A performance evaluation of the Kad-protocol

Master Thesis

by

René Brunner

from Sinsheim

submitted at

Corporate Communications Department

Prof. Dr. E. Biersack

Institut Eurécom

France

Lehrstuhl für Praktische Informatik IV

Prof. Dr. W. Effelsberg

Fakultät für Mathematik und Informatik

Universität Mannheim

Germany

November 2006

Supervisor: Dipl.-Wirtsch.-Inf. Moritz Steiner

Acknowledgements

First, I would like to thank my supervisor Prof. Dr. Ernst Biersack, who gave me the honour to work at the Eurecom Institute. His comments and feedback were a great enrichment for this Master Thesis as well as for my research experience.

I am very grateful for the assistance and support of my advisor Dipl.-Wirtsch.-Inf. Moritz Steiner. His expertise and advices supported me in many ways.

I want to express my gratitude to Prof. Dr. Wolfgang Effelsberg who made this extraordinary exchange between the University of Mannheim and the Institute Eurécom possible.

Finally, I would like to thank all people, who supported me directly or indirectly. I am very grateful for the kindness and helpful environment during my stay at the Institute Eurecom.

Abstract

Most efficient peer-to-peer protocols deploy structured overlay networks based on Distributed Hash Tables (DHTs). These have been extensively studied through theoretical simulations and analysis over the last few years. Recently, the popular eMule and aMule file-sharing applications incorporate a widely-deployed Kademlia-based DHT, called Kad. The Kad-network with over a million simultaneous users enables the observation of its behaviour in practise. This Master Thesis consists of two main parts. Firstly, it describes and observes the structure and processes of the Kad-protocol in detail. Secondly, it empirically evaluates the performance of Kad with the focus on the data item distribution process.

Contents

List of Algorithms

List of Figures

List of Tables

Abbreviations

CCDF	Complementary cumulative distribution function
CDF	Cumulative distribution function
CPU	Central processing unit
DHCP	Dynamic host configuration protocol
DHT	Distributed hash table
DSL	Digital subscriber loop
e2dk	eDonkey
GMT	Greenwich mean time
GNU	GNU's Not Unix
ID	Identifier
ID3	Identify an MP3
IP	Internet protocol
Kad	Kademlia-based protocol
MD4	Message-Digest 4
MP3	MPEG-1 audio layer 3
NAT	Network address translation
P2P	Peer-to-peer
RTT	Round-trip time
SQL	Structured query language
TCP	Transmission control protocol
UDP	User datagram protocol
UML	Unified modeling language

1. Introduction

Peer-to-peer (P2P) applications continue by many measures to be one of the most important applications in the internet. The most representative P2P-based applications are file sharing systems, besides storage and communication systems. P2P accounts for more than 60% of the internet traffic at the end of 2004 [5]. Moreover there are actually millions of simultaneous connected users spread out on different continents and states.

This success has motivated the research of new P2P protocol designs. Especially the Distributed Hash Table (DHT)-based are the most promising standard for overlay networks. They provide an elegant distributed solution for an efficient mapping of data to locations. Unfortunately these achievements were not widely-deployed and the research was limited to simulations, theoretical analysis and limited-scale experiments.

Now, decentralised file sharing application gained in popularity and the Kademlia-based protocol Kad connects over a million of simultaneous users. Kad is incorporated by the two filesharing clients eMule and aMule. These make it possible to observe the behaviour of a widely deployed DHT in practise. Often, these results disprove significantly the theoretical tests.

In practise, *churn* the changing availability of the peers [56], can affect the performance of all DHT-based overlay networks. This concerns the routing table, which has to cope with peers, which might be stale or have new contact information. But also the publishing process is affected, when peers with published data are no longer available. The churn is opposed by replication of content or by the frequency of updating the routing table to assure a better quality. However, this is done at the cost of a higher network overload.

In a file sharing application the data distribution process is the one with the most messages and operative traffic. So there is a high potential for performance improvements. For example, a *load* protection system was developed to avoid an overload on popular data items. But as well there are faintness of changing the published data by fraud peers. So, this Master Thesis has its main focus on the distribution performance and the propability of finding distributed data.

First, the background of Kademlia is illustrated in chapter 2. Thereafter structured and unstructured overlay networks are briefly described. The related work to this paper is presented at the end of the section .

Actually there are no or only brief documentations of a real-world Kademlia-based application. Either the protocols are commercial where the source is closed and not accessable or it is an open source project with scanty documentations. In chapter 3 the main architecture and structure of the Kademlia-based Kad protocol will be examined. In the following chapter 4, the iteration and

lookup process will be explained. Chapter 5 details the publishing process, which will be the main intention of the measurement and analysis.

For the comprehension, testing, measurement and analysis of the Kad protocol, an aMule client was modified for special behaviours. This analytical framework is deployed in chapter 6. Furthermore the structure of the database integration is employed, which builds with its billions of entries the base for the profound analysis of the Kad peers and its behaviours.

Chapter 7 illustrates the measurements of the Kad protocol and of the data publishing process. Then the benefits of different ways to improve the publishing performance are analysed. So the scheme of data distribution to different peers has an important role.

