.

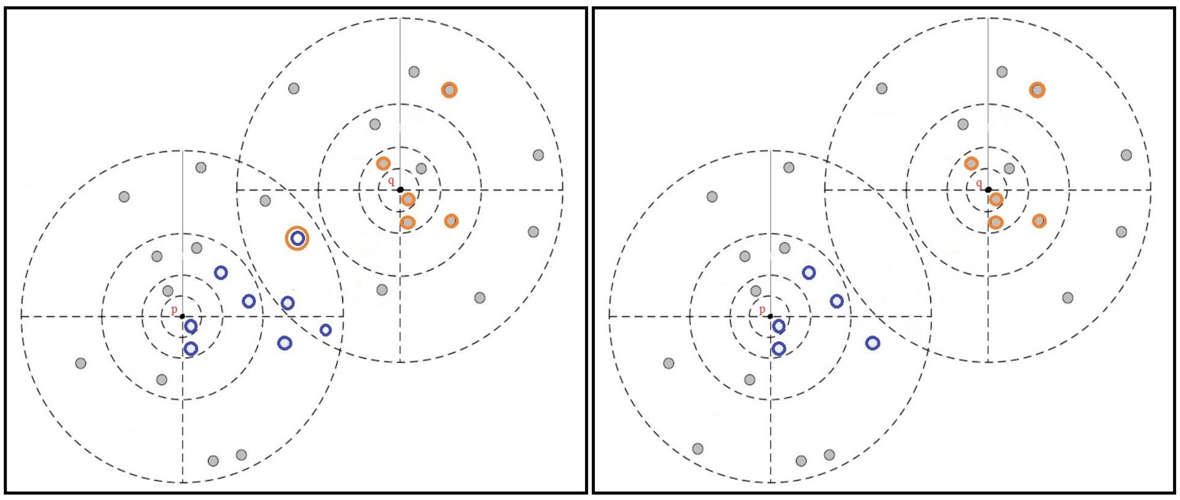


Figure 5.9: Possible routing table models. The original graphic was adopted from [GSR+12] and changed for explanation purposes.

peer distribution is sparse. So, even the number of peers set to 10000 in the 70 km x 70 km simulation area seems to be too small for this overlay to fill all buckets.

5.2.5 Number of Buckets vs. Number of Lookups

The same effect can be revealed while investigating the impact of the number of buckets to the rate of the successful lookups.

The usage of the term *lookups* in this particular part will be explained. Under lookups, all attemps to update the routing table are meant. Such updates can be the result of the periodic update operations or the result of the requests generated by the nodes. Under *successful lookups*, the cases in which some node in a bucket appears to be online and needs to be removed from its old position and added to the end of the bucket is meant.

In Figures 5.10 to 5.13, the average rate of the total lookups, as well as the average rate of successful lookups for 5 seeds is presented while testing the overlay with the number of peers set to 1000. The size of buckets and the value of k were set by default.

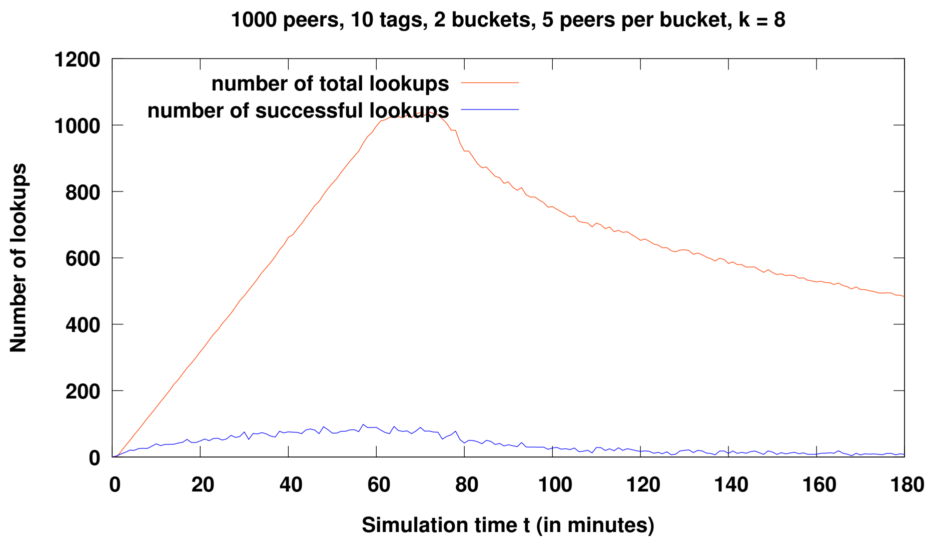


Figure 5.10: Average number of total lookups vs. average number of successfull lookups for 1000 peers with the number of buckets set to 2.

The size of the images was intentionally made larger than by other graphs, because the rate of the successful lookups converges almost to zero and for this reason can be very difficult to recognise on a small plot.

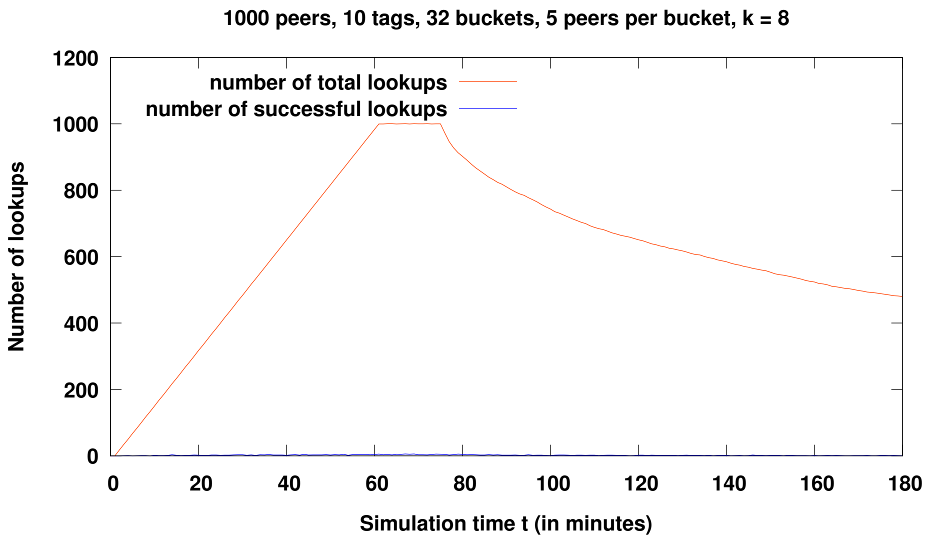


Figure 5.11: Average number of total lookups vs. average number of successfull lookups for 1000 peers with the number of buckets set to 32.

