
# Implementation of a Mesh-based Overlay for Location-based Search

Bachelor Thesis

by

Mustafa Yilmaz

born in
Düsseldorf

submitted to

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

Juli 2015

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

# Abstract

In this thesis, I will present a novel location-based peer-to-peer overlay which is called GeoMesh. It supports location-based search in multidimensional space. The main focus is to create and implement an overlay that can handle the requirements of location-based overlays such as finding the $k$-closest nodes to a given location or finding all nodes respectively data objects in a certain area. Additionally, it should preserve some important properties such as locality and directionality which will be discussed and explained in the upcoming chapters. This overlay is currently implemented in the PeerfactSim.KOM overlay network simulator, hence implementation details will be presented. The overlay is scalable, efficient and stable in terms of churn.

# Acknowledgments

I want to thank my family, my fellow student Erol Yildirim and all the people who have supported me.
I especially want to thank Tobias Amft who gave me feedback and answered my questions during my work on this thesis.

# Contents

**List of Figures** ix

**List of Tables** xi

**1 Motivation** 1
- 1.1 Peer-to-Peer Overlays . . . 1
- 1.2 Location-based Overlays . . . 1
- 1.3 Overview . . . 2

**2 Related Work** 3

**3 Design of Mesh** 5
- 3.1 Overview . . . 5
- 3.2 Requirements . . . 6
- 3.3 Positioning and Distance . . . 6
- 3.4 Routing Table . . . 6
- 3.5 Find k-Closest Nodes . . . 8
- 3.6 Area Search . . . 10
- 3.7 Join and Leave . . . 11
- 3.8 Routing Table Maintenance . . . 12
- 3.9 Finger Table . . . 12
- 3.10 Conclusion . . . 13

**4 Implementation** 15
- 4.1 Identifier, Contact Information and Position . . . 15
- 4.2 Routing Table . . . 18
- 4.3 Messages and Operations . . . 19
- 4.4 Node . . . 22

**5 Evaluation** 25
- 5.1 Routing Table Size and Search Results . . . 25
- 5.2 Metrics . . . 27

5.2.1 Scalability . . . 27
5.2.2 Stability and Efficiency . . . 27
5.2.3 Area Search . . . 28
5.3 Setup . . . 28
5.3.1 Scalability . . . 29
5.3.2 Stability and Efficiency . . . 29
5.3.3 Area Search . . . 29
5.4 Results . . . 30
5.4.1 Scalability . . . 31
5.4.2 Stability and Efficiency . . . 33
5.4.3 Area Search . . . 36
5.5 Analysis of the Results . . . 38
5.5.1 Scalability . . . 38
5.5.2 Stability and Efficiency . . . 38
5.5.3 Area Search . . . 38

**6 Conclusion** **39**
6.1 Future Work . . . 39

**Bibliography** **41**

# List of Figures


3.1 Bearing angle . . . . . 7
3.2 Routing table structure ($k = 8$) . . . . . 7
3.3 Node $n$ initiates an iterative search for the k-closest nodes to position $P$. . . . . 9
3.4 Node $n$ returns the k-closest nodes to position $P$ that are not in the *identifier* set $S$. . . 9
3.5 Node $n$ initiates an area search with radius $r$, with parameter $k$ which should first find the $k$-closest nodes to $P$. . . . . 10
3.6 Node $n$ returns nodes if his nodes lie within the search area which is defined by the radius $r$ around point $P$. . . . . 11
3.7 This function returns true if $P_1$ is a point in an area defined by point $P$ and radius $r$ . . 11
3.8 Node $n$, periodically, executes the $fix\_fingers$ procedure to update the respective finger table entry, given a direction $i \in \{0,1,2,3\}$ which describes the bearing angle of $i \cdot 90°$ from $n$'s position . . . . . 13
3.9 Given a starting point $P$, a direction $i \in \{0,1,2,3\}$ and a distance $d$, this function returns the end point, which is $d$ away and has a bearing angle of $i \cdot 90°$ to position $P$ on a sphere with Radius $R$ (see [Bea]). . . . . 13

5.1 Success ratio when $s = 16$ . . . . . 26
5.2 Success ratio when $s = 24$ . . . . . 26
5.3 Success ratio when $s = 32$ . . . . . 26
5.4 Success ratio when $s = 40$ . . . . . 26
5.5 Average Hop Count of Find $k$-Closest Nodes Operations . . . . . 31
5.6 Average Response Time of Find $k$-Closest Nodes Operations . . . . . 31
5.7 Overlay Traffic Per Peer . . . . . 32
5.8 Number of Present Peers Per Minute . . . . . 33
5.9 Success Ratio of Find k-Closest Nodes Operation Per Minute . . . . . 33
5.10 Success Ratio of Finding at Least One Node Per Minute . . . . . 34
5.11 Hop Count of Successful Find k-Closest Nodes Operations Per Minute . . . . . 34
5.12 Response Time of Successful Find k-Closest Nodes Operations Per Minute . . . . . 35
5.13 Number of Present Peers Per Minute . . . . . 36
5.14 Success Ratio of Area Search Operations Per Minute . . . . . 36
5.15 Number of Present Peers Per Minute . . . . . 37

5.16 Success Ratio of Area Search Operations Per Minute . . . . . 37

# List of Tables

# Chapter 1

# Motivation

## 1.1 Peer-to-Peer Overlays

A P2P system consist of peers which are participants of an overlay network. An overlay network is a network built on top of another network. Unlike the basic client-server model in which a central server provides services, ressources or data, each peer in in a P2P overlay network act as a server and a client at the same time. This means that peers in overlay networks are are able to retrieve and provide data from and for peers. Peer-to-peer overlays can be divided into two categories, the centralized and decentralized overlays. In decentralized overlays each node queries nodes to find data in the system, while nodes in centralized overlays query a central indexing server to find the respective node which provides the desired data. Decentralized overlays can be further subdivided into two categories, the structured overlays and the unstructured overlays. In unstructured overlays, data is searched using keywords. These types of overlays have some major drawbacks in terms of scalability and efficiency. In structured overlays, on the other hand, these problems do not occur. The most common structured overlays base on the idea of distributed hash tables (DHT). A DHT can be described as portions of a hash table distributed to peers where data objects are stored in those hash tables. Data objects and peers can be identified by identifiers (calculated by a hash function), that means in the (key, value) pair of a hash table entry, the key is used to retrieve the data object which is stored in the hash table.

## 1.2 Location-based Overlays

Recently, location-based services are becoming more and more popular. These services, as the name implies, allow users to get spatial information just by searching for certain locations and keywords such as restaurants or cinemas. In P2P environments, these services are provided by location-based overlays. Unlike overlays which use one-dimensional identifiers to find data, peers in location-based

overlays have coordinates which describe the location of a peer in multidimensional space. Each peer in the system stores spatial information to which he is locally and relatively close to in order to enable other peers to find spatial information by searching for nodes which are close to the search location.

## 1.3 Overview

The next Chapter discusses related work and their drawbacks, it highlights some existing work which influenced the design of GeoMesh. In Chapter 3, the design is presented, along with important definitions and formulas. In Chapter 4, the implementation of GeoMesh in the overlay network Simulator PeerfactSim.KOM [SGR+11] is presented. In Chapter 5, the overlay is evaluated using different metrics and setups.

# Chapter 2

# Related Work

In this Chapter, existing work and their drawbacks are examined. We will briefly talk about and explain tree-based and DHT-based approaches and continue on with Geodemlia and SpatialP2P which are overlays that mostly influenced the design choice of GeoMesh.

Recently, a lot of work has been made in order to be able to implement location-based services in P2P environments. Most of these approaches are either tree-based [THS07] [AB09] or are built on top of DHT overlays using space filling curves.

In tree-based overlays, the underlying multidimensional space is progressively subdivided into zones or areas and peers which are positioned in those areas are responsible for spatial information in it. Peers are therefore ordered hierarchically as the divided space. The root of the tree represents the peer which is responsible for the whole area, whereas the lower levels of the tree represents the peers which are responsible for the subdivided areas. Also, peers which are at the leafs of the tree provide information of their area, while peers at the upper levels of the tree, are used to route to those peers. This structure has some major drawbacks in terms of scalability and load-balancing, because each peer in the overlay always needs to contact peers at the upper level of the tree, when searching for spatial information. Also, those overlays can become unstable or can completely fail when peers, especially peers in the upper level of the tree, leave the system (due to failure).

Other approaches are built on top of DHT overlays using space filling curves. Because of the properties of the underlying DHT overlay the above mentioned problems do not occur, but performance problems do occur, because neighboring multidimensional information is not always stored in neighboring nodes in the one-dimensional structure of DHT overlays.

In P2P overlays such as SpatialP2P [KSS09] and Geodemlia [GSR+12], the above mentioned problems do not occur. In addition to that it is very efficient in terms of searching for location-based information, because both overlays provide properties such as locality and directionality. Locality implies that neighboring multidimensional information is stored in the neighboring nodes in the system while directionality implies that the structure of the system preserves orientation in multidimensional space [KSS09]. For example, directionality is preserved when an information which lies north of a peer $p$'s position (in multidimensional space), also lies north of $p$'s position in the system. This

actually means, that searching for location-based information in an overlay is like searching in multi-dimensional space.

Considering all the mentioned aspects, the main goal is to design an overlay which is stable under churn and efficient at once. Because of the fact that Geodemlia is an overlay which fulfills these properties, the focus was mainly on it. In fact, the iterative search and the idea of dividing the space into directions is adopted from Geodemlia, which is explained in detail in the upcoming Chapter.

In this Chapter, we have seen existing location-based overlays which are either not stable or not efficient. We have seen Geodemlia and SpatialP2P which are efficient overlays, especially Geodemlia is stable and efficient at once. In the next Chapter the design of GeoMesh is presented.

# Chapter 3

# Design of Mesh

This Chapter first gives an overview of the properties and functions provided by the GeoMesh overlay. Subsequently, we will define the requirements which peers need to fulfil in order to participate in the overlay network. Then, we will step by step construct the protocol, first by defining the positioning of peers and the respective distance function. Then, definitions and important formulas will be introduced in order to understand the routing table structure. After understanding the routing table structure, we can finally introduce functional elements such as search, join and leave. Lastly, to improve the routing efficiency, a *second routing table* will be introduced which is called finger table. The term finger table is referring to the term in Chord [SMLN$^+$03].

## 3.1 Overview

GeoMesh provides methods such as finding the set of k-closest nodes to a position or searching for nodes within a circular area in multidimensional space. Peers find their routing table contacts by using the find $k$-closest nodes function to be able to join the overlay. GeoMesh also provides important properties such as locality and directionality, by using an appropriate routing table structure. Again, locality implies that neighboring multidimensional information is stored in the neighboring nodes in the system while directionality implies that the structure of the system preserves orientation in multidimensional space [KSS09]. To improve the efficiency in terms of routing steps, peers also use a *second routing table* structure which is called finger table referring to the term in Chord [SMLN$^+$03] which provides contacts in each cardinal direction relative to $p$'s positon (in Section 3.9 the finger table is discussed in detail). In addition to that, GeoMesh is designed to work with spherical coordinates.