2. Background P2P - Related Work

Peer-to-peer (P2P) Internet applications became more and more popular over the last few years. This was also a consequence of the Recording Industry Association of America (RIAA), which forced server based file sharing systems like Napster to shut down [32]. So especially the peer-to-peer file sharing applications are now one of the main sources of Internet traffic [19, 52]. Moreover, a variety of applications took advantage of this peer-to-peer approach. Among these are storage and backup systems, cooperative content distribution systems to Web caching system and also communication applications like Skype [31].

The P2P overlay networks are naturally distributed systems, without any central control or hierarchical structure. Their characteristic is the symmetric role of each peer, which can be a server and also a client. This is the opposite to actual server based structures on the Internet. This can be confirmed by the upcoming and the success of asynchronous-DSL connections, that advance the download and neglect the upload [43].

P2P is characterised by peers in self-organised overlay networks, which overlay the Internet Protocol (IP). They offer different features like a robust wide-area routing architecture, efficient lookup for data or references, selection of other nearby peers, content replication, permanence, hierarchical naming, trust and authenticity, anonymity, massive scalability and fault tolerance [32]. The main characteristics can be summarised to the following:

- *Decentralisation* - many autonomous clients without a central control
- *Scalability* - the system has to adapt large extentions of peers
- *Fault tolerance* - the network must be resilient, especially in regard to *stale* peers
- *Load balance* - routing messages have to be balanced to reduce the network overhead

Another important characteristic is the *security*. However, until now it is not covered in the protocols themselves but it is covered by the application level. Finally there is a low level on security. Especially the DHT-based overlay protocols suffer from man-in-middle and Trojan attacks [32].

All peer-to-peer systems are based on overlay network protocols. These are responsible for the storing and retrieval of references or objects. Therefore it uses a message routing and a list containing other peers of the network. Generally, there are two different types of overlay protocols classified depending on a *structured* or *unstructured* organisation of peers and stored references. The

pros and cons of these two approaches are still debated by the research community [6]. Another study clarifies the differences of these two overlay structures by focusing on their characteristics [46]. The most significant differences are in terms of resilience, message overhead, query performance and load balancing.

2.1. Unstructured overlay network

An unstructured overlay network is formed by peers joining and leaving the network. Thus they have to follow only a few loose rules. The whole network operations are totally decentralised and follow no specific requirements for the network topology or for the data placement. Especially the data placement has no certain properties in contrary to structured overlay networks. To retrieve a data item, a peer will query all its neighbours in a certain radius, also called *flooding*. This design is high resilient and fault tolerant because the whole network will not be interrupted when peers go offline. But in contrary to structured overlay protocols, the "blind" searches cause a large network overload and is not scalable [32].

One of the most popular and investigated application based on unstructured overlay protocol is Gnutella [20]. Early versions of Gnutella implements simple flooding strategies. But newer versions improved query efficiency and reduced the overhead for control traffic by introducing *superpeers/ultrapeers* [37]. An advantage of the Gnutella protocol is that *popular* data is found easily and quickly. But a lot more hops are needed to find *rare* data.

2.2. Structured overlay network

Structured overlay networks topologies are tightly controlled and the data is not placed randomly on peers anymore. Peers will be chosen at a specific position, which makes queries more efficient. Most structured P2P systems use Distributed Hash Table (DHT), which organise the peers and content locations in a certain order [32].

Each peer in network will be assigned a random NodeID, which gives a fixed location to the peer in a large P2P space. Data objects are identified by a unique *key* from the same P2P space. This allows the overlay network a scalable storage and the retrieval of the data by the *key* location. It is important that the location of a peer in the overnet network is abstract and not geographically determined.

The peers in a structured overlay network maintain a routing table with a small subset of peers. Generally, these are neighbouring peers identified by their NodeID, IP address and message port. So a query can progressively approach a *key* by routing through peers with a closer NodeID. There are differnet DHT-based systems, which are differentiated by their data and peer organisation and their routing algorithm. All of these systems have in common that a data object

can be retrieved in $O(log(n))$ hops on average, where n is the total of peers in the network [32].

The main structured P2P overlay networks are: Content Addressable Network (CAN) [47], Tapestry [61], Chord [54] Pastry [51] and Viceroy [34]. The DHT-based Kademlia's [35] XOR topology-based routing is like the first phase of the routing algorithm of Pastry and Tapestry. It will be described, measured and analysed in detail in the following chapters.

After the proposal for new overlay network protocols were published, several studies started to measure the DHT performance under churn: [26, 33, 25, 30, 21, 49]. But in the meantime, also other studies examined that session times differ significantly from the theoretical approach [53, 15, 45, 18, 60]. Especially the introduction of a new parameter *tolerance zone* in the real implementations, opens a new basis for further investigation.

Recently, with the raising popularity of real DHT-based filesharing applications, new studies came up measuring real DHT-based P2P networks [55, 57, 23, 46]. Generally these investigated either the Kademlia based Kad or Overnet [44] network, which had millions of users at that time. These studies are primarily concentrated on the *behaviour* of the peers, the *query performance*, as the *structure* of the routing table. This thesis proceeds with the empirical investigation of the *publishing process optimisation*.

Another approach is the identifying of different attacks on DHT-based networks [41, 28, 29]. These studies show the dangers, which are not covered in the theoretical approach. This thesis will also highlight a possible entrapment, when fraud peers intercept published data.