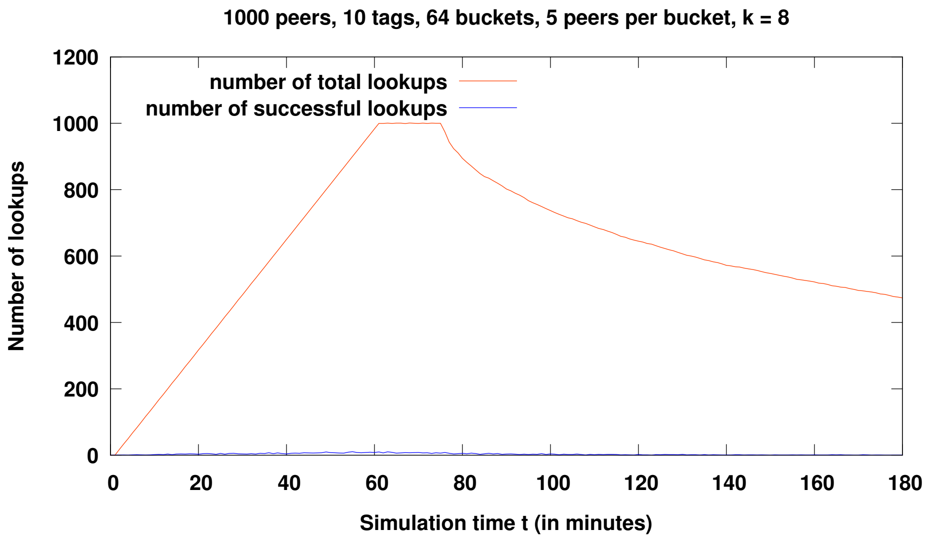


Figure 5.12: Average number of total lookups vs. average number of successfull lookups for 1000 peers with the number of buckets set to 64.

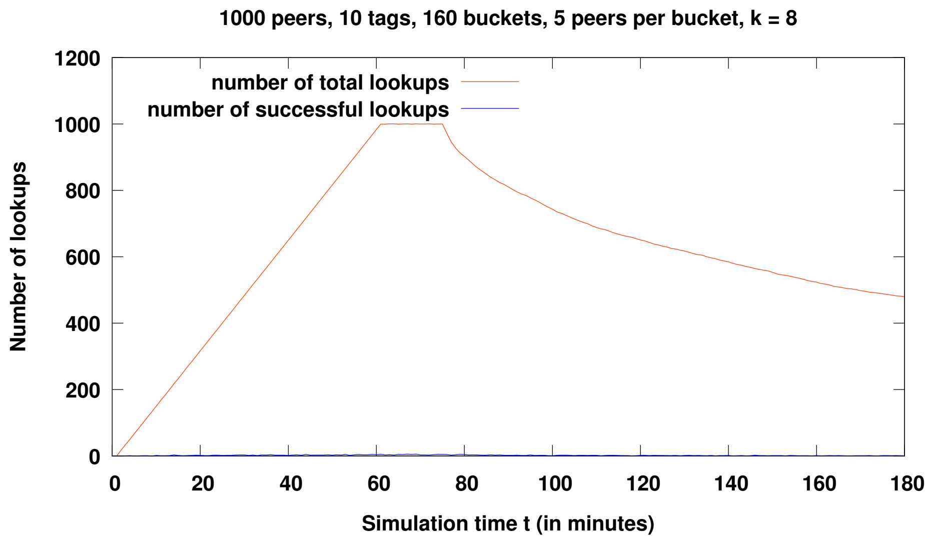


Figure 5.13: Average number of total lookups vs. average number of successfull lookups for 1000 peers with the number of buckets set to 160.

The same effect as in the previous plots can be recognized by testing the network with the number of peers set to 10000. The Figure 5.14 demonstrates the results of this test.

As mentioned in the previous section, a large amount of buckets leads to the increased possibility that many buckets remain empty, especially, by a sparse peer distribution. This leads to the situation in which just a few or possibly no *k* nearest nodes can be found while performing the *findKNearestNodes()*-operation. As a result, the updates can not be fulfilled efficiently, because the *findKNearestNodes()*-operation is the core method of all updates. As a consequence, the routing table is updated by accident and does not grow dynamically.

5.2.6 Number of Buckets vs. Number of Stored/Found Objects

After investigating the influence of the number fo buckets on the basic operation of this protocol, it is useful to examine the impact of this parameter to the performance of the *store()*-operation by inspecting the number of stored objects.

For this reason, the number of possible tags was set to 10. In these tests, the following default parameters were used: size of buckets = 5, k = 8, radius = 20 km. Consider Figure 5.15. The first tests were made in the overlay with 1000 peers.

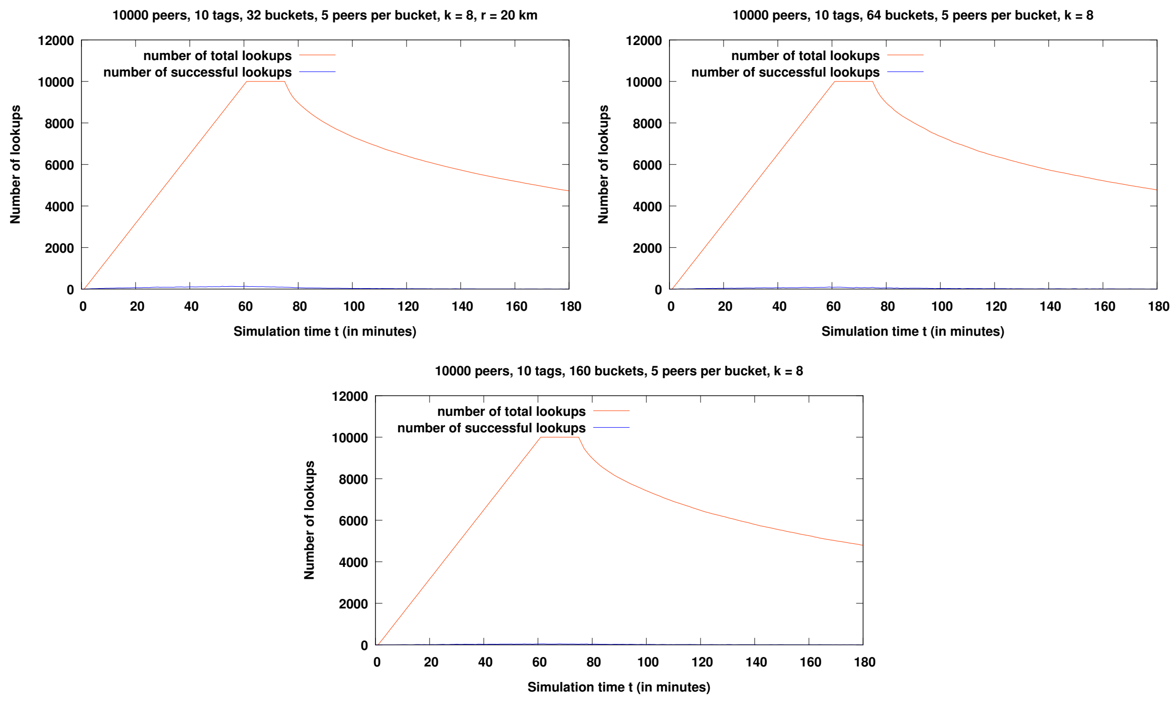


Figure 5.14: Average number of total lookups vs. average number of successfull lookups for 10000 peers with the number of buckets set to 32, 64, 160.