## 3.2 Requirements

Peers in GeoMesh use the underlay protocol TCP to contact each other. An identifier is required to uniquely identify a peer in the system. A peer, who wants to participate in the overlay network, has to determine an identifier first. In GeoMesh, a peer calculates his identifier $id \in [0, 2^{160})$ by hashing his IP-address and Port number, this is done by using SHA-1 [ErJ01]. In addition to that, each peer also must determine his position in space in terms of geographic coordinates.

## 3.3 Positioning and Distance

In GeoMesh, peers and data objects are bound to spherical points. A (spherical or geographical) point $P = (\phi, \lambda)$ is a coordinate tuple where $\phi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ is denoted as the latitude and $\lambda \in [-\pi, \pi]$ as the longitude. The distance between two points $P_1 = (\phi_1, \lambda_1)$ and $P_2 = (\phi_2, \lambda_2)$ on a sphere can be calculated by using the Haversine-Formula [Sin84] which is shown in (3.1).

$$d(P_1, P_2) = R \cdot arcsin\left(min\left(1, \sqrt{a}\right)\right) \tag{3.1}$$

where

$$a = sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + cos(\phi_1)\, cos(\phi_2)\, sin^2(\lambda_2 - \lambda_1) \tag{3.2}$$

and $R$ is the radius of the sphere.

## 3.4 Routing Table

In this Section we will deal with important formulas and definitions, to be able to understand and introduce the routing table structure of peers in this overlay. We will straightaway start with the definition of a bearing angle and will see the use of it later in the Section.

A bearing angle can be roughly described as the clockwise angle between a reference line and the line which connects the point of measurement and the point of interest(Figure 3.1). The term line is defined as the shortest route between two points on a sphere, the reference line is a line which connects the point of measurement and the reference point. For the sake of simplicity and clarity, each peer in the system uses the north line as the reference line and with this, a peer is able to determine in which direction another peer is positioned by using (3.3) in order to calculate the bearing angle [Bea],

which is illustrated in 3.1. So, given a point of measurement $P_1 = (\phi_1, \lambda_1)$ and a point of interest

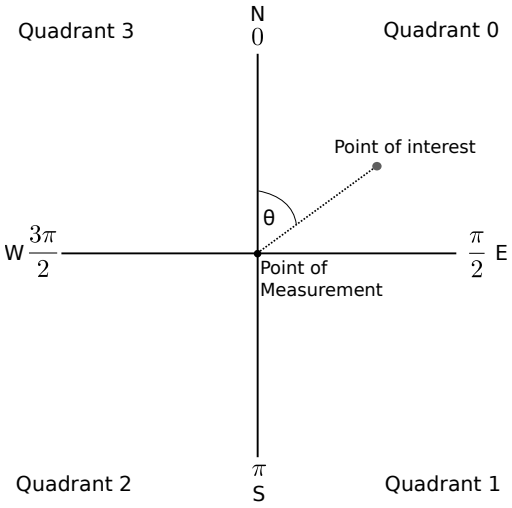



Figure 3.1: Bearing angle

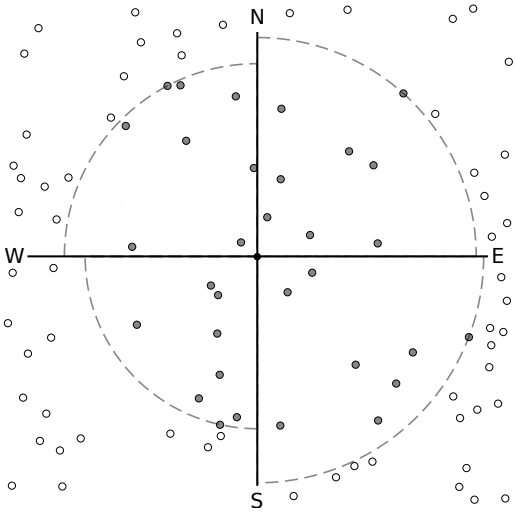



Figure 3.2: Routing table structure ($k = 8$)

$P_2 = (\phi_2, \lambda_2)$, the bearing angle $\theta \in [0, 2\pi]$ can be calculated by using formula (3.3).

$$\theta = (atan2(c,d) + \pi) \ mod \ 2\pi \tag{3.3}$$

where

$$b = sin(\lambda_2 - \lambda 1) cos(\phi_2) \tag{3.4}$$

and

$$c = cos(\phi_1) sin(\phi_2) - sin(\phi_1) cos(\phi_2) cos(\lambda_2 - \lambda_1) \tag{3.5}$$

In Figure 3.1, the term quadrant is used to abstract *bearing angles*. Given two peers, $p_1$ at a point of measurement and $p_2$ at a point of interest, $p_1$ can determine the quadrant $i \in \{0,1,2,3\}$ in which $p_2$ is positioned, by calculating (3.3) and (3.6)

$$i = \left\lfloor \frac{\theta}{2\pi} \right\rfloor \tag{3.6}$$

With the help of these definitions, we can now introduce the routing table structure and design of GeoMesh:

- Each peer p maintains a set $S$ of peers and contact information which should depict the routing table
  - A peer and his contact information in $S$ is stored in the following way:

    - IP address and port
    - Identifier $id \in [0, 2^{160})$
    - Position

- Set $S = S_0 \cup S_1 \cup S_2 \cup S_3$ can be subdivided
- Set $S_i$ stores at most k peers, it only consists of peers, which are located in quadrant $i$ and peers which are stored in $S_i$ are nodes closest to p's position, that p knows about. ($i \in \{0,1,2,3\}$)

The idea of dividing the space into four quadrants and using the peers in each quadrant is adopted from [GSR+12]. To get an overview, the routing table of a peer is shown in 3.2. So, the routing table structure ensures properties such as directionality and locality, this means that peers in GeoMesh can actually *use* the space for routing for position-based information, which is depicted in the upcoming Sections.

## 3.5 Find k-Closest Nodes

In previous Sections, we mainly focused on the structure of GeoMesh. We dealt with important formulas along with important definitions, in order to be able to create a system in which peers can actually *use* the space for routing, yet we have not specified a protocol how this can be done. So, in this Chapter we will see how peers actually search for nodes in the overlay. In this Section and in the upcoming Sections the term *node* instead of *peer* is used.

In Figure 3.3, we can see the pseudo-code which shows how a peer finds the set of k-closest nodes to a position $P = (\phi, \lambda)$. Before we can discuss the pseudo-code, we have to understand which information the sets $S_1$, $S_2$, $S_3$ and $S_4$ contain. The sets $S_2$ and $S_3$ only consist of node identifiers, while the remaining sets consist of objects which store contact information, position and the identifier of the respective node. For simplicity, we treat these objects as nodes(e.g. for a node $n \in S_1$, $n.position$ would return the position of $n$).

A node $n$ who wants to find the k-closest nodes to a position $P$, first sets himself as the contact node $n'$ (line 6) and sends $n'$ (himself) a message (line 8) which contains the position $P$, the number of nodes $k$ which should be returned and $S_3$ which consists of node identifiers that have been found during the search process, in order to not receive those nodes again. In particular, a node $n'$ who receives a message from $n$, iterates through a set of contacts $C$ which are contacts from his routing table and only returns the k-closest nodes from the set $C$ to position $P$, which are not in the *identifier* set that he received from $n$. This is shown in the pseudo-code in Figure 3.4. After that, node $n$, who

```
1:  n.find_closest_nodes(P,k)
2:    S1 = ∅
3:    S2 = ∅
4:    S3 = {n.identifier}
5:    S4 = ∅
6:    n' = n
7:    do
8:      S1 = n'.get_closest_nodes(P,k,S3)
9:      S2 = S2 ∪ {n'.identifier}
10:     for each node ∈ S1 do
11:       S4 = S4 ∪ {node}
12:       S3 = S3 ∪ {node.identifier}
13:     for each node ∈ S4 do
14:       if (d(node.position,P) < d(n'.position,P)) then
15:         n' = node
16:   while (n'.identifier ∉ S2)
17: return S4
```

Figure 3.3: Node $n$ initiates an iterative search for the k-closest nodes to position $P$.

```
1:  n.get_closest_nodes(P,k,S)
2:    L1 = [ ]
3:    C = n.routing_table.all_contacts
4:    for each node ∈ C do
5:      if (node ∉ S) then
6:        L1.add(node)
7:    sort(L1,P)
8:  return L1.sub_list(0,k).to_set()
```

Figure 3.4: Node $n$ returns the k-closest nodes to position $P$ that are not in the *identifier* set $S$.

receives the reply, temporarily stores these nodes in $S_1$ (line 8). Then he puts $n'$ to the set $S_2$ (line 9), which is a set that consists of nodes that already have been contacted. After that, those nodes in $S_1$ are stored in $S_4$ and the identifiers are stored in $S_3$ (line 10-12). Then, node $n$ determines the closest node $n'$ out of all nodes that he have encountered at that time which are nodes in $S_4$ (with the help of the haversine formula (3.1)) (line 13-15). Then, if the identifier of node $n'$ is in the set $S_2$, the search is over and the set $S_4$ contains the k-closest nodes to position $P$ including nodes which have been found during the search process. Otherwise, the search goes on and node $n'$, is the next node to be contacted. The algorithm in Figure 3.4 is for the most part self-explanatory, but lines 7-8 might be unclear. In line 7, $n'$ sorts the nodes in list $L$ according to their distances to $P$ in ascending order. In line 8, the function $sub\_list(0,k)$, returns a list from $L$ from index 0 (inclusive) to $k$ (exclusive). The function $to\_set()$ converts the list into a set.

To sum it up, in each step $n$ encounters new nodes, he then determines the closest node to position

$P$ out of all nodes that he have encountered at that time. If he already contacted the closest node, then the search is finished and the k-closest nodes (including all nodes that have been found during the search process) are returned.
Important note: The algorithm in Figure 3.3 describes searching in a graph, in reality, the searching node would first ping the closest contact node and wait for a reply, if no reply is sent back after some time, he would choose the next closest contact node and repeat this prodecure until he gets a reply. After that, he would ask this contact node for his $k$-closest nodes (to the target point). Also, nodes who are not responding can be removed from the search result.