3. Kad architecture and structure

This chapter describes the main functions and algorithms of the Kademlia-based DHT network **Kad**. It mostly applies to P2P open source project and it is supported by the eMule [11] and the aMule [1] client. Despite its most recent appearance, it already has over one millions simultaneous users, and is thought to be the widest deployed DHT-based protocol.

In the following sections the terms **client**, **peer**, **node** and **contact** will be introduced. The client presents the running application of the peer. A peer is an active connecting point in the Kademlia network, which is applied by the client. A node is another peer with a fixed hash ID in the Kademlia network. The contacts are known peers or nodes, which are guarded by the routing table. Furthermore a contact which is already known by the client, but it cannot be ascertained wheather he is alive or not, will be called a **possible contact**.

3.1. Kad background

During the last years, different file sharing applications came up with new or modified Kademlia-based overlay networks. Often the networks of these applications are not compatible amongst each other, because they use different operation codes or **opcodes**. However, the main functions and the principal algorithms are the same, as their implementations are based on the same overlay protocol.

3.1.1. Kad networks

Besides the Kad protocol, also other similar Kademlia-based overlay networks exist. Overnet [44] was the first applied among these. Unfortunately it is implemented commercially by the eDonkey2000 client [10], which has a closed source code. But other open source applications like aMule/eMule, MLdonkey [38] and KadC [36] reverse-engineered partially the Overnet protocol. RevConnect-KAD [48], part of the opensource filesharing application RevConnect, is based on the open source protocol eMule-Kad. Finally there exists two main families of Kademlia protocols, the Overnet-based and the Kad-based. Recently BitTorrent [4] also started to implement a Kademlia-based protocol for its search algorithm.

Most of these described Kademlia-based clients are **Hybrids**, which means that they support the server based *eDonkey* protocol [16] besides Kademlia. Afterwards, as the Kademlia support was added to eDonkey, both protocols have the same standard functions. These are for example the socket implementation, the downloading process or the user interface. Often these are mingled

together in the aMule source code, without any clear separation. But as well as the implementations of the Kad protocol was strongly inspired by the eDonkey specification [22].

The difference between the two clients aMule and eMule is that aMule runs on nearly all system environments in contrary to eMule which is only designed for Windows. This report describes the Kad protocol of the aMule client, because applying the tests and analysis on Unix has several advantages. At the beginning eMule and aMule started with an identical Kad implementation, but as these continue their development, similar behaviours may vary.

3.1.2. Network protocol

Kad uses UDP as message protocol, which allows one peer to send messages very quickly and uncomplicated to another peer. This design advances the Kad protocol because the operation and iteration process consecutively induces a peer to send many messages. Furthermore, there is no need to establish a session, because generally a peer sends only one request to another peer. When it obtains a response, it will iterate to the next peer. Also the client has to handle with *churn*, so stale peers would slow the establishment of a TCP connection. In contrary UDP sends messages to other peers and it lets the application level handle with the stale peers.

As the peers join and leave the Kad network like they want, it is impossible to make longer sessions or relations with other peers. So a client has to always be aware that his contacts have left. The solution is a timer, which takes all control of the messages. It terminates the operations like a iteration, search or publishing process after a defined timeout. But the timer also periodically induces different processes like the republishing of files or the checking for a firewall. Appendix A lists all parameters for the timer.

Each UDP packet starts with one byte called ID, which identifies the Kademlia protocol. aMule-Kad and eMule-Kad use `0xE4` for standard packets and `0xE5` for zlib-compressed packets. The following byte is called **opcode** and represents the operation code for the Kad protocol. Generally the opcode defines the different types of requests or responses. A list with all possible opcodes can be found in the appendix B. Overnet uses, contrary to the Kad protocol, the packet identifier `0xE3` only and does not support compressed messages.

An overlay client has typically two port numbers, each for a different message type. One port is used to send and receive messages of the standard and service operations. aMule and eMule therefore use the UDP port 4672 as a standard port for the Kad protocol. The other standard port is 4662 and it uses the TCP for uploading and downloading files. Furthermore, this port functions like the firewall check for other services. These two ports are referred respectively to as **messaging port** (UDP) and **service port** (TCP). Optionally, the two standard port numbers can be changed by the users.

0101	
1100	
1001	distance is 1 + 8 = 9

Table 3.1.: XOR Example

3.1.3. Kademlia 128-bit space

The Kademlia protocol has a space of 128 bit, this means that there is the possibility to place 2^{128} different objects. The most important objects to place are the individual peers. Therefore the peers have a corresponding client Id (see section 4.1.3), which determines the position in the *Kad-space*. As the space is virtual, geographically close peers can be far away in the Kademlia space. The passive objects like files and keywords also obtain an ID to place them in the virtual space. But they need a **host peer** near to them, which takes responsibility for them.

The XOR metric is applied [35] to find out the distance between the individual objects. Within this method, differences in higher bit positions will represent a larger distance than differences in smaller bit positions.

Figure 3.1 illustrates in detail the XOR calculation to find out the distance between two objects. Peer S calculates the distances to four other peers in the 4 bit number space. The result shows clearly that objects in the same subtree are *closer* to each other than they are to other objects in other subtrees.