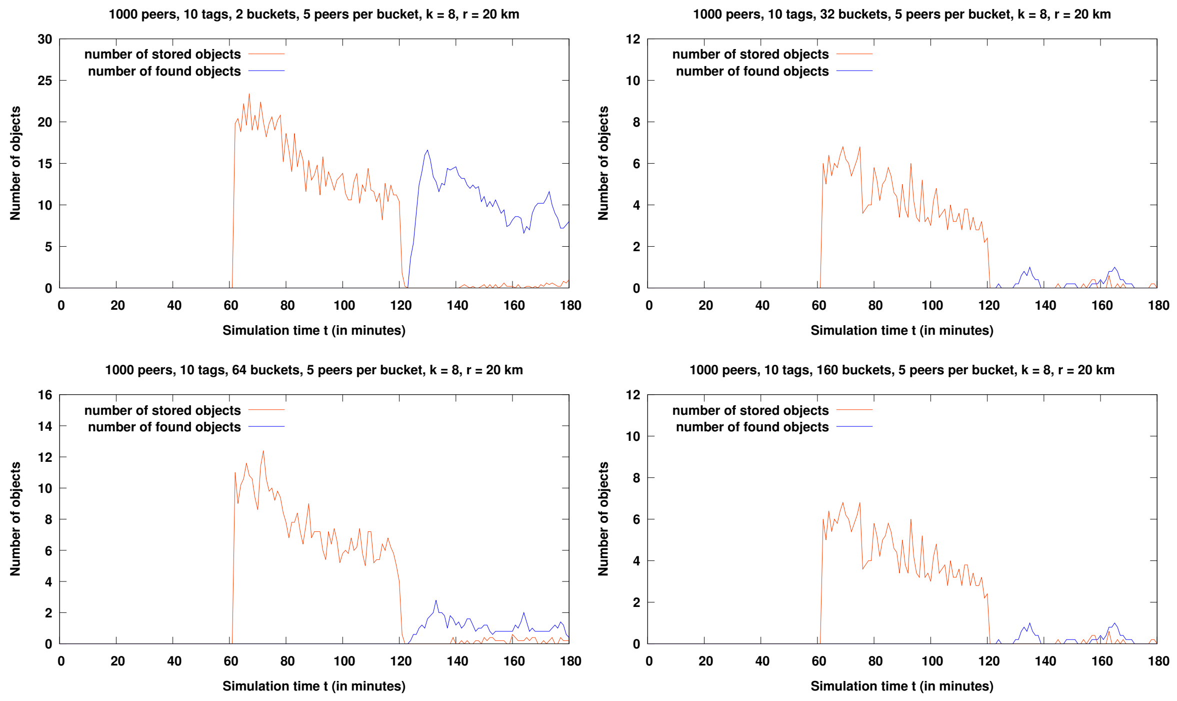


Figure 5.15: Average number of stored and found objects for 1000 peers with 10 tags and the number of buckets set to 2, 32, 64, 160.

In this Figure, the number of stored objects for 64 buckets is a little larger than the number of stored objects for 32 buckets, but it depends on the randomly generated coordinates. These graphics provide insight into the tendency as a whole: the number of stored objects decreases with the increasing number of buckets.

The second test was made for 10000 peers. Its results are presented in Figure 5.16.

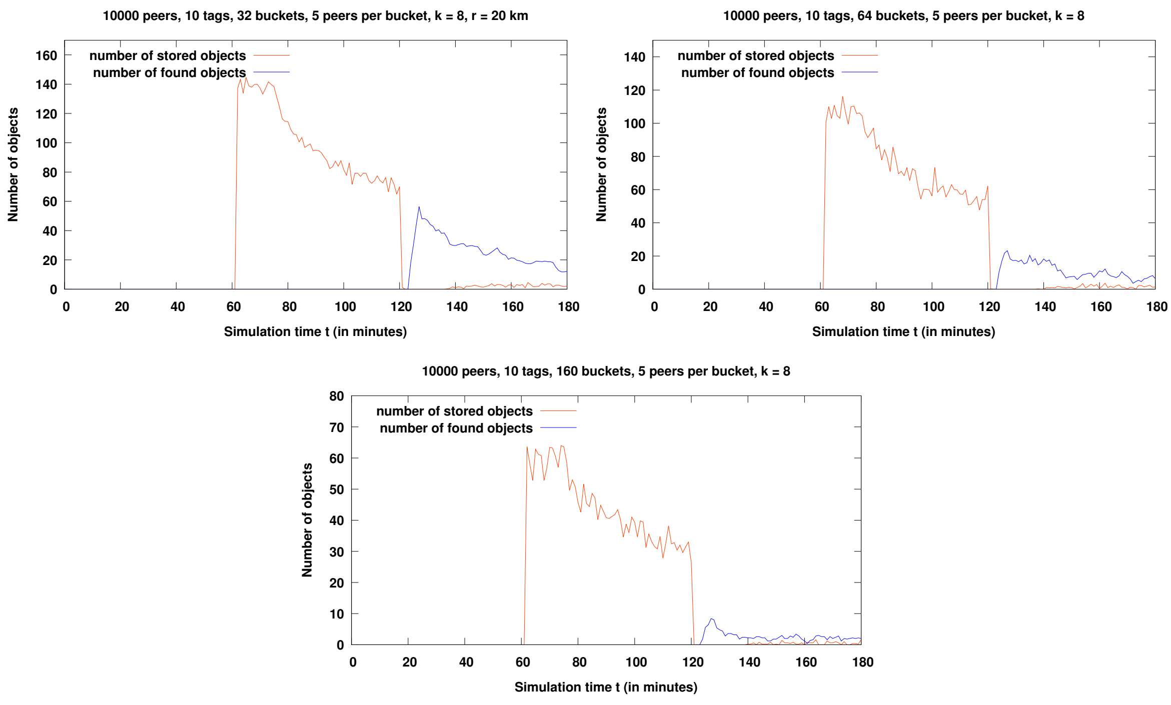


Figure 5.16: Average number of stored and found objects for 10000 peers with 10 tags and the number of buckets set to 32 and 64, 160.