## 3.6 Area Search

```
1:  n.area_search(P,k,r)
2:    S_1 = ∅
3:    S_2 = n.find_closest_nodes(P,k)
4:    for each node ∈ S_2 do
5:        if (point_within_area(node.position,P,r)) then
6:            S_2 = S_2 ∪ {node}
7:    if (S_2.is_empty()) then
8:        return S_2
9:    else
10:       S_3 = ∅
11:       S_4 = ∅
12:       while (S_2.is_not_empty()) do
13:           for each node ∈ S_2 do
14:               S_3 = S_3 ∪ {node.identifier}
15:           n' ∈ S_2
16:           S_2 = S_2 \ {n'}
17:           S_4 = S_4 ∪ n'.get_nodes_within_area(P,r,S_3)
18:           S_2 = S_2 ∪ S_4
19:   return S_2
```

Figure 3.5: Node $n$ initiates an area search with radius $r$, with parameter $k$ which should first find the $k$-closest nodes to $P$.

```
1: n.get_nodes_within_area(P, r, S)
2:   S_1 = ∅
3:   for each node ∈ n.routing_table.all_contacts do
4:     if (point_within_area(node.position, P, r)) then
5:       if (node.identifier ∉ S) then
6:         S_1 = S_1 ∪ {node}
7: return S_1
```

Figure 3.6: Node $n$ returns nodes if his nodes lie within the search area which is defined by the radius $r$ around point $P$.

```
1: point_within_area(P_1, P, r)
2:   if (r > d(P, P_1)) then
3:     return true
4:   else
5:     return false
```

Figure 3.7: This function returns true if $P_1$ is a point in an area defined by point $P$ and radius $r$

The *area_search* function internally utilizes the $find\_closest\_nodesfunction$. If no point lies in an area then the $k$-closest nodes are returned. Otherwise each node in the search area who have been found is queried and asked for additional nodes in the search area. If there is no node left to contact in the set $S_2$, the search is finished.

## 3.7 Join and Leave

In Section 3.5 we saw the most crucial part of the system, which is the find $k$-closest nodes function, this function enables joining peers to find the $k$-closest nodes. Due to the fact that each peer stores maximum $s$ nodes in his routing table, it is recommended that each joining peer searches for the $s$-closest nodes to join the overlay, as it was successfully evaluated with it. With this, we can now discuss the join and leave process of nodes. A node $n$ who wants to join the overlay first needs to have at least one node in his routing table to initiate a search for the closest nodes to his position. This can be done by using a bootstrapping node. More precisely, a bootstrapping node is a node who is already participating in the overlay. So, node $n$ adds the bootstrapping node into his routing table and would call the function $n.find\_closest\_nodes(n.position, k)$ which returns a set $S$ of k-closest nodes and nodes who were additionally found during the search process, this is depicted in the previous Section

and Figure 3.3. When the operation is finished, node $n$ then stores the contacts in $S$ to his routing table. Important, each node can store at most k nodes from each quadrant to his routing table, which was depicted in Section 3.4. So, node $n$ stores the k-closest nodes from each quadrant, if possible, to his routing table, which was found during the search process. After that node n sends a notifier message in order to inform them. Nodes which receive this message, can add $n$ to their routing table.

A node who wants to leave, on the other hand, doesn't need to perform some extra procedures, he can directly leave the overlay. A node who becomes aware (during a search or a ping operation) that a node in his routing table is not answering, would then remove this particular node.

## 3.8 Routing Table Maintenance

Each peer, periodically, sends a ping message to the nodes in his routing table. If nodes reply, we store them in a set which we is called reachable contacts, if nodes are not responding we remove them from the set. Each node uses the contacts in the reachable contacts set, to search or to reply to searches.

## 3.9 Finger Table

To improve the overlay protocol in terms of routing steps, peers can also use a second routing table structure which we will call finger table referring to the term finger table in Chord [SMLN$^+$03]. The idea of using finger tables is adopted from [KSS09] which is referred as indexed peers.

The finger table of a peer is a two-dimensional $4 \times m$ array. In particular, for a peer in position $P_1$, $finger\_table[i][j]$ ($i \in \{0,1,2,3\}$) returns the closest peer to position $P_2$ which is $2^j$ away from the position $P_1$ (here the earth is being modelled and we use meter as default unit) and has a bearing angle of $i \cdot 90^\circ$ measured from $P_1$. The bearing angles $0^\circ$, $90^\circ$, $180^\circ$ and $270^\circ$ (in degrees) describe the cardinal directions, which was illustrated and discussed in Section 3.4. The procedure in Figure 3.8 shows how the finger table entries are updated. This procedure is periodically executed to ensure the finger table is correctly maintained and permanently updated. The input argument can be described as the bearing angle $i \cdot 90^\circ$, for instance, if $i = 0$, then the finger table entry in north direction from $n$'s position is updated, as $0 \cdot 90^\circ = 0^\circ$, bearing angles and directions are discussed in Section 3.4 and illustrated in Figure 3.1, for simplicity we denote the variable $i$ as direction. The pseudo-code is for the most part self-explanatory, so we concentrate on the important parts. In line 5 $n$ calls the function in Figure 3.9 which returns him a position $P$ which is $2^{next}$ away in direction $i$ from his own position. Node $n$, then, searches for the closest node to position $P$ (line 6). The function $find\_closest\_nodes$ returns a set which has the closest node including nodes which were found during the search process

```
1:  n.fix_fingers(i)
2:    next = next + 1
3:    if (next > m) then
4:        next = 0
5:    P = get_position(n.position, direction, 2^next)
6:    S = n.find_closest_nodes(P, 1)
7:    n' ∈ S
8:    for each node ∈ S do
9:        if (d(node.position, P) < d(n'.position, P)) then
10:           n' = node
11:   finger_table[direction][next] = n'
```

Figure 3.8: Node $n$, periodically, executes the $fix\_fingers$ procedure to update the respective finger table entry, given a direction $i \in \{0,1,2,3\}$ which describes the bearing angle of $i \cdot 90^\circ$ from $n$'s position

and therefore determines the closest node out of $S$ (line 8-10). He, then, updates his respective finger table entry (line 11) in direction $i$ and $next$.

```
1:  get_position(P, i, d)
2:    φ1 = P[0]
3:    λ1 = P[1]
4:    θ = i · π/2
5:    δ = d/R
6:    φ2 = arcsin(sin(φ1) cos(δ) + cos(φ1) sin(δ) cos(θ))
7:    λ2 = λ1 + atan2(sin(θ) sin(δ) cos(φ1), cos(δ) − sin(φ1) sin(φ2))
8:  return {φ2, λ2}
```

Figure 3.9: Given a starting point $P$, a direction $i \in \{0,1,2,3\}$ and a distance $d$, this function returns the end point, which is $d$ away and has a bearing angle of $i \cdot 90^\circ$ to position $P$ on a sphere with Radius $R$ (see [Bea]).

An important note, as we will model the earth, we use the mean radius of the earth (in meters). In the pseudo-code in Figure 3.9 the radius $R$ is kept general.

## 3.10 Conclusion

The main goal was to establish an overlay which should preserve properties such as locality and directionality [KSS09], because only these properties enables peers to search for position-based information efficiently by *using* the actual space in which they are positioned. Also, the use of spherical coordinates enables us to actually implement this overlay protocol in real applications. The overlay currently supports iterative search, but can be modified to support recursive search. The iterative

search has the advantage, that the node who started the search has a somehow global view of the network, because as the search progresses, he more and more encounters new nodes and can pick the *right* node to contact. With the help of the finger table, peers are able to route in logarithmic amount of steps with respect to the number of peers in the system. In the next Chapter, we will see the implementation of this overlay in the overlay network simulator PeerfactSim.KOM [SGR+11].

# Chapter 4

# Implementation

In this Chapter, we will mainly focus on the implementation of the GeoMesh overlay which was developed throughout this thesis in the overlay network simulator PeerfactSim.KOM [SGR+11]. This Chapter will have a similiar approach as the previous Chapter, because we first focus on the base components of the system and later on we will see the functional components.

## 4.1 Identifier, Contact Information and Position

**MeshContact**
In the previous Chapter, we have seen that each peer in the overlay has to determine an identifier by using the SHA-1 hash function and his position in the (multidimensional) space to participate in the overlay network. In the implementation, this kind of information is stored in an object of the class *MeshContact*. Each time a new node is created, along with that, an object of the class *MeshContact* is created. This object stores contact information such as IP address, the identifier of that node in the overlay and his position in the "space". These objects also represent routing table entries in the system. An object of *MeshContact* is created by using the following constructor:

- *MeshContact(TransInfo info, double[] coord)*

*TransInfo* is an interface and every class which implements this interface has to have the following methods:

- *getPort()*
- *getNetID()*

As we use IPv4, *getNetID()* will return an IP address. The second argument is an array of type double and denoted as *coord*, *coord[0]* holds the longitude and *coord[1]* holds the latitude. For simplicity, we will also say *x* which represents the longitude and *y* which represents the latitude.

The class *MeshContact* contains the following fields:

- *nodeId*
  The node identifier which is a BigInteger object that can handle large numbers, as we use an identifier space of 160 bits.

- *nodeInfo*
  A TransInfo object which holds the IP address and port number as mentioned above.

- *MESH_BIT_LENGTH*
  A static variable which defines the overlay identifier space which is set to 160 by default.

- *coord*
  The position information of a node in multidimensional space as mentioned above.

The class contains the following methods:

- *getIdentifier()*
  Returns the BigInteger object nodeId.

- *getTransInfo()*
  Returns the TransInfo object nodeInfo.

- *getCoord()*
  Returns the double array coord.

- *getSHA1Hash(String stringToHash)*
  A static utility method which generates a SHA-1 hash value of length *MESH_BIT_LENGTH* by hashing stringToHash. With this the node identifier *nodeId* is calculated.

- *equals(Object obj)*
  A method which determines if *this* object is equal to *obj* by checking the identifiers for equality.

- *getTransmissionSize()*
  Returns the size of *this* object which is the sum of the sizes of nodeId, nodeInfo and coord in Bytes.

- *hashCode()*
  It calculates a hash code for an object of *MeshContact* to be able to use those objects in data structures such as *HashMap* or *HashSet*.

**MeshNodeGeographicalPositioning**

Since each node in the overlay uses coordinates for searching, the class *MeshNodeGeographicalPositioning* is used by peers to calculate their position, to calculate the distance between two points, to be able determine the quadrant of a peer. Also the function *pointWithInArea* is introduced in the previous Chapter, which the *area_search* function utilizes. The function *getCoordByAddingDistanceTo* is introduced with the name *get_position* in the previous Chapter, which the *fix_fingers* function utilizes. The following methods are provided by *MeshNodeGeographicalPositioning* class:

- *getPosition()*
  Returns a *double* array with the longitude and latitude. The longitude is by default a random value from $-\pi$ to $\pi$ and the latitude is by default a random value from $-\pi/2$ to $\pi/2$. Each node uses this function to determine his position.

- *getDistance(double[] coord1, double[] coord2)*
  This method returns the distance (in double) between *coord1* and *coord2* by using the haversine formula in equation (3.1), $R$ is set to 6371 which is the mean radius of the earth in kilometer.

- *getQuadrant(double[] coord1, double[] coord2)*
  This method returns the number of the quadrant in which the point *coord2* lies measured from *coord1*.

- *pointWithinArea(double[] loc, double[] coord, double r)*
  This method returns true if the coordinates of *loc* lie within an circular area defined by the radius *r* around a point *coord*.