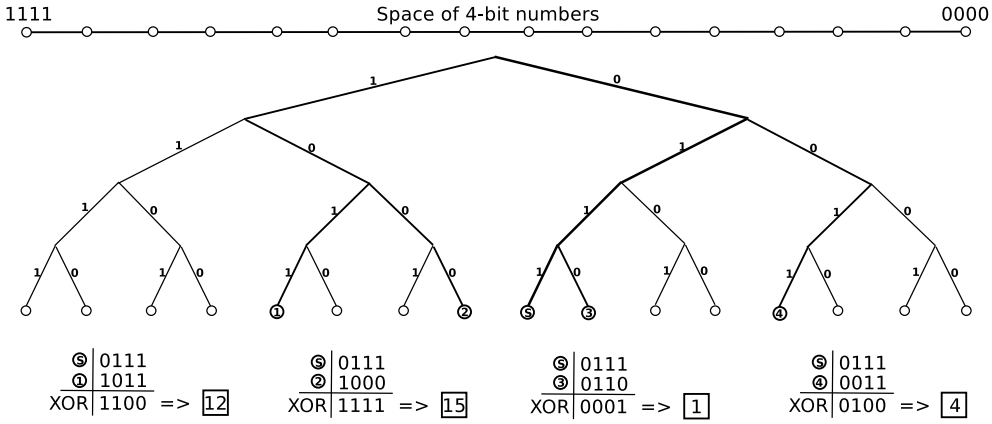

Figure 3.1.: The binary tree is a simplified representation of a theoretical routing table with the XOR-metric. In this example the starting peer S calculates his distance with the XOR-metric to the peers 1, 2, 3 and four. As a result peer 3 is the *closest* with a XOR of 1 to the peer S. So positions in the same subtree are much closer together than they are to positions in other subtrees

Name	Type	Description
`contact-ID`	128-bit number	A unique hash identifier randomly chosen
`distance`	128-bit number	The XOR distance to that peer
`IP`	number	The IP address of the peer
`UDP`	number	The UDP port (standart 4672)
`TCP`	number	The TCP port (standart 4662)
`type`	number	Quality of the peer
`last_type_set`	date	Last type check
`inUse`	number	Count current actions to that contact
`creation`	date	Time of the contact creation
`expiration`	date	Lifetime of the peer

Table 3.2.: List of attributes for an individual contact in the routing table

3.2. Routing table

The routing table organises the **contacts** of a peer into several ***K*-buckets** which are the **leaves** of the routing tree. Again these K-buckets are organised by the **routing zones** which are all **nodes** of the routing tree. Each K-bucket represents one subtree and has K representative **contacts**. So if the client wants to locate a node in another subtree, it will hop to an acquainted node from that bucket.

3.2.1. Contacts

Each peer has a list of known peers, called **contacts**, which are structured by the routing table. The maximum contacts of a peer is limited by a variable to 5000. All attributes of an individual contact are listed in table 3.2.

Each peer has a `creation` and `expiration` date, which helps to eliminate non-responding or disconnected peers. Furthermore, it has a `type` and a `last type set` variable, which are responsible for the degree of availability over the time. Besides the `contact-ID`, the `XOR-distance` of the `client-ID` and the `contact-ID` is calculated at the beginning. This avoids the CPU to calculate the distance within each utilisation. The `inUse` variable indicates how many current actions are running with that contact. This prevents the cancelling of a contact, when it is included in a lookup process.

`Type` has a scale from 0 to 4, where 0 is the best and means that this client has a very good availability over the time. On the contrary 4 means that this client is rarely connected and will propably be eliminated at the next occasion. So the contacts pass through different states described in table 3.3. Their `type` will within its creation be initialised to 3. Afterwards, when the client receives an alive sign from the contact it will improve the type to 2 otherwise type 4 will be assigned. For example when a client receives a message from a contact which it already has for longer than two hours in its routing table, the contact will obtain type 0.

Type	Preview lifetime (hours)	Description
0	2	longer than 2 hours active
1	1.5	active since 1 -2 hours
2	1	less then 1 hour active
3	0	Just created
4	-	Preview for its deletion

Table 3.3.: Shows the different types of the contacts with their lifetimes

The clients are saved in the file `nodes.dat`, which can be read by most Kademlia clients like eDonkey, eMule and aMule. It serialises the ID, IP, UDP port, TCP port and the type of the contacts. This file can contain maximal 2^{32} contacts [1]. But, as mentioned above the contacts are limited in the source code to 5000. Normally there should not be a client with the type 4 in the file, because then it is considered for elimination. But anyway if there is one, it will be ignored by the reading of the contact list. There is a regularly updated `nodes.dat` file with about 114 nodes in the internet to download [42]. That can be used to accelerate bootstrapping, which is faster with several clients, than with only one.

3.2.2. Structure

The Kademlia routing table is a *proper binary tree*, which means that it has either zero or two children. But it is not a *perfect binary tree*, where all leaves have the same depth. Due to this incompleteness, the routing table needs to know only a small subset of peers in the network like figure 3.2 shows.

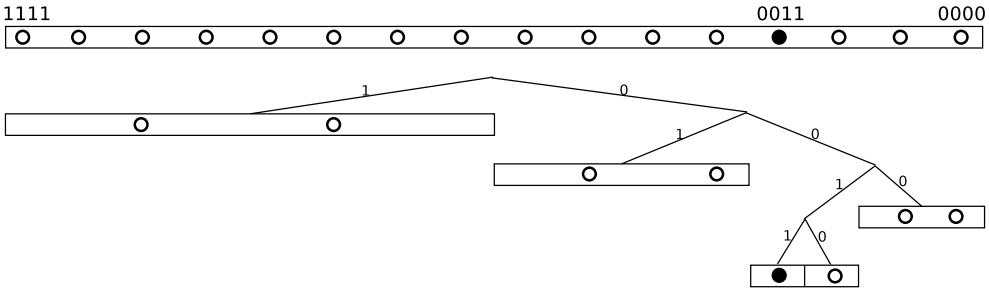

Figure 3.2.: Presents a pseudo routing table for a simplified 4-bit number space. The ***K*-buckets** represent subtrees with a group of K contacts.