Equally to the first tests, this one shows the same tendency. The larger the number of buckets, the smaller is the number of stored objects. The explanation for such behavior lies in the decreasing performance of the *findKNearestNodes()*-operation shown in Section 5.2.4, as this operation is the main tool of the *store()*-procedure. This happens as a result of an amount of buckets being empty.

Such a small amount of found objects can also be explained by reminding the testing scenario. The coordinates for the *areaSearch()* are generated randomly. This fact together with a sparse peer and file distribution, as well as the inappropriate number of buckets decreases the probability to find a given object.

One more parameter influences the number of stored objects: the churn rate. All these plots have some commonality: the number of stored objects decreases after the first 75 - 80 minutes of the simulation. This is the result of the active churn process.

Now, to ensure that the tendency observed in Figure 5.16 depends only on the number of buckets and not on the density of the file distribution, the range of the randomly generated tags will be set to 100. The results of this investigation are shown in Figure 5.17.

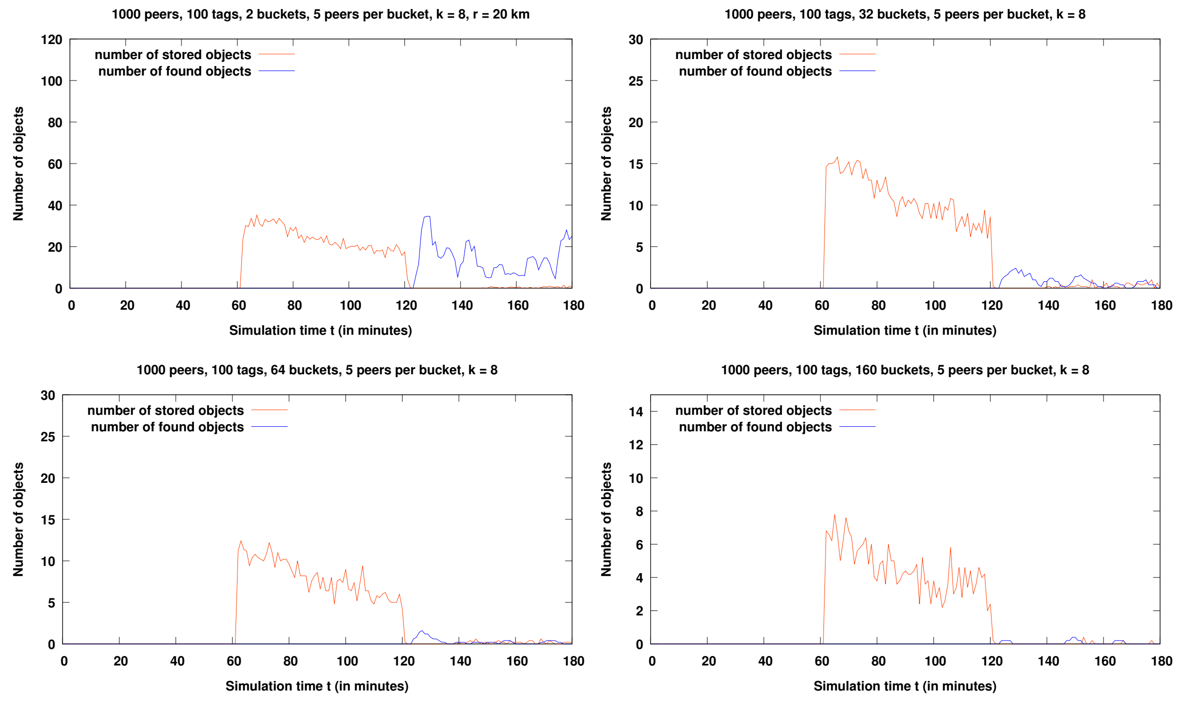


Figure 5.17: Average number of stored and found objects for 1000 peers with 100 tags and the number of buckets set to 2, 32, 64, 160.

The first two plots present the number of objects stored in the overlay with 1000 peers. The Figure 5.18 provides the same observations as in the previous plots but in the overlay consisting of 10000 peers. In these three plots, the same effect as for the set up with 10 tags can be observed.

Moreover, all these graphics reveal one more disadvantage of this overlay: it is extremely slow. One can clearly see in these plots, that some objects are stored even after the *store()*-operation is considered to be fulfilled. The reason for such behaviour is an enourmous amount of *findKNearestNodes()*-operations shown in Section 5.2.4 that are performed during the whole testing process as a part of the *store()*-, *areaSearch()*- or *update*-operations. Such a giant amount of operations is also the reason, why no tests for 10000 peers with 2 buckets could be done and no results of such tests could be presented.

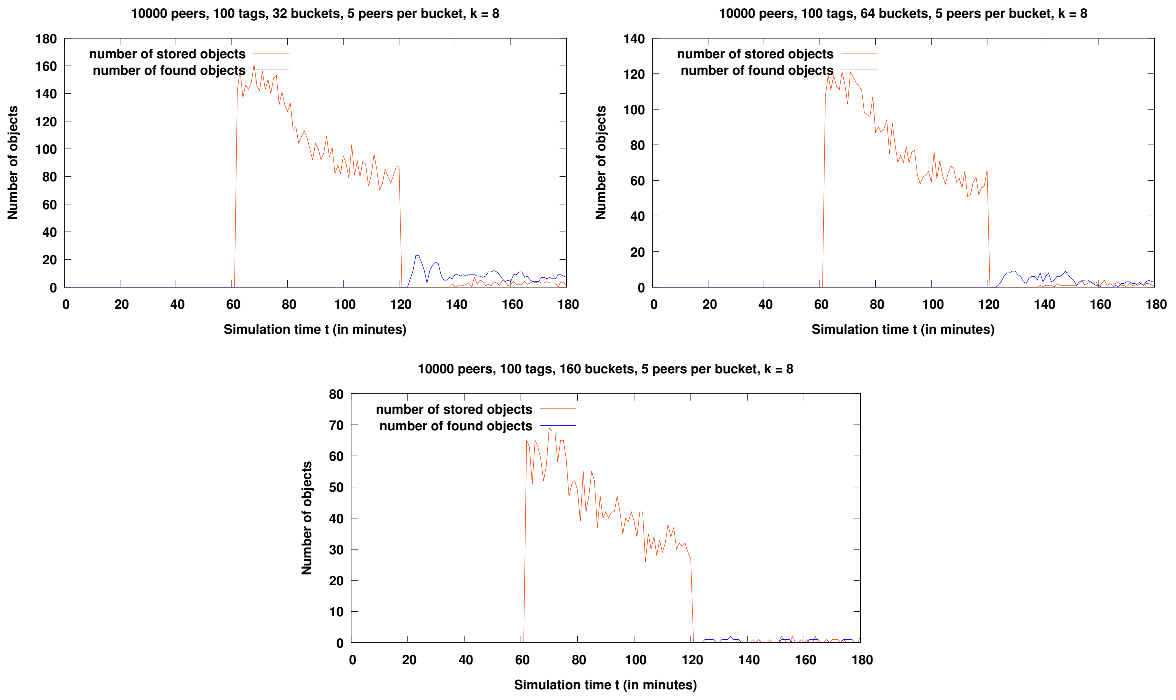


Figure 5.18: Average number of stored and found objects for 10000 peers with 100 tags and the number of buckets set to 32, 64 and 160.