- *getCoordByAddingDistanceTo(double[] coord, int direction, double distance, String unit)*
  This method returns a double array which holds the coordinates of a point which is *distance unit* away from *coord* in *direction* $0-3$ which describes the four cardinal directions, as we discussed in the previous Chapter. As the earth is being modelled, the String *unit* should be either "cm", "m" or "km".

- *getCoordInDegrees(double[] coord)*
  This method expects coordinates in radians. It converts the coordinates coord to degrees and returns them.

## 4.2 Routing Table

**MeshRoutingTable**

Each node in the overlay creates an object of the *MeshRoutingTable* class. In the previous Chapter, we have seen that the routing table of each node $n$ consists of four parts. Each part $i \in \{0,1,2,3\}$ has nodes from quadrant $i$ relative to $n$'s position. In order to achieve that, each object of the *MeshRoutingTable* class contains a list of lists of *MeshContact* objects which is called *neighborsListByQuadrant*. For instance, the contact list from quadrant $i$ can be accessed with *neighborsListByQuadrant.get(i)*. We will first have a look on the field variables and then discuss the methods provided by the class *MeshRoutingTable*.

The class *MeshRoutingTable* contains the following fields:

- *nodeCoord*
  This field variable is an array of type double. It holds the coordinates of the node to which the MeshRoutingTable object belongs.

- *maxNeighborsFromEachQuadrant*
  With this field variable a node knows the maximum number of contacts he can store from each of the four quadrants in his routing table. (Note: This is a system wide parameter)

- *neighborsListByQuadrant*
  As mentioned above.

- *neighbors*
  This field variable is a set which contains all neighbors from *neighborsListByQuadrant*.

- *reachableContacts*
  This field variable is a set which contains all nodes which were online during the last PeriodicPingOperation (which is described below).

The class *MeshRoutingTable* has the following methods:

- *init()*
  Initializes the lists *neighborsListByQuadrant* and *neighbors*.

- *contains(MeshContact contact)*
  Returns true if the routing table contains *contact*.

- *addContact(MeshContact contact)*

Adds *contact* to the list neighbors and *neighborsListByQuadrant.get(i)* where *i =getQuadrant(nodeCoord, contact.getPosition())* from the class *MeshNodeGeographicalPositioning*.

- *removeContact(MeshContact contact)*
  Removes *contact* from the routing table, if it is present.

- *isEmpty()*
  Returns true if the routing table is empty, otherwise false.

- *clear()*
  Removes all contacts from the routing table.

- *getContactsAsSet()*
  Returns all contacts from the routing table in a set.

- *getReachableContacts()*
  Returns all reachable contacts which were online during the last periodicPingOperation (which is described below).

- *getContactsFromQuadrant(int i)*
  Returns contacts from *neighborsByQuadrant.get(i)*.

- *getContactWithMaxDistInQuad(int i)*
  Returns the furthest away contact(in the space) from *neighborsByQuadrant.get(i)*.

- *buildRoutingTableFrom(ClosestNodesResult result)*
  This method is used to build the routing table from *result*, which will be described in the upcoming Sections.

- *size()*
  Returns the number of contacts in the routing table.

## 4.3 Messages and Operations

**Messages**

In a P2P overlay network, nodes are only able to communicate with each other by sending messages through the underlay. A message can trigger different actions at the receiving node. For instance, a

simple ping message only needs to be replied, while a message for the k-closest nodes would trigger more actions. More precisely, the receiving node would extract all the information from the arrived message, would query his routing table for the k-closest nodes and could only then reply. In the simulation environment a *MeshNode* class implements the *TransMessageListener* interface, in order to receive messages. Messages which arrive at a node have different types which we will see in the following.

The *EduBaseMessage* class provides the most basic properties such as the *TransInfo* objects which identifies the sender and receiver of the message in the underlay. Each type of message class in GeoMesh extends the *EduBaseMessage* class.
The following types of messages are implemented:

- *GetNeighborsMessage*
  This message is sent by nodes who are searching for the *k*-closest nodes(to a target point). The message contains the target point, a list of node identifiers which were already found.

- *GetNeighborsResponse*
  This message is the response of the *GetNeighborsMessage*, the node replies with the k-closest nodes he knows about and which were also not in the list of node identifiers which he received.

- *NotifyMessage*
  This message is sent by nodes who have successfully found their neighbors in order to notify them. The message contains the *MeshContact* object of the sender node. A node who receives this message can add the sender node to his routing table. If his routing table *neighborsByQuadrant.get(i)* (where *i* is the quadrant in which the sender node is positioned) is full, he will pick the furthest away node out of this list and compare the distance with the sender node. If the sender node has a shorter distance, he will remove the furthest away node out of *neighborsByQuadrant(i)* and add the sender node.

- *NotifyResponse*
  This message informs the sender of the *NotifyMessage* that the message arrived.

- *PingMessage*
  This message is sent by nodes who want to check the online status of nodes in their routing table.

- *PingResponse*
  This message is the response of the *PingMessage*. A node who receives this message knows that the sender node is alive.

- *AreaSearchMessage*

This message is sent by nodes who are searching for nodes(or data objects) in a certain area. A search area is defined by the radius around a point. So the message consists of a target point (while the radius is a system wide parameter) and list of node identifiers which were already found.

- *AreaSearchResponse*
  This message is the response of the *AreaSearchMessage*, the sender node replies with nodes that lie within an area and were not in the list of node identifiers which he received.

**Operations**

The term operation can be described as the action performed by a node, e.g. a node who wants to search for the $k$-closest nodes invokes a FindClosestNodesOperation in order to find them. In the following we will focus on important operations which we discussed in Chapter 3.

An operation is an instance of a class that extends the abstract class *AbstractOperation*. An operation can be invoked by methods such as *scheduleImmediately()* which should execute an operation immediately or *scheduleWithDelay*($t * Simulator.SECOND\_UNIT$) which should be executed in $t$ seconds. A class which extends the class *AbstractOperation*, must also implement:

- *execute()*
  This method is being executed when an operation has been invoked.

- *getResult()*
  The result of an operation.

Also, a node, who started an operation, is able to send messages during an operation to other nodes. If the operation class implements the *TransMessageCallback* then he is able to receive messages in the operation class.
A class which implements the *TransMessageCallback*, must also implement:

- *messageTimeoutOccured(int commId)*
  If a timeout has occured, this method is called.

- *receive(Message msg, TransInfo senderInfo, int commId)*
  The received message is the response of the sent message.

Following operations are currently implemented:

- *FindClosestNodesOperation*
  Is the implementation of the *find_closest_nodes(P, k)* function from Chapter 3. It utilizes the class *GetNeighborsMessage* and *GetNeighborsResponse*.

- *AreaSearchOperation*
  Is the implementation of the *area_search(P, k, r)* function from Chapter 3. It utilizes the class *AreaSearchMessage* and *AreaSearchResponse*.

- *FixFingersOperation*
  Is the implementation of the *fix_fingers(i)* procedure from Chapter 3, this operation is periodically executed and utilizes the FindClosestNodesOperation class.

- *NotifyOperation*
  A node which notifies his neighbors executes this operation. For more details see above.

- *PeriodicPingOperation*
  A node periodically pings contacts in his routing table and stores the nodes which responded in the *reachableContacts* set in the *MeshRoutingTable* class. Nodes which not responded are removed from the *reachableContacts* set. In the current version, this operation pretends to ping the nodes, but it internally checks from a global list if nodes are online and simulates a ping operation.

- *PeriodicSearchOperation*
  A node periodically searches for the *k*-closest nodes to a random point.

- *PeriodicAreaSearchOperation*
  A node periodically starts an AreaSearchOperation to a random point.

## 4.4 Node

**MeshNode**

In the simulator, the objects of the class *MeshNode* depict nodes in the overlay. Each of these objects uses the above mentioned components and functional elements.
The *MeshNode* class has the following field variables:

- *nodeContact*
  Each instance of *MeshNode* creates a *nodeContact* object of type MeshContact where contact information, position and the identifier is stored.

- *MESH_PORT*
  A static variable and therefore a system wide paramater. Each node is reachable under *MESH_PORT*.

- *fingerTable*
  A two-dimensional array. For more details please refer to Chapter 3.

- *fixFingersOperationEast*
  A periodically executed operation which updates finger table entries in east direction. For more details please refer to Chapter 3.

- *fixFingersOperationWest*
  A periodically executed operation which updates finger table entries in west direction. For more details please refer to Chapter 3.

- *fixFingersOperationNorth*
  A periodically executed operation which updates finger table entries in north direction. For more details please refer to Chapter 3.

- *fixFingersOperationSouth*
  A periodically executed operation which updates finger table entries in south direction. For more details please refer to Chapter 3.

- *isActive*
  Set to true or false. In reality, this variable is not necessary. But a node in the simulator should somehow stop executing operations or stop replying to other nodes, if he is not present anymore.

- routingTable An instance of *MeshRoutingTable*, described in the previous Section.

- fingerUnit
  A String which is either "cm", "m" or "km". As the earth is being modelled, we can specify a unit. For instance, *fingerTable[i][j]* is $2^j$ units away in direction i, as discussed in Chapter 2.

- fingerRange
  The range of each finger table. By default fingerRange is set to 25.

The *MeshNode* class has the following methods:

- *join()*
  This method is used by nodes to join the overlay. A node searches for the closest nodes to his position by using the operation FindClosestNodesOperation. If the search was successful, the method *joinOperationSuccess(ClosestNodesResult result)* is called.

- *joinOperationSuccess(ClosestNodesResult result)*

This method is called, when a node successfully found his neighbors. All components and operations are instantiated. Periodic operations are scheduled immediately and neighbors are notified.

- *messageArrived(TransMsgEvent receivingEvent)*
  The *MeshNode* class implements *TransMessageListener*, so it can receive messages. Those messages are handled in this method. This method handles messages of type *GetNeighborsMessage*, *AreaSearchMessage*, *NotifyMessage* and *PingMessage* (see above).

- *closestFinger(double[] target, int i)*
  This method is called to calculate the closest "finger" contact from direction *i*.

- *closestFinger(double[] target)*
  This method is called to calculate the closest "finger" from all directions. It internally utilizes the *closestFinger(double[] target, int i)* method.

- *getClosestContactsTo(double[] target, int k)*
  This method determines the k-closest contacts from the reachableContacts set and finger table.

- *connectivityChanged(ConnectivityEvent ce)*
  This method is invoked whenever a node's online status has changed.

# Chapter 5

# Evaluation

## 5.1 Routing Table Size and Search Results

Before we begin with the actual evaluation, we first want to test if searching initiated from different positions on a sphere, always returns the k-closest nodes to target point $P$, which is calculated before the experiment starts and globally known by each peer in the system. Due to the fact that the routing table size also has a big impact on the search result, we will vary the routing table sizes in each test. Also, each peer needs to have the right contact nodes in his routing table, in order to sufficiently reply to queries (see Chapter 3), so each peer who joins the overlay initiates a search for the $k'$-closest nodes to his position where $k'$ is the maximum routing table size. We will run each test five times with different seeds and average the results. No finger tables or routing table maintenance operations were used during each test.