In reality the Kad routing table differs from the original Kademlia routing table. In principal, the Kad calculates the XOR distance to the own client ID. As a result the distance of the peer to itself is 0 and to the other peers a 128 bit number. This simplifies the illustration of the routing table (see figure 3.3) and above all, the structure and implementation of source code. The remaining methods have the same layouts like the methods in the original Kademlia [35].

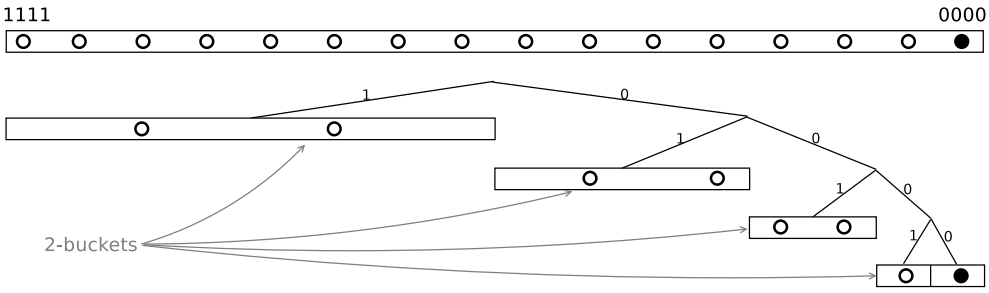

Figure 3.3.: Contrary to the previous routing table, the *XOR-distances* to the peer, owning the routing table are illustrated here. It has the address 0000, because the *XOR-distance* to itself is obviously zero.

The real routing table of the Kad protocol will look like in figure 3.4 when it is complete. That means that it contains nearly 6360 contacts. But in reality it is impossible to reach that number, because this would mean that there were 2^{128} peers with a unique hash ID.

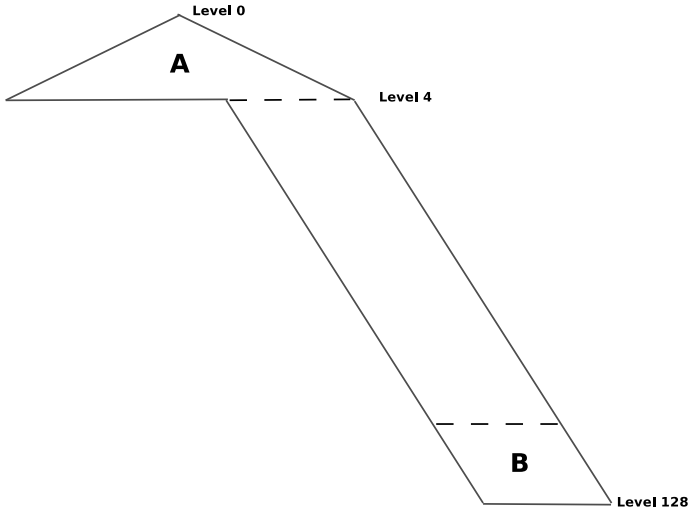

Figure 3.4.: Shows the shape of the Kad routing table. Subshape **A** is presented in detail in figure 3.5 and subshape **B** is described in figure 3.6

Kad divides the table structure in levels and positions. The levels indicate the layer beginning from the root, which has level zero. For example the highest level of a 4-bit number space like figure 3.5 would also be 4. Accordingly, a 128-bit number Kad-space has maximal 128 levels. The position indicates the n-th node in a level starting from the table holding node zero. So level four has nodes with the postions from 0 until 15. The higher levels are limited from 0 until 9.

The complete routing table spans 128 levels. Beginning with level 5 the positions of each of the following levels are limited to maximum 10. This weighs

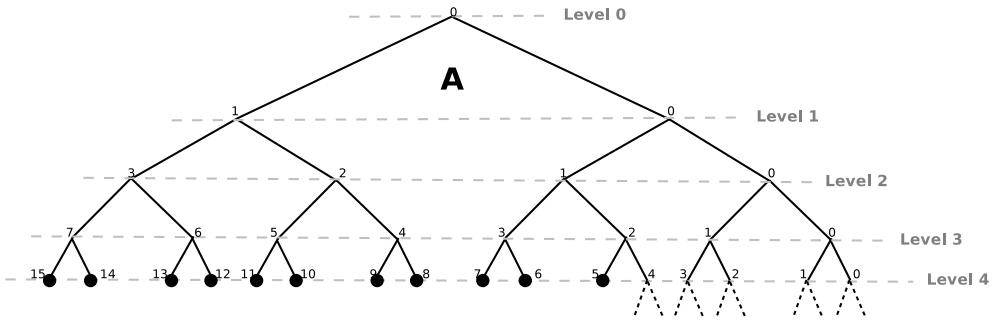

Figure 3.5.: This is an extract of the whole routing table beginning from the root until level four. The positions of the routing zones are spread out horizontally. Level 4 has for example routing zones from position 0 until position 15. Where position 5 until 15 has K-buckets and the zones from 0 until 4 have more subtrees.