5.2.7 Radius Size vs. Number of Found Objects

To investigate the impact of the radius size to the performance of the *areaSearch()*-operation, the following tests were done. The number of all found tags was counted, while varying the size of radius. These tests were conducted with the following parameters: 10000 peers, number of buckets = 32, bucket size = 5, k = 8. The radius size was set to 10 km, 20 km, 30 km. First, consider the results of the tests made with 10 possible tags shown in Figures 5.19 and 5.20.

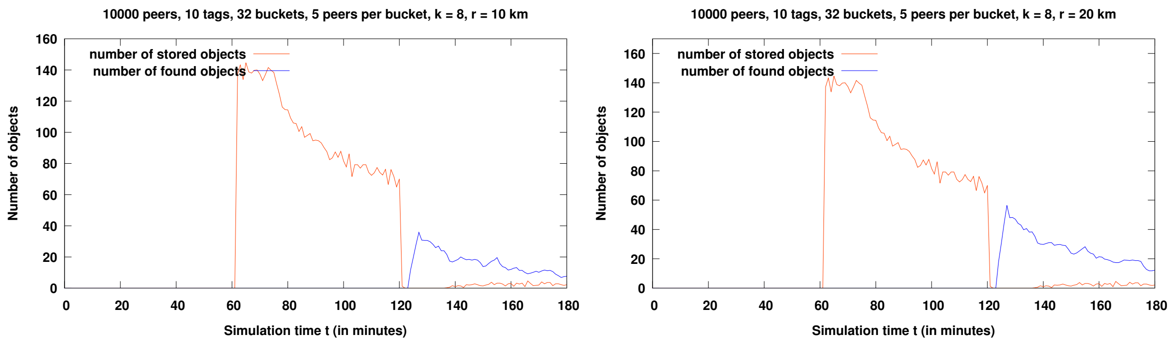


Figure 5.19: Average number of stored and found objects for 10000 peers with 10 tags and the radius size set to 10 km and 20 km.

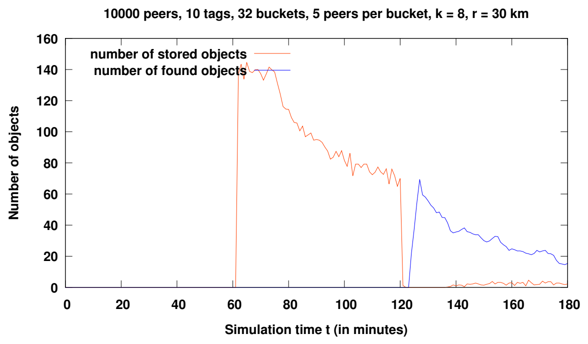


Figure 5.20: Average number of stored and found objects for 10000 peers with 10 tags and the radius size set to 30 km.

Now, consider the results of the tests shown in Figure 5.21. The tests were made with the same parameters but with the range of randomly generated tags set to 100.

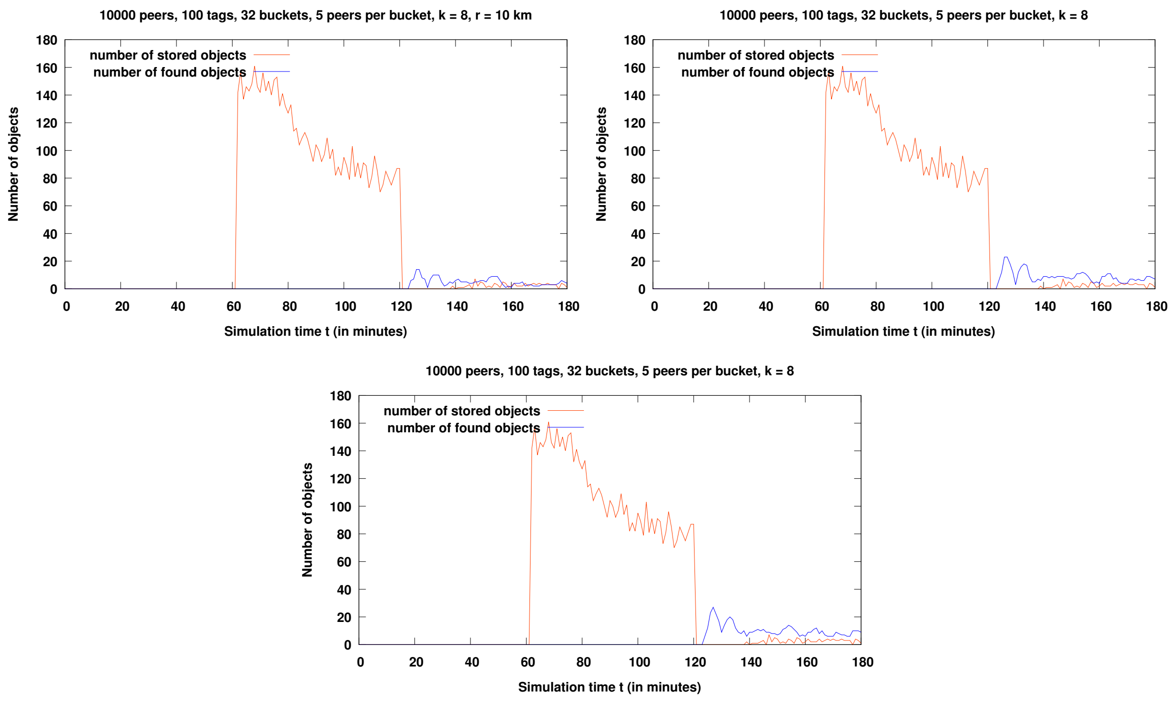


Figure 5.21: Average number of stored and found objects for 10000 peers with 100 tags and the radius size set to 10 km, 20 km, 30 km.