First, a random point $P$ on the sphere is calculated. Then, we let 5000 peers join the overlay. Then, each peer initiates a search to point $P$ (which is globally known by every peer in the system). If a peer has found the k-closest nodes, we denote this as successful search, and the success ratio is calculated by the number of successful searches of every peer divided by the number of peers(which searched for the k-closest nodes to $P$) in the system. The following table should again highlight the test cases.

| Parameters | |
|---|---|
| Peer Distribution | Uniformly Distributed on a Sphere |
| Number of Peers | 5000 |
| Maximum Routing Table Size $s$ | $16, 32, 40, 48$ |
| Number of Closest Nodes $k$ to Find | $1, 2, 4, 8, 16$ |

A reminder: A routing table can store up to $s$ nodes and can also be subdivided into four parts each part stores up to $s/4$ contacts from each of the four quadrants (Chapter 3).

Figure (5.1) shows that the closest node to the same target position is found by almost every peer in

the system. It also shows that nodes taking an inconvenient search path (due to their position) might not find the closest node. An inconvenient search path can occur, when a node on the search path does not return closer nodes to the target position although there might be closer nodes in the system, this happens in cases when the maximum routing table size is relatively speaking small. More precisely, a node $n$ who joins the overlay notifies his contact nodes in his routing table in order to be known by them. But $n$ ignores those nodes that can and must also store $n$. This problem becomes more apparent for $k > 2$, while the success ratio is more or less the same for $k = 2$, it decreases when searching for four or more closest nodes. This can be simply solved by slightly increasing the maximum routing table size, as illustrated in figure (5.2), (5.3) and (5.4).

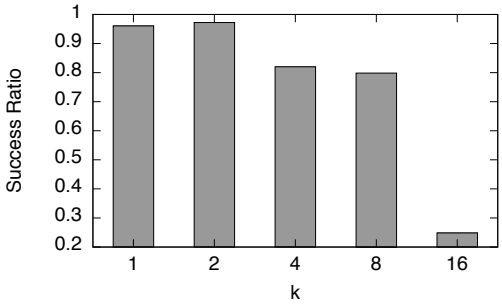



Figure 5.1: Success ratio when $s = 16$

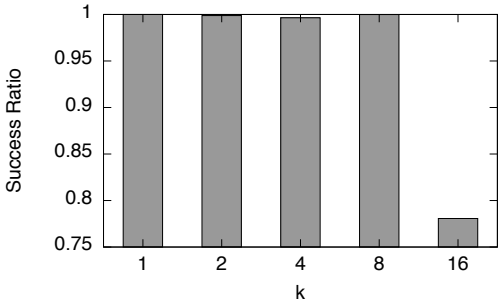



Figure 5.2: Success ratio when $s = 24$

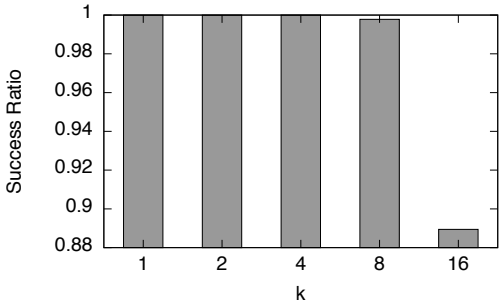



Figure 5.3: Success ratio when $s = 32$

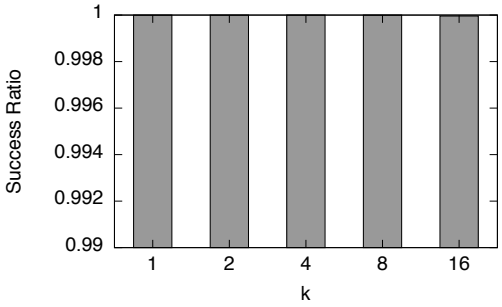



Figure 5.4: Success ratio when $s = 40$

In this Section, we saw that the parameter maximum routing table size $s$ is an important factor. We saw that for $s = 40$ and $k = 1,2,4,8$ each node is always able to find the $k$-closest nodes to a target point. Although $s = 48$ showed a slightly better result, we will evaluate the overlay with $s = 40$.

## 5.2 Metrics

### 5.2.1 Scalability

- **Average Hop Count of Find $k$-Closest Nodes Operations:** The average hop count per find $k$-closest nodes operation is the overall number of messages sent by nodes to find the k-closest nodes to a random point divided by the overall number of find $k$-closest nodes operations started by each peer in the overlay. This metric is used to test the performance of the find $k$-closest nodes operation by varying the overlay size. It also indicates if an overlay is scalable.

- **Average Response Time of Find $k$-Closest Nodes Operations:** The response time per find $k$-closest nodes operation is the time the operation needs to find the k-closest nodes to a random point. The average response time per find $k$-closest nodes operation is the sum of all response times divided by the the overall number of find $k$-closest nodes operations started by each peer in the overlay. This metric is used to test if the overlay is responsive by varying the overlay size.

- **Overlay Traffic Per Peer:** The overlay traffic in $bytes/sec$ per peer, this metric is used to check the average number of bytes a peer receives every second during a simulation. The overlay size is increased in each experiment.

### 5.2.2 Stability and Efficiency

- **Number of Present Peers Per Minute:** This metric describes the number of present peers in the overlay per minute. This metric is useful in scenarios where churn is active.

- **Success Ratio of Find k-Closest Nodes Operations Per Minute:** If a peer has found the $k$-closest nodes, we denote this as successful search and the success ratio is calculated by the number of successful searches of every peer per minute divided by the number of peers who searched for the k-closest nodes per minute. This metric is useful in scenarios where churn is active, to test the find k-closest nodes operation in changing conditions, due to peers joining and leaving the system.

- **Success Ratio of Finding at Least One Closest Node Per Minute:** If a peer has found at least one node out of the k-closest nodes, we denote this as a successful operation, the success ratio is calculated by the number of successful operation per minute divided by the number of all find $k$-closest nodes operations started by each peer in the overlay per minute. This metric is useful in scenarios where churn is active, to test if the find k-closest nodes operation can find at least

one node sufficiently.

- **Hop Count of Successful Find $k$-Closest Nodes Operations Per Minute:** The number of messages sent by nodes which successfully find the k-closest nodes per minute divided by the number of all find $k$-closest nodes operation started by each peer in the overlay per minute. This metric is useful in scenarios where churn is active, to test if joining and leaving peers influences the performance of the system.

- **Response Time of Successful Find $k$-Closest Nodes Operations Per Minute:** The response time of each successful find $k$-closest nodes operation per minute divided by all find $k$-closest nodes operations. This metric is useful in scenarios where churn is active, to test the responsiveness of the overlay in changing conditions, due to peers joining and leaving the system.

### 5.2.3 Area Search

- **Success Ratio of Area Search Operations Per Minute:** If a peer has contacted every node in an area, we denote this as successful area search operation and the success ratio of successful area search operations per minute is calculated by the number of successful area search operations per minute divided by the number of started area search operations per minute. This metric is used in order to test if each area search operation can return all location-based information in an area with the assumption that each node would return his location-based information.

## 5.3 Setup

In this Section, we will shortly present the parameters and scenarios and the metrics used for each test. The following global parameters are used for each of the upcoming tests:

| Global Parameters | |
|---|---|
| Peer Distribution | Uniformly Distributed on the Earth |
| Latency Model | GNP Latency |
| Maximum Routing Table Size $s$ | 40 |
| Ping Operation Interval | 60 Seconds |
| Finger Range | 25 |
| Finger Unit | Meter |
| Update *Next* Finger Table Entry Interval | 10 Seconds |
| Churn (If Active) | KAD Churn |

Note: A node only sends ping messages to nodes in his routing table and not to nodes in his finger table.

### 5.3.1 Scalability

The number of peers are increased in each test (50, 100, 250, 500, 1000, 2000, 5000). We use the metrics from Subsection 5.2.1. In the first half hour we let peers join the overlay. Then, we wait 10 minutes, so each peer is able to update their fingers. Subsequently, from the 40th minute to 100th minute, each peer starts a find k-closest nodes operation every minute (to a random point on the earth) which we will measure as described by the first two metrics. The last metric in Subsection 5.2.1 is used to measure the overlay traffic per peer during each simulation. Also, no churn occurs during each simulation. The tests are repeated with different values of $k$ ($k = 1, 4, 8$) to compare the results.

### 5.3.2 Stability and Efficiency

In the first half hour we let 2000 peers join the overlay. Then, we wait 10 minutes, so each peer is able to update their fingers. Subsequently, from the 40th minute to 100th minute, each peer starts a find k-closest nodes operation (to a random point on the earth) every minute which we will measure as described by the metrics in Subsection 5.2.2. Also, from the 40th minute to 100th minute, churn occurs, to test the efficiency and stability under changing conditions. The tests are repeated with different values of $k$ ($k = 1, 4, 8$) to compare the results.

### 5.3.3 Area Search

In the first half hour we let 2000 peers join the overlay. Then, we wait 10 minutes, so each peer is able to update their finger table. Subsequently, from the 40th minute to 100th minute, each peer starts an area search operation every minute (to a random point on the earth) which we will measure as described by the metric in Subsection 5.2.3. The search radius $r$ is set to 500 $km$ and the parameter $k$ is set to 4, as the area search operation internally utilizes the find $k$-closest nodes operation. In the first test churn will not occur, in the second test churn will occur from the 40th minute to 100th minute.
Important Note: The search radius $r$ is intentionally set to 500 $km$, in order to ensure that nodes are positioned in search areas. If no node is positioned in an area, the area search operation returns the $k$-closest nodes to the search position.

## 5.4 Results

The results are presented in the upcoming pages.