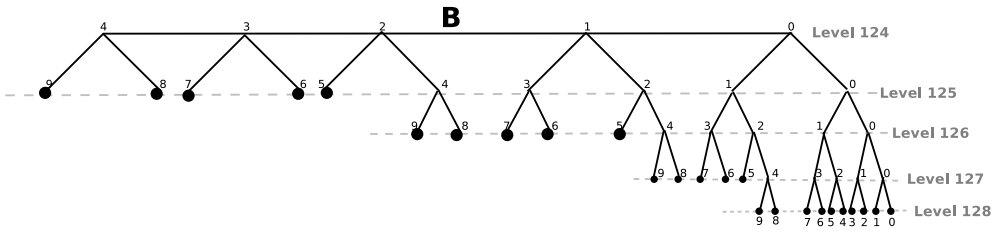

Figure 3.6.: This extract of the routing table shows the closest peers for the peer holding the contacts. These are guarded in the higher levels.

the tree more to the side of the owner peer, which has the XOR-distance 0 to itself. It also enables the structure to keep disproportionate to closer contacts. The maximum possible contacts organised by the routing table are calculated in the following way:

- Total contacts = (Total of buckets) * K
- Total contacts = (leafs of level 4 + leaves on level 5 until 127 + leaves at level 128)* K
- Total contacts = $(11 + 123 * 5 + 10) * 10 = 6360$

Therefore the maximum number of levels is 127 and the maximum number of zones per level are 9. However, by the fourth level more buckets are added than the official Kademlia proposes. This improves the number of lookup hops and is described by Stutzbach as *discrete symbols* and *split symbols* [55]. The maximum K-buckets in the routing table are 636, which are multiplied by *K=10* to 6360.

3.2.3. Insert node

Each time a known node is seen by a client, it will be set *alive* by updating the `expiration` date with the lifetime described in table 3.3. Otherwise, if the node is an already known contact, it will be inserted into the routing table. New contacts are obtained either through a bootstrap request, where 20 random contacts of another client are returned or left within the standard iteration process. The iteration process motivates the client to send requests to peers, which will reply with several closer nodes to a determined target. The client inserts these immediately and without verification into its routing table. Of course there is also a third possibility, that the client is passively contacted by an unknown contact.

A client will only insert a new contact into its routing table, when the *contact-ID* is non existant. Otherwise the old contact will be overridden by the new information. Even when the random initialisation of the client-IDs makes it extremely rare, the possibility remains that the peers will have the same hash ID. But principally what happens is that peers changing their internet connection will reappear with new contact information $< IPaddress : messageport >$.

Algorithm 1: Insert node

```
Data: Contact
root.addContact(0);
 addContact(level);
if is leaf then
    if contact exists then
        update contact information expiration time, IP address, ...);
    else
        if bucket not full then
            add Contact to bucket;
        else
            can split split;
            subtree[bitnumber(level)].addContact(level + 1);
        endif
    endif
else
    subtree ← subtree corresponding the bitnumber at position=level of
    the contact;
    Increment level;
    subtree addContact(level);
endif
```

To insert a new contact, the client runs along the routing table according to the contact's distance. Therefore it checks the n-th position of the 128 bit number, where n is the current level of the routing zone. If the value is zero it will continue in the right child, which contains the closer contacts. When the distance to a leaf is covered, there are two possibilities; either the leaf, which is

a k-bucket, contains less than k contacts and the new node is inserted in this bucket. Otherwise, when the bucket is full with k contacts, the leaf is *split*. However, at level 5 of the routing table, only the first 5 position are allowed to split.

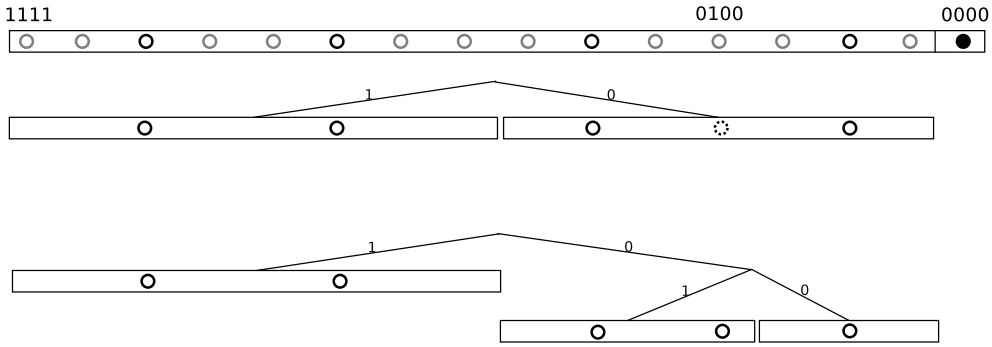

Figure 3.7.: In this example, a 2-bucket obtains a third contact. Then the contact table will split the leaf in two new leaves. The three contacts are divided to the new leaves.

The splitting of a leaf on level n will add two new leaves on a higher level $n + 1$. All contacts which have $bit = 1$ on position n+1 will be added to the left leaf and the others with a $bit = 0$ will be added to the right leaf with nearer contacts.