As expected, the larger the radius size, the larger is the number of found objects. The churn rate also has its impact on the number of found objects, because some peers, which stored a given object, go offline. It can be observed in the previous graphs, in which the number of found objects decreases as the simulation comes to the end.

To examine, how probable it is to find the objects within an area with various radii, consider Figures 5.22 and 5.23 in which the probability of finding the objects is presented.

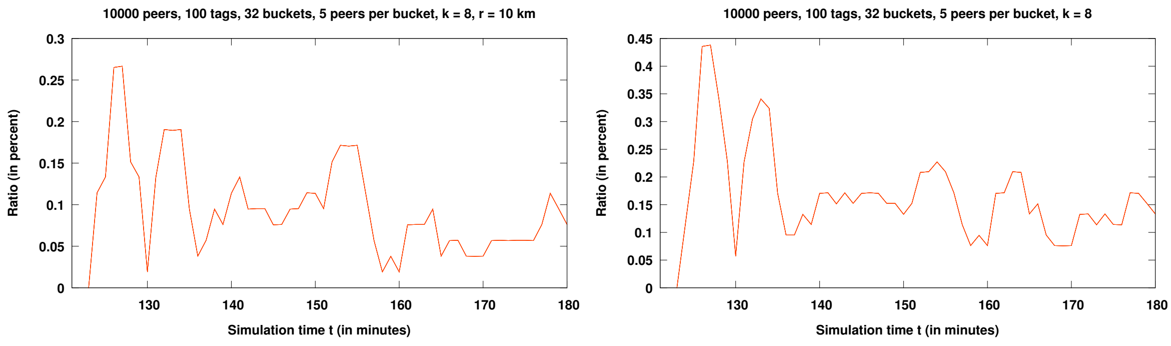


Figure 5.22: The probability to find objects in the overlay within an area with the radius size set to 10 km and 20 km. The number of peers was set to 10000 and the number of tags = 100.

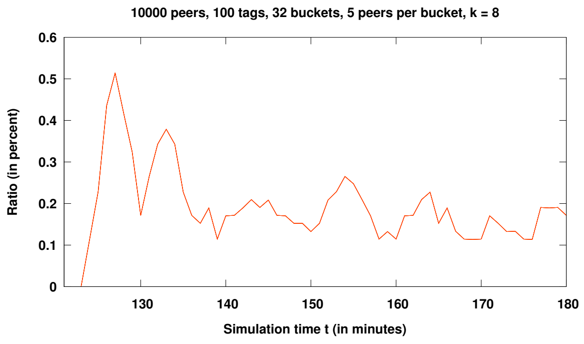


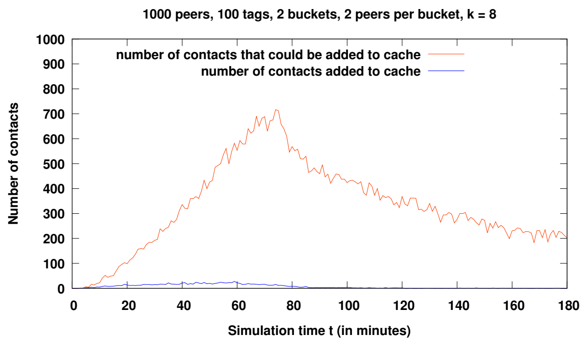
Figure 5.23: The probability to find objects in the overlay within an area with the radius size set to 30 km. The number of peers was set to 10000 and the number of tags = 100.

These graphics show that the probability to find some object in the overlay is, indeed, larger, if the radius of the search area is larger. This probability is, however, still very small as a result of the effects seen in the previous sections.

5.2.8 Size of Buckets

As already mentioned in Chapter 3, if a node finds a new contact while executing the *findKNearestNodes()*-request, it defines the bucket for this node and checks, whether this bucket is full or not. In case the bucket is full, the oldest node from this bucket must be contacted. If this node is still online, the newly found contact must be stored in cache.

In this section, an investigation of the impact of the bucket size on the number of newly found nodes to be contacted and stored in cache is provided. For this reason, the bucket size is varied from 2, 5, 10 to 20. The results are shown in Figures 5.24 and 5.25. The parameters for this investigation are as follows: the tests were made in the overlay consisting of 1000 peers with 5 different seeds, the value of k is set to 8 and the number of randomly generated tags equals 100.



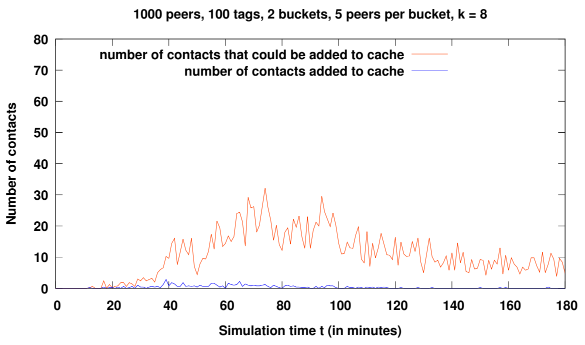
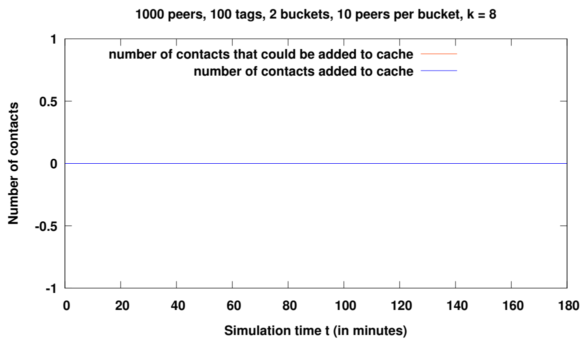


Figure 5.24: Number of newly found contact vs. number of contacts added to cache for 1000 peers and 100 tags with the size of buckets set to 2 and 5.



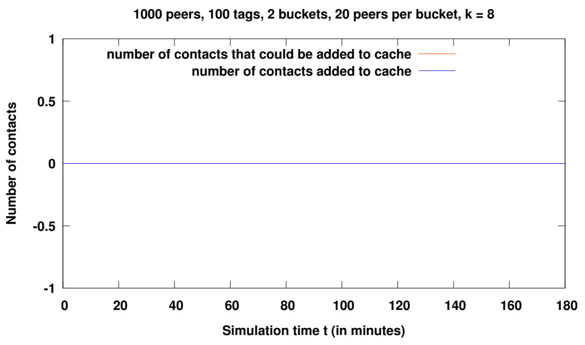


Figure 5.25: Number of newly found contact vs. number of contacts added to cache for 1000 peers and 100 tags with the size of buckets set to 10 and 20.