### 5.4.1 Scalability

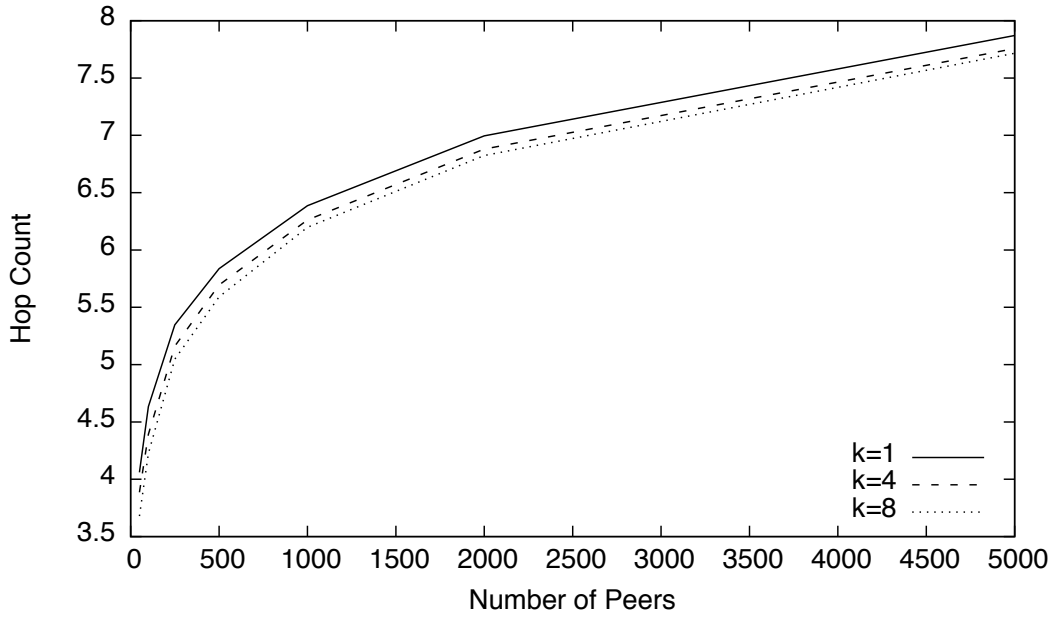



Figure 5.5: Average Hop Count of Find *k*-Closest Nodes Operations

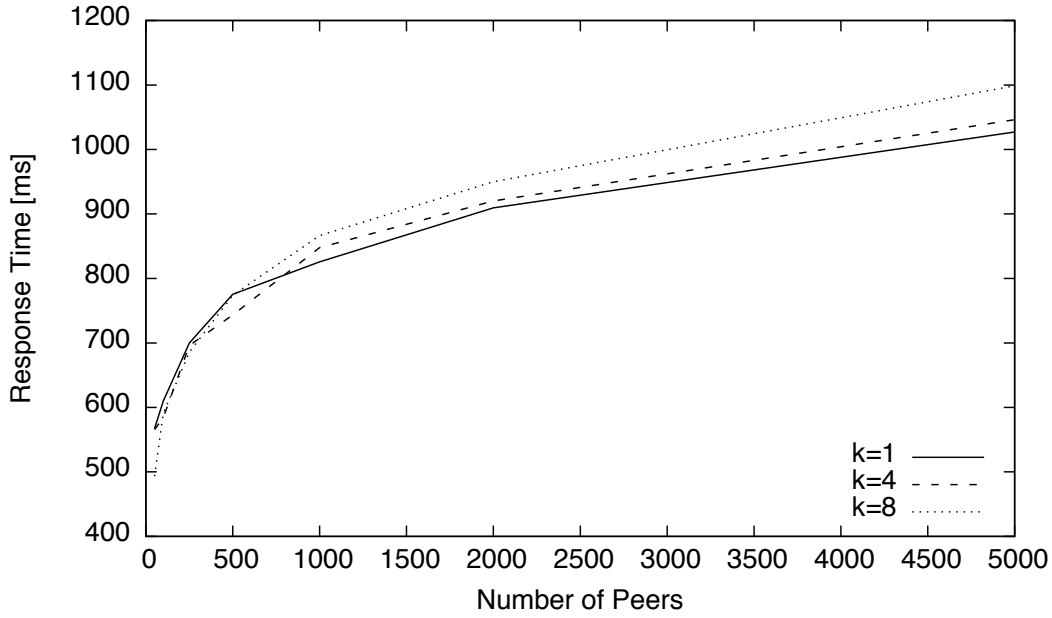



Figure 5.6: Average Response Time of Find *k*-Closest Nodes Operations

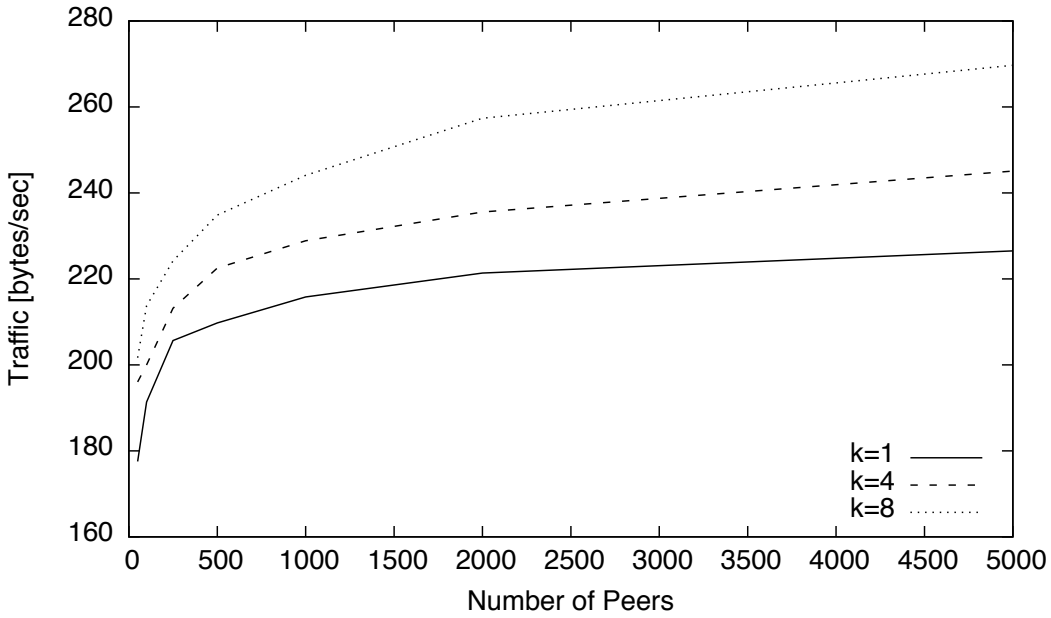



Figure 5.7: Overlay Traffic Per Peer

### 5.4.2 Stability and Efficiency

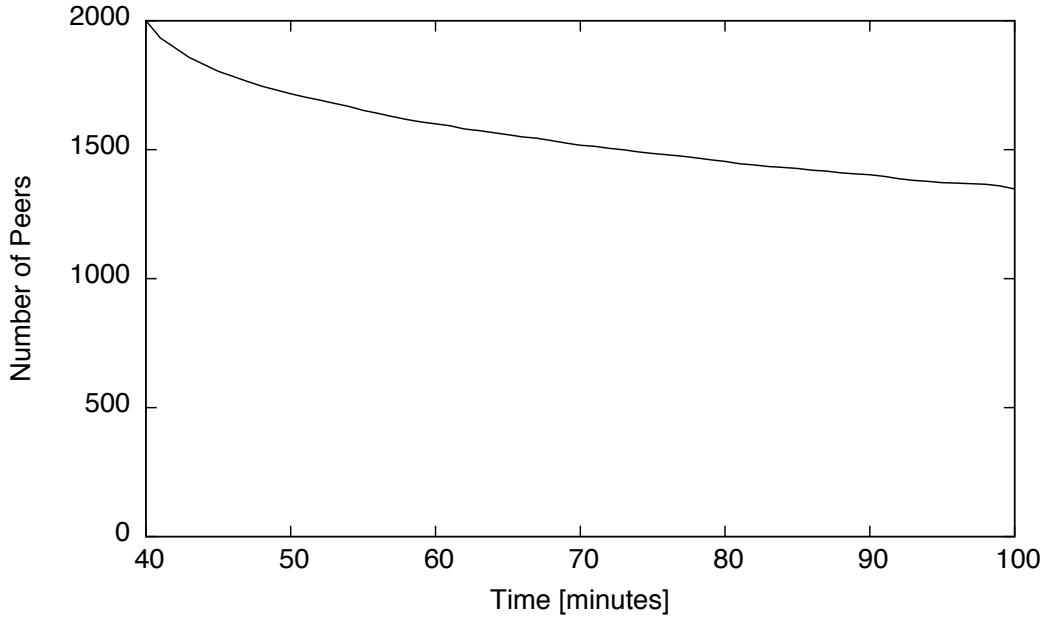



Figure 5.8: Number of Present Peers Per Minute