3.2.4. Routing table maintenance

The Kademlia protocol is intended to remove stale contacts from the routing table to improve the performance, which is disturbed by the churn. It is proven that with the increase of the available time, the propability becomes greater such that a peer stays connected. So the routing table prefers peers which are connected for a longer time. However when a new peer is found it replaces the peer of a bucket, which was seen for the shortest time. This improves the whole quality of the contacts in a bucket [35]. But the Kad protocol does not implement this feature. It follows other strategies in that each client is checked periodically in regard to its lifetime. Therefore the client sends a hello request to the contact, which has 2 minutes to respond. Otherwise the routing table removes this contact. When the tree is no longer equal after an elimination, it executes a *merge* of leaves.

Finally, the maintenance has two main objectives, to sort out stale contacts and to prevent attacks. Without the actualisation of the routing table, bogus contacts could nest among the contacts and distribute faulty information.

The routing table continually improves it's structure by looking for more contacts. Therefore two different strategies exist: a **random lookup** and a **self lookup**. The random lookup is started in order to find new contacts, which correspond to a certain bucket. So a search is created with an artificial target,

which has the same first bits like that bucket. This lookup is only executed for buckets having less than five identical bits. A self lookup is started to find newer contacts with five or more identical bits. The target in this case will be the clientID of the peer itself. The executed search is described in detail in section 4.2.

3.2.5. Number of users and files

aMule has its own algorithm to calculate all the nodes in the Kad network. However, it is vague because it depends on the constellation of the routing table and their contacts. That is why it may happen that a client with about 270 contacts indicates once that there are 1.4 million peers and another time that there are 20 million peers.

The number of Kad users is calculated for each leaf of the routing table. Then the official published result will be the maximum of the results. Two methods will define the total peers, each begins the calculation at a leaf. When the leaf is in a smaller level than five, which is generally only the case after the starting of the client, it will take the maximum possible users. This would be $2^4 * k$ for a leaf of level four, which corresponds to a maximum of 16 leaf with each k contacts. Otherwise, when the leaf has a higher level it calculates the total amount of a subtree, which includes that leaf and extrapolates the result:

- $Totalusers = 2^{(subtree \quad level)} * k * \frac{contacts \quad of \quad subtree}{10 * 10}$

As the clients calculate the hash IDs randomly, they are proportionally dispersed in the Kad-space. So the total of the peers will lead to a nearly balanced binary tree. Again this signifies that the level of the tree must be nearly the same for each leaf. Therefore the quantity of root nodes on the same level is calculated. This is multiplied by the size of the buckets. Then this is multiplied by the representative contacts for that subtree.

There is no real way to find out how many files exist in the Kad network. The difficulty is to get the average files per peer. However, the client counts his indexed keywords, references to sources, to have an approximate average of files per peer. If the client rests while connected, this value can get approximately representative, but it will never be exact. Finally, the total number of existing files in the network is the average per file multiplied by the total amount of users.

3.3. Initialisation processes

The connecting of a peer to the network requires the setup of the state of the peer. So the initialisation process is to establish a peer in the network. After that, the setup processes help to maintain the connection.

3.3.1. Bootstrapping

To join the Kad network a peer must first go through a bootstrap process. Therefore a Kademlia client needs at least one active node in the Kad network with its $< IPaddress : messageport >$. As a response to a bootstrap request the demanded peer will reply with a list of further nodes from the network. There are also some internet pages like overnet [44] which keep the `nodes.dat` (see section 3.2.1) with about hundred actual notes ready for download. Due to this file the initialisation process will be accelerated and facilitated. After containing one or several nodes an aMule client works autonomous and finds the nodes, which are of interest for its routing table.

In particular the bootstrapping process works like it is shown in figure 3.8. The peer which wants to bootstrap from another peer sends his data within the KADEMLIA_BOOTSTRAP_REQ in the format: $< ClientID : senderIP : messageport : serviceport : type >$ to the node $< targetIP : targetmessageport >$. The `type` of the peer is set to zero because the receiving peer will calculate the `type` value when the sender peer is added to its routing table. After receiving the request the peer will reply with the `KADEMLIA_BOOTSTRAP_RES` message in the format: $< n* < ClientID : senderIP : messageport : serviceport >>$. Here `n` is fixed to 20 and defines the quantity of nodes which the routing table will randomly return. The already obtained `type` information will be handed over and the demanding client overtakes them. Evidently, this list contains only n of his own contacts without itself.

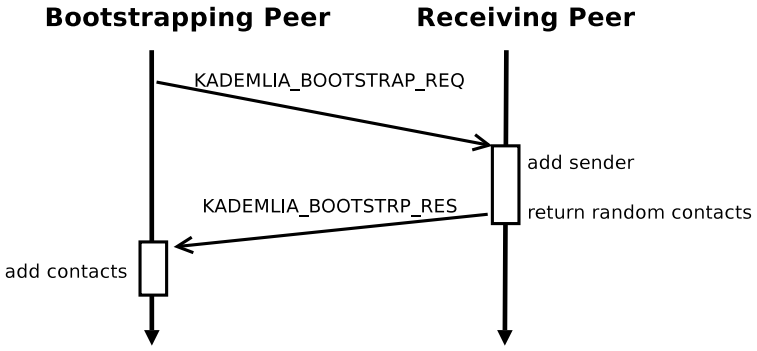

Figure 3.8.: The bootstrap process

After completing a successfull transaction, the demanding peer adds the new contacts to his routing table (see figure 3.2.3). Then the bootstrapping client will check the obtained nodes for a valid `IP address` and `port` number but not if they are still connected. This will be assigned to the maintenance process of the routing table, which will assume the transferred type.