For these tests, the number of cases, when it was needed to contact to oldest node, was counted. The first plot shows, how many lookups were made, if the size of buckets was set to 2, and as a result how many newly found nodes could be added to cache. The second graph shows a significant reduction in the number of contact that could be possibly added to cache in comparison to the first plot. This happens, because more contacts can now be stored in a bucket. In this case, if a new contact is found, it can be in most cases directly stored within a bucket and there is no need to check, whether the oldest node is still online or not.

The last two graphs show clearly, that 10 and 20 peers per bucket reduce the number of lookups to zero, because there is enough storing space in each bucket.

As expected, the reduction of the bucket size leads to the increased number of nodes to be contacted in order to update the routing table.

5.2.9 Value of *K* Nearest Nodes vs. Number of Stored/Found Objects

In this section, the influence of the value for *k* parameter used in the *findKNearestNodes()*-operation is investigated. In Figures 5.26 and 5.27, the results of this investigation are shown.

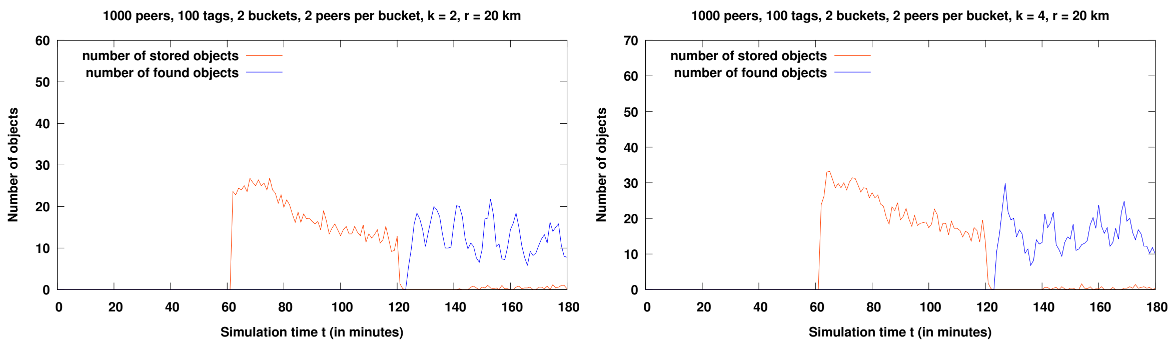


Figure 5.26: Number of stored and found objects for 1000 peers and 100 tags with k = 2, 4.

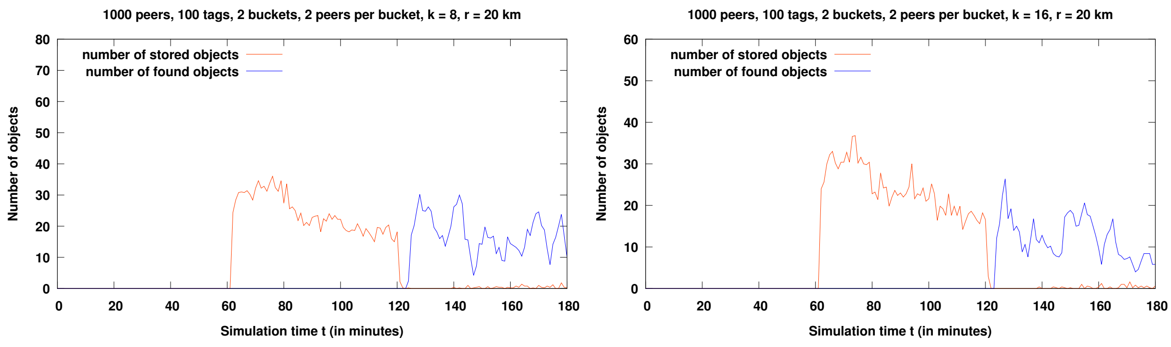


Figure 5.27: Number of stored and found objects for 1000 peers and 100 tags with k = 8, 16.

The tests were made in the overlay with 1000 peers. The range of the possibly generated tags was set to 100. The routing table of all peers consisted of 2 buckets. The size of the bucket was set to 2. The value for *k* was varied from 2, 4, 8 to 16.

In the first graph made for *k* = 2, the maximum number of stored objects lies slightly under 30 and the maximum number of found objects lies slightly above 20. The second plot shows, that these numbers increase to about 35 and 30, respectively, if *k* is set to 4. Such a behaviour can be explained as follows: on the one hand, the larger the number of *k* nearest nodes found by the *findKNearestNodes()*-operation, the larger is the number of objects stored on these nodes around a given location. As a result, there are more copies of the same object stored in the overlay which leads to an increased probability to find a given tag. On the other hand, the larger the number of *k* nearest nodes found by the *findKNearestNodes()*-operation, the larger is the probability that these nodes have some information about a given tag or about other nodes that have such information. This, in its turn, leads to an increased probability to find a given object while performing the *areaSearch()*-operation.

The last two graphs don't show any significant improvements. Logically, this leads to the conclusion that the larger value for *k* improves the results of the *store()*- and *areaSearch()*-operations, but it is insignificant, if the peer distribution is dense or the number of buckets is to large.

Chapter 6

Conclusion

In this chapter, the summary of this thesis is provided, as well as some ideas for future work are presented.

6.1 Summary

The goal of this bachelor thesis was to implement a routing protocol for location-based search called *Geodemlia* and to evaluate the performance of this overlay.

To develop this protocol, the event-driven simulator called *PeerfactSim.KOM* was used. As it is impossible to work with this simulator without a scarce understanding of its functionality, in Chapter 2 a description of its main properties and structure is provided. In order to understand what makes *Geodemlia* worth implementing, a brief overview of the basic routing protocols is given in the same chapter. The basic modules that can be found in each overlay are also presented there.

After that, the preparation for the implementation of the overlay begins. To implement an overlay, a good understanding of its basic principles is needed. For this reason, Chapter 3 was developed. In this chapter, a detailed theoretical discription of the protocol's functionality is given. After that, an attempt to assign the main operations to the basic modules described in Chapter 2, in order to develop a better structure of our implementation, was made.

After the theoretical part, the practical approach was provided in Chapter 4. In this chapter, the structure of the protocol's implementation in Java is presented, and the basic modules are described.

After that, the protocol's implementation has to be tested and the results have to be analysed. That is why, in Chapter 5, the results of the testing process are shown.

After the performed tests and analyses, the main advantages and disadvantages of this protocol must be pointed out.

One of the advantages of *Geodemlia* is that it does no't depend on a single central bootstrap node which can be a bottleneck in some situations. The whole overlay will not be laid down, in case one central node goes offline. One more advantage of *Geodemlia* is the mechanics of leaving the overlay. As it is not required to inform other nodes, if some node goes offline, the process of implementing the protocol becomes much easier in comparison to *Chord*.