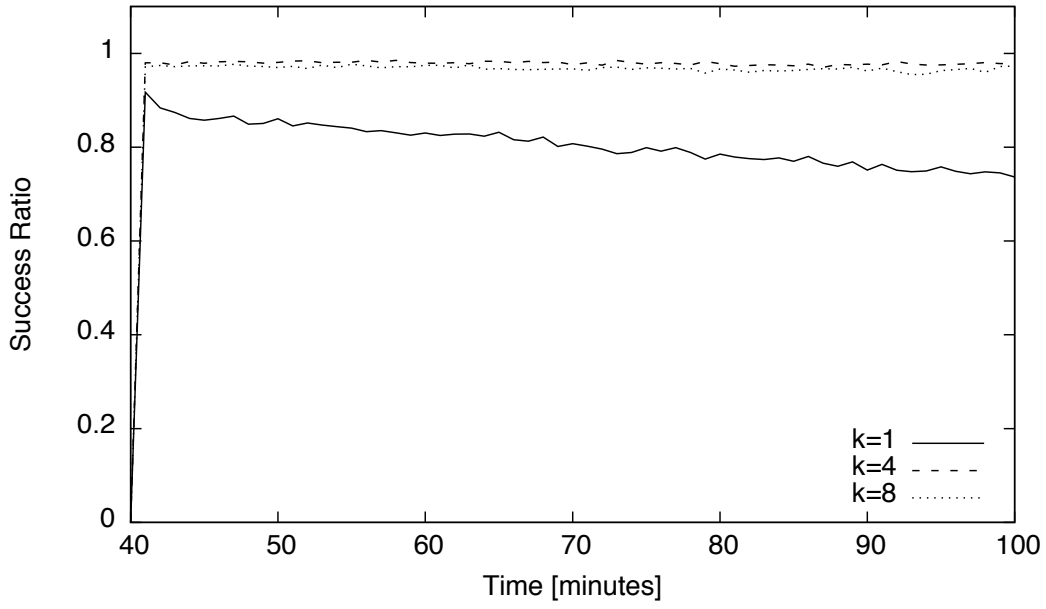



Figure 5.9: Success Ratio of Find k-Closest Nodes Operation Per Minute

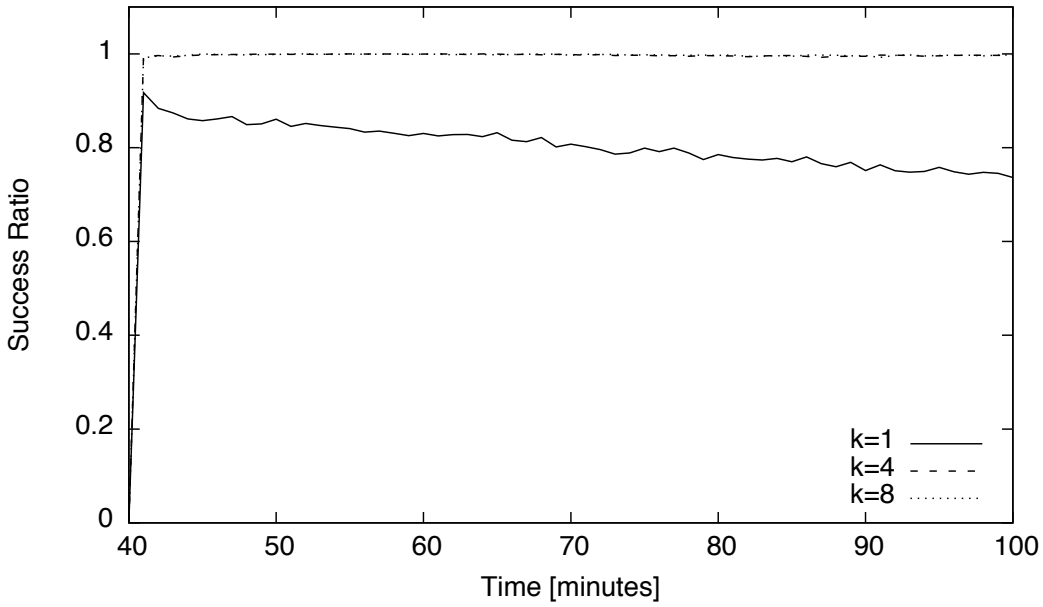



Figure 5.10: Success Ratio of Finding at Least One Node Per Minute

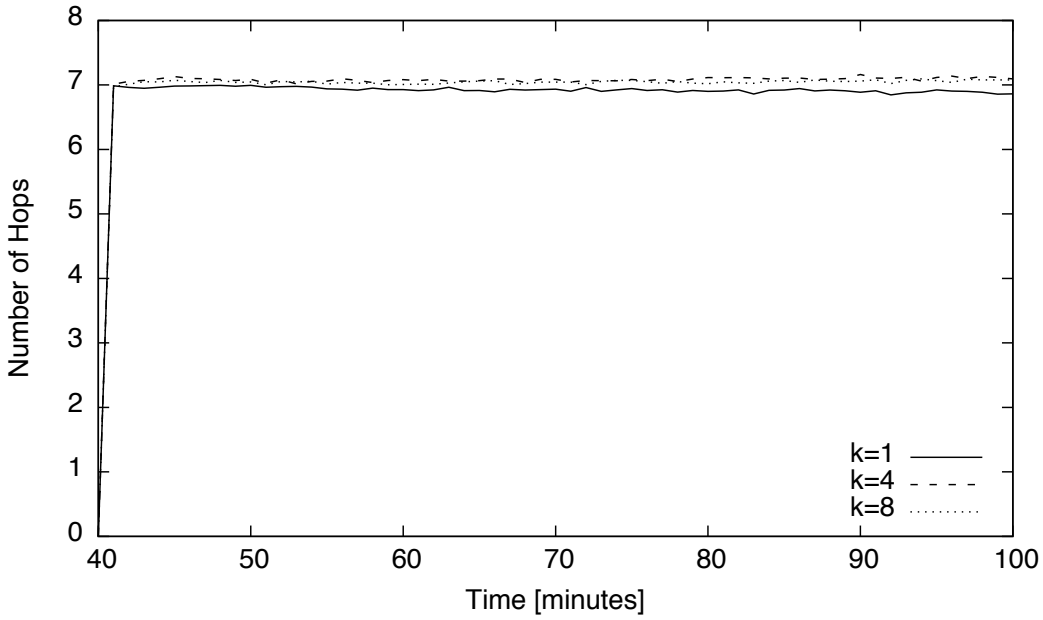



Figure 5.11: Hop Count of Successful Find k-Closest Nodes Operations Per Minute

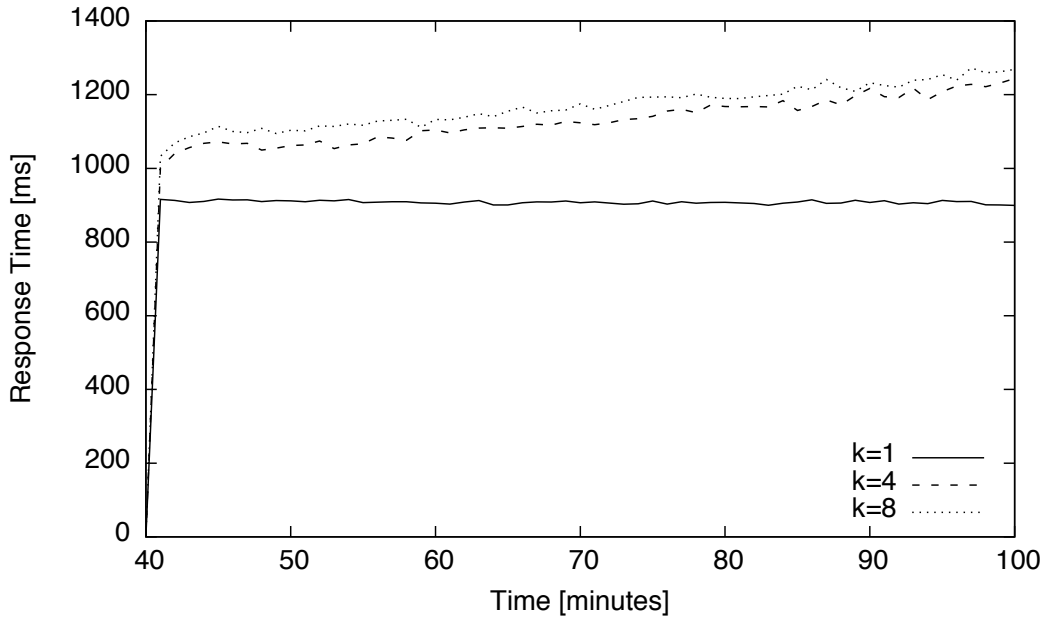



Figure 5.12: Response Time of Successful Find k-Closest Nodes Operations Per Minute