The design of the routing table is, however, not one of the strenghts of this protocol. It sounds great theoretically, but in practice, such a bucket partition, as a result of an amount of buckets being empty intensified by a sparse peer distribution, leads to a poor performance. Under this circumstances, the updating mechanism also leads to an inefficient performance. All this leads to the conclusion that the routing tables do not expend dynamically, which makes it difficult to reach the nodes that are further away on the periphery.

Moreover, due to the large amount of *findKNearestNodes()*-operations, performed as a part of other operations, leads to the overload in the network what makes the overlay extremely slow. This means that the updating mechanism should be changed and it should take the churn factor and the load-balancing into consideration. The scalability of this protocol proved to be insufficient for a large amount of peers and complex scenarios.

6.2 Future Work

The idea of this protocol is theoretically very elegant, but practically can obviously be improved. First of all, the updating mechanism could be changed.

As an example, a node could update its routing table by regularly checking the status of a random node from a random bucket by just pinging it. This method does not allow the nodes to learn about new contacts while performing the update operations, but it does not overload the network with hundreds of thousands messages. For this reason, the proposed update mechanism could be extended with the already existing one in the following way: in the periods of time in which the network is very active, the proposed update method should be used; in the periods of time when the network is more passive, the existing update mechanism could be executed. It is surely not very easy to implement, but it could ensure that the nodes learn about new contacts while performing updates. Of course, all these changes should be made after an appropriate number of buckets was chosen.

One more idea for future work is associated with the coordinate system in *Geodemlia*. In its theoretical representation, as well as in the given implementation all peers were assumed to have static coordinates [GSR+12]. Assuming that some peers use mobile phones or tablets on the run leads to the conclusion that the whole overlay won't function properly in such situations. So managing this problem could be an excellent challenge.

For extending the *PeerfactSim.KOM* simulator, it could be possible to create new basic modules like *NetConnectivityModule* which could be used by different protocols simultaniuosly. Such basic modules could maintain the interactions between different overlays, for instance, by using messages.

Bibliography

[AAG$^+$] ABERER, Karl; ALIMA, Luc O.; GHODSI, Ali; GIRDYIJAUSKAS, Sarunas; HARIDI, Seif; HAUSWIRTH, Manfred: The essence of P2P: A reference architecture for overlay networks.

[Car] CARROLL, Lewis: *Alice's Adventures in Winderland, Available at: https://archive.org/details/AlicesAdventuresInWonderland, 2017.*

[CN13] CHOPDE, Nitin R.; NICHAT, Mangesh K.: Landmark Based Shortest Path Detection bz Using A* and Haversine Formula, 2013.

[FG13] FELDOTTO, Matthias; GRAFFI, Kalman: PeerfactSim.KOM. The Peer-to-Peer System Simulator – Community Edition, 2013.

[Geo] GEOBYTES: *IP Locator Service, Available at: http://geobytes.com/iplocator/, 2017.*

[gnu] *Gnuplot, Available at: http://www.gnuplot.info, 2017.*

[GRS$^+$13] GROSS, Christian; RICHERZHAGEN, Bjorn; STINGL, Dominik; WEBER, Jan; HAUSHEER, David; STEINMETZ, Ralf: GeoSwarm: A multi-source download scheme for peer-to-peer location-based services. In: *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on* IEEE, 2013, S. 1–10.

[GSR$^+$12] GROSS, C.; STINGL, D.; RICHERZHAGEN, B.; HEMEL, A.; STEINMETZ, R.; HAUSHEER, D.: Geodemlia: A Robust Peer-to-Peer Overlay Supporting Location-Based Search, 2012.

[Hij16] HIJMANS, Robert J.: Introduction to the "geosphere" package, 2016.

[JFK12] JAMES F. KUROSE, Keith W. R.: *Computer networking: A top-down approach - 6th ed.* Pearson, 2012

[KSS09] KANTERE, V.; SKIADOPOULOS, S.; SELLIS, T.: Storing and Indexing Spatial Data in P2P Systems, 2009.

[KW06a] KNOLL, Mirko; WEIS, Torben: Optimizing locality for self-organizing context-based systems. In: *Self-Organizing Systems*. Springer, 2006, S. 62–73.

[KW06b] KNOLL, Mirko; WEIS, Torben: A P2P-Framework for Context-Based Information. In: *1st International Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI) at Pervasive*, 2006.

[LYH13] LEE, Choong C.; YUN, Haejung; HAN, Dongho: Gossip-based Aggregation in Large Dynamic Networks. In: *Journal of Electronic Commerce Research* 14 (2013), Nr. 3.

[MM] MAYMOUNKOV, Petar; MAZIÈRES, David: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.

[Pee] PEERFACTSIM.KOM: *PeerfactSim.KOM - Community Edition, Available at: http://peerfact.com/simulator-details, 2017.*

[SENB07] STEINER, Moritz; EN-NAJJARY, Taoufik; BIERSACK, Ernst W.: A Global View of KAD, 2007.

[SMK$^+$01] STOICA, Ion; MORRIS, Robert; KARGER, David; KAASHOEK, M. F.; BALAKRISHNAN, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: *Proc. of ACM SIGCOMM '01*, 2001.

[Ste11] STEINMETZ, Prof. Dr.-Ing. R.: *Documentation for PeerfactSim.KOM*. 2011.

[TU] TU, Darmstadt: *Chord, Kademlia, Available at: peerfact.kom.e-technik.tu-darmstadt.de/de/overview/layered-architecture/overlay-layer, 2017.*

[Wika] WIKIPEDIA: *Chord, Available at: https: //en.wikipedia.org/wiki/Chord_(peer-to-peer), 2017.*

[Wikb] WIKIPEDIA: *Client/Server/Modell, Available at: https://en.wikipedia.org/wiki/Client-Server-Modell, 2017.*

[Wikc] WIKIPEDIA: *Kademlia, Available at: https: https://en.wikipedia.org/wiki/Kademlia, 2017.*

[Wikd] WIKIPEDIA: *Peer-to-Peer, Available at: https: https://en.wikipedia.org/wiki/Peer-to-peer, 2017.*

[Woo] WOOLF, Nicky: *DDoS attack that disrupted internet was largest of its kind in history, experts say, Available at: https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet, 2017.*

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 28.February 2017 Olga Batiukova

Please add here

the DVD holding sheet

This DVD contains:

- A *pdf* Version of this bachelor thesis
- All LaTeXand grafic files that have been used, as well as the corresponding scripts
- **[adapt]** The source code of the software that was created during the bachelor thesis
- **[adapt]** The measurment data that was created during the evaluation
- The referenced websites and papers