### 5.4.3 Area Search

**No Churn**

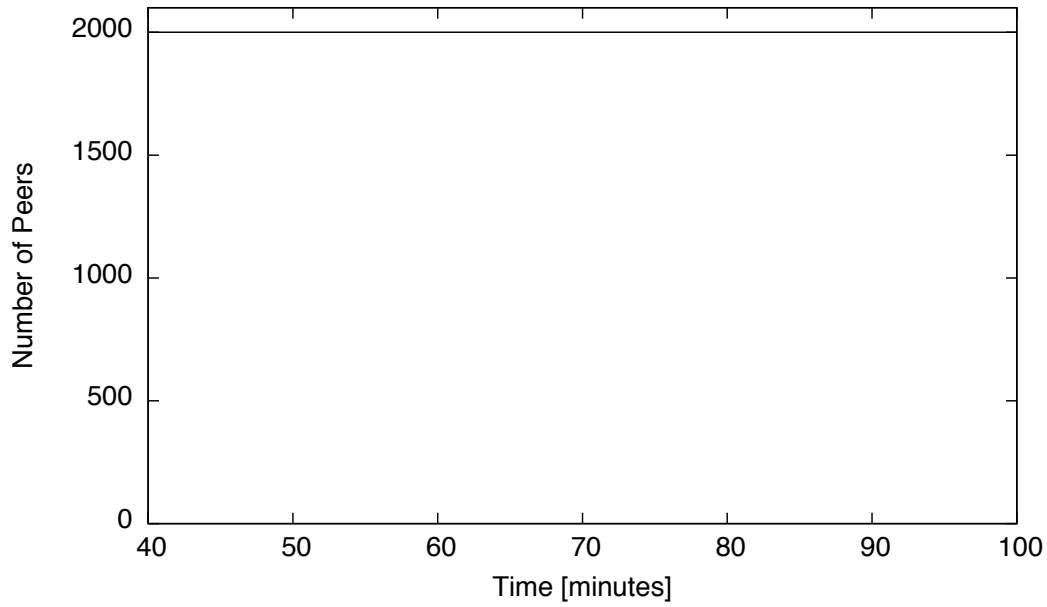



Figure 5.13: Number of Present Peers Per Minute

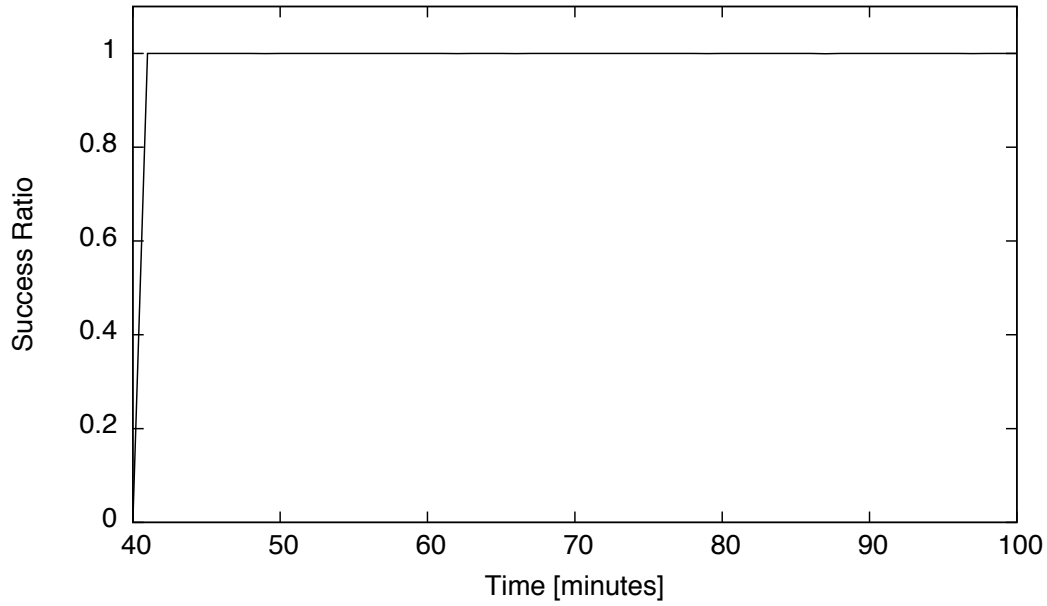



Figure 5.14: Success Ratio of Area Search Operations Per Minute

**Churn**

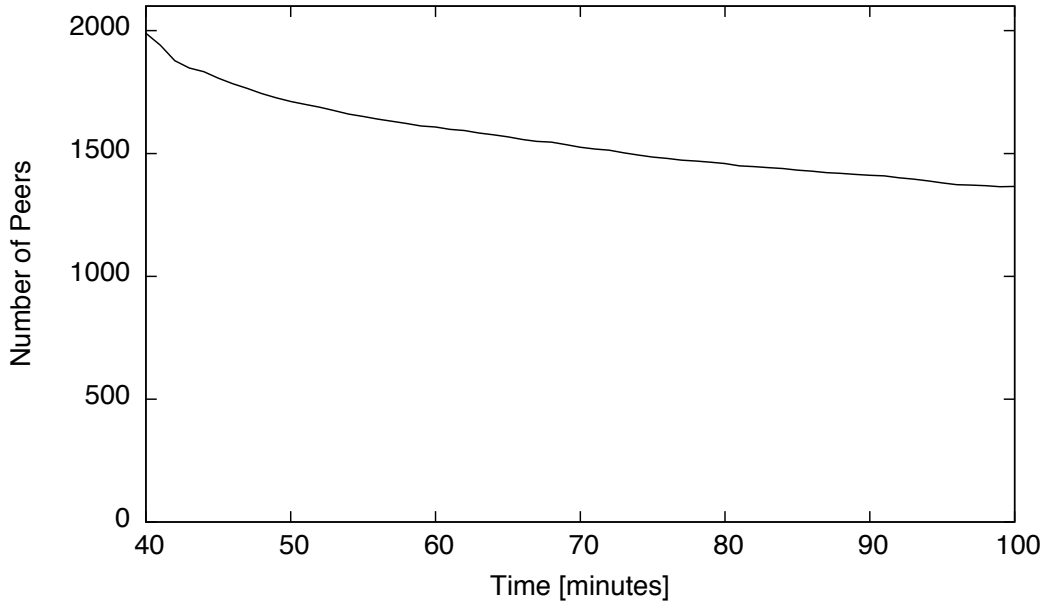



Figure 5.15: Number of Present Peers Per Minute

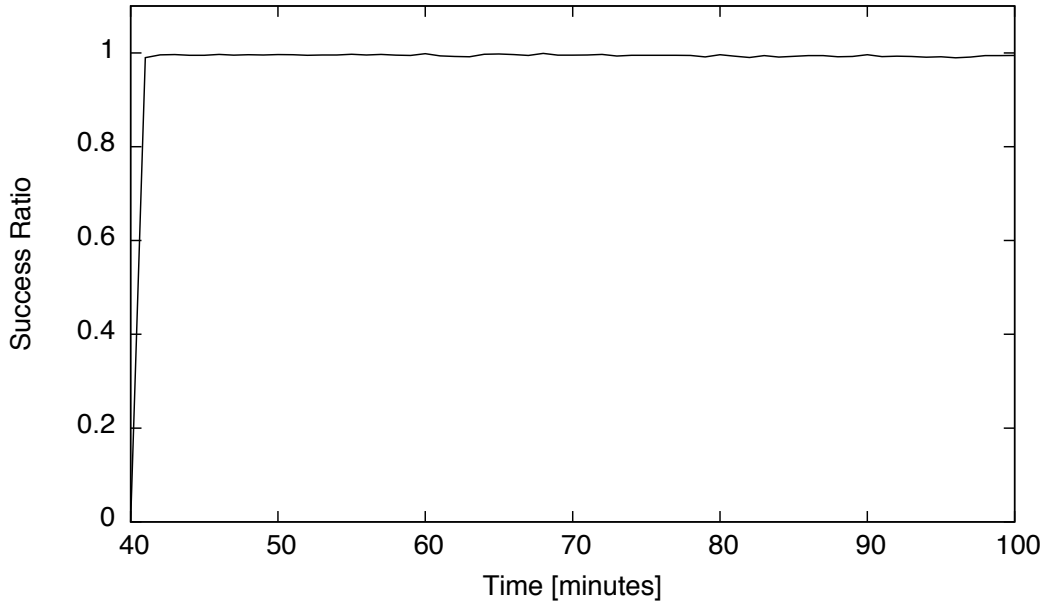



Figure 5.16: Success Ratio of Area Search Operations Per Minute

## 5.5 Analysis of the Results

### 5.5.1 Scalability

In Figure 5.5 we can see that the average hop count logarithmically increases with increasing number of peers in the system. This indicates that the overlay is scalable. Figure 5.6 and Figure 5.7 confirms it, because we can again observe that the curves in both Figures are logarithmic. The first two Figures show that the efficiency for all values of $k$ is fairly the same. The last figure illustrates that for $k = 4, 8$ the overlay traffic per peer is much worse, which is to be expected.

### 5.5.2 Stability and Efficiency

In Figure 5.8, we can see from 40th minute onwards that the number of peers in the system decreases, this has a big impact on the success of the find $k$-closest nodes operation for $k = 1$, as we can see in Figure 5.9 the success ratio decreases. The reason for that is that during the search process for the closest node (to a point) only one node is returned in each step, if $k > 1$ and if the closest node does not reply after a certain time, the operation would choose the next best node to contact (see Chapter 3), but for $k = 1$ the search is over if a node on the search path does not reply. For $k = 4$ and $k = 8$, this problem does not occur and the success ratio is close to 1. Figure 5.10 shows that the find $k$-closest nodes operation is successful at finding at least one node out of the $k$-closest nodes when $k = 4, 8$. Figure 5.11 shows that the hop count of successful find k-closest nodes operation per minute is almost the same for each value of $k$. While the hop count is more or less the same the response time increases as shown in Figure 5.12, the reason for that is that for $k > 1$ the next best node is chosen after a connection cannot be established with the closest node.

All in all we can say that the results indicates that the overlay is stable in terms of churn when using $k = 4, 8$. We can observe a small increase in response time, but apart from that the efficiency does not suffer under churn.

### 5.5.3 Area Search

The area search operation is tested with the assumption that each peer stores location-based information, so when a node contacts every peer in an area he has found every data objects in that area. The test without churn shows that each area search operation per minute is successful. The test with churn shows that the value is very close to 1, because it internally utilizes the find $k$-closest nodes operation, which is analyzed in the previous Subsection.

# Chapter 6

# Conclusion

During this thesis a novel location-based peer-to-peer overlay was developed and implemented in the overlay network simulator PeerfactSim.KOM which is called GeoMesh. Related work has been considered, in order to design an overlay which is efficient, scalable and stable at once. Most of the work on this topic has problems under changing conditions when peers join and leave the overlay which is often referred as churn. These problems are discussed in Chapter 2. Also, overlays such as Geodemlia and SpatialP2P have been considered. Both overlays are efficient and scalable at once, so the main focus was on both overlays which also influenced the design of GeoMesh. Chapter 3 begins with definitions and explanations, which should depict the requirements of such an overlay. The join, leave, search methods are important components in every peer-to-peer overlay, because a P2P overlay network is a self-organizing network. So, these procedures and functions are also presented in Chapter 3. Location-based overlays uses peer positions to find location-based information, so methods such as find $k$-closest nodes or area search are important components. In Chapter 4, the implementation is presented, which shows how GeoMesh can be implemented in the overlay network simulator PeerfactSim.KOM. Subsequently, the overlay is evaluated in Chapter 5, which shows good results in terms of scalability, efficiency and stability.

## 6.1 Future Work

The current version of the implementation does not support retrieving data objects, this can be further improved, also another improvement would be data replication in which a peer can store his location-based data in his neighboring peers, so data objects are permanently present in the system.

# Bibliography


[AB09] Shah Asaduzzaman and Gregor v Bochmann. Geop2p: an adaptive and fault-tolerant peer-to-peer overlay for location based search. *The 29th IEEE ICDCS*, 2009.

[Bea] Calculate distance, bearing and more between latitude/longitude points. `http://www.movable-type.co.uk/scripts/latlong.html`. Accessed: 2015-05-20.

[ErJ01] D Eastlake 3rd and Paul Jones. Us secure hash algorithm 1 (sha1). Technical report, 2001.

[GSR+12] Christian Gross, Dominik Stingl, Björn Richerzhagen, Andreas Hemel, Ralf Steinmetz, and David Hausheer. Geodemlia: A robust peer-to-peer overlay supporting location-based search. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 25–36. IEEE, 2012.

[KSS09] Verena Kantere, Spiros Skiadopoulos, and Timos Sellis. Storing and indexing spatial data in p2p systems. *Knowledge and Data Engineering, IEEE Transactions on*, 21(2):287–300, 2009.

[SGR+11] Dominik Stingl, Christian Gross, Julius Rückert, Leonhard Nobach, Aleksandra Kovacevic, and Ralf Steinmetz. Peerfactsim. kom: A simulation framework for peer-to-peer systems. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 577–584. IEEE, 2011.

[Sin84] Roger W Sinnott. Sky and telescope. *Virtues of the Haversine*, 68(2):159, 1984.

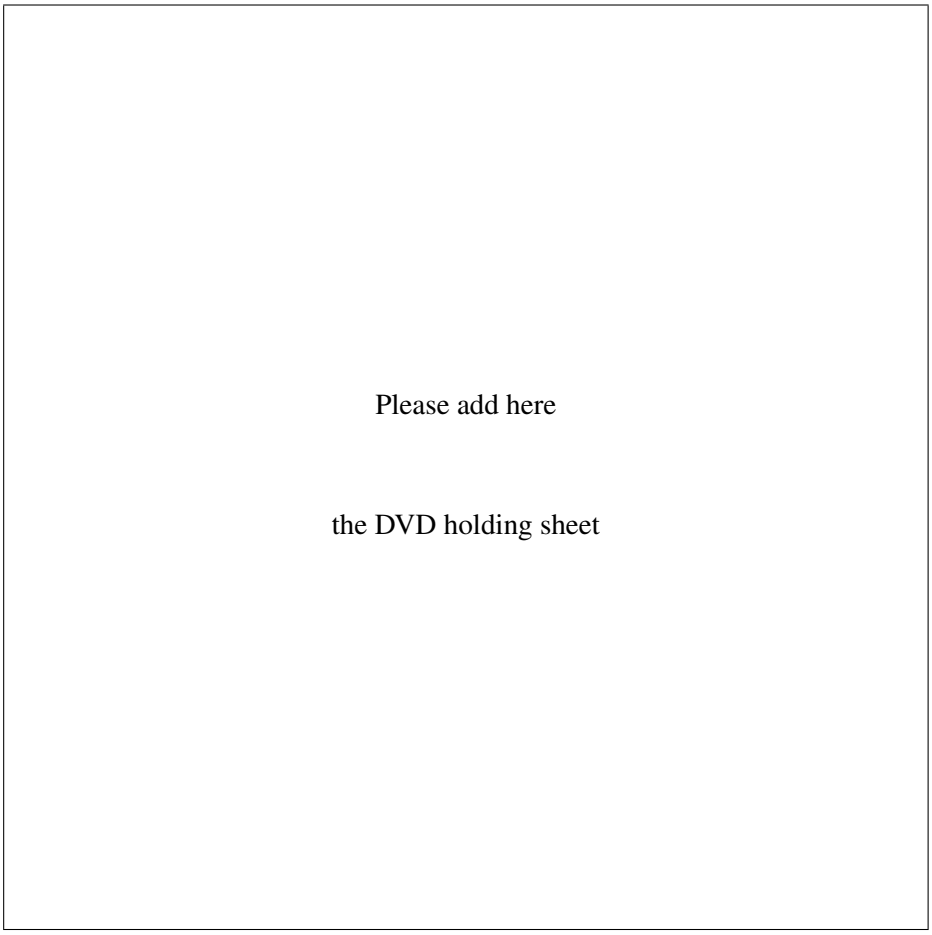
[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, 2003.

[THS07] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16(2):165–178, 2007.

# 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, 22.Juli 2015 Mustafa Yilmaz




**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