3.3.2. Initial handshake

When a client obtains new contact data, e.g. after a bootstrapping, he will perform an initial hand shaking process. This is a typical procedure for clients or peers to come into contact and to assure that the other is still alive. Therefore a peer sends a KADEMLIA_REQ $< TYPE : targetID : receiversclientID >$ to the new contact. In return, when this one is still online, it will reply with a KADEMLIA_RES message: $< TYPE* < targetID : TYPE : PEER >>$. If the handshake is executed after the starting of the client, then it will send a firewall check message and start the process described in section 3.3.3. Otherwise, when the firewall check is already terminated the client will send a KADEMLIA_HELLO_REQ $< senderPEER >$ message to the opposite peer. In turn this one will return the KADEMLIA_HELLO_RES $< receiverPEER >$.

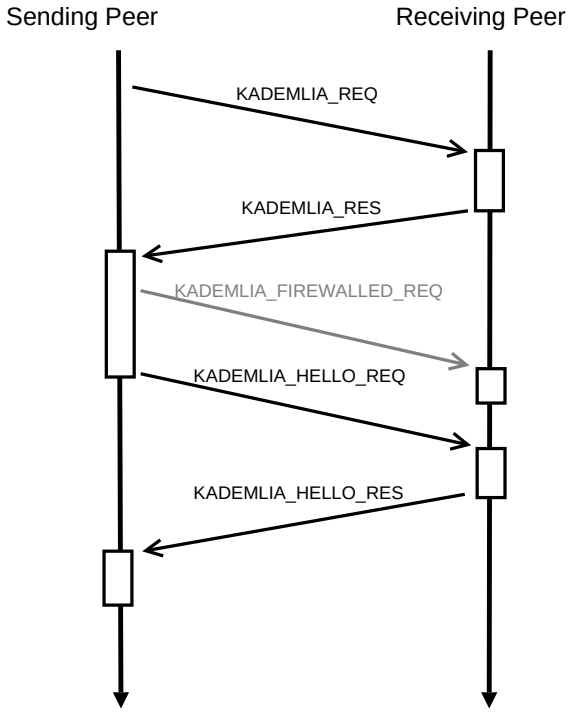

Figure 3.9.: The process of the initial handshake

Moreover, the aMule client checks, if its contacts are still alive every minute. Therefore it sends a KADEMLIA_HELLO_REQ to one contact of each routing table leaf. In detail, it removes first all contacts, which have passed their `expiration` date. Then it takes the contact with the closest expiration date and sends an hello request to him.

3.3.3. Firewall check

The "`firewalled`" status indicates the accessability of the ports and corresponds to the eDonkey status "low id". There are two reasons that a client

can be firewalled; a firewall or a Network Address Translation (NAT). The firewall blocks all incoming messages of peers, which were not connected before by default. When a host is behind a private network and has its IP-address masqueraded, unknown incoming messages can not reach the destination peer [1]. A solution is to forward the concerning ports of the router to the client or to open the corresponding ports in the firewall. Actually there are many articles in the internet, which describe the `firewalled` state and the solution for an open connection. However, Kad has a difference to the *eDonkey*, which isn't so obvious. Even though the Kad client runs behind a router with a NAT it can obtain the status firewalled when the router redirects packets for the outgoing UDP port (default 4672).

The Kad protocol has its own process to find out, whether the clients ports can be accessed directly. This firewall checking process is executed immediately after establishing the connection to the Kad network. After the initial checkup the timer will periodically repeat this process every hour. Therefore the timer sets a variable to notify the client that the firewall checking process is activated. Then there are three possibilities to trigger the process, which are shown in figure 3.10. A request is sent directly after an incoming message and it is included in the normal communication between two peers. This assures that the contacted peer is very likely to still be alive.

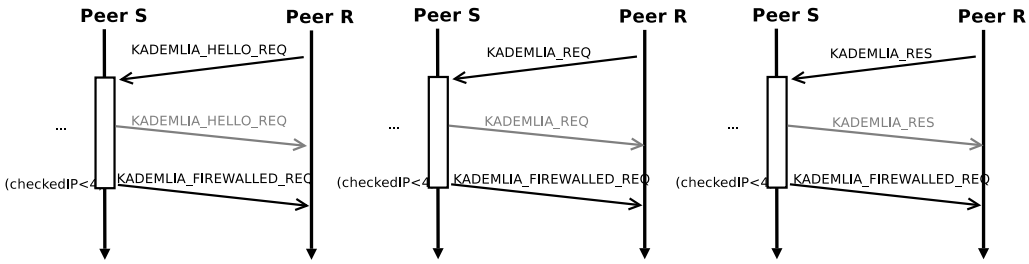

Figure 3.10.: The three triggers for the firewall check

When the firewall checking process is triggered by the sender peer, it sends the KADEMLIA_FIREWALLED_REQ including its TCP port (see figure 3.11). Then the receiving peer puts the incoming IP address into the response package KADEMLIA_FIREWALLED_RES and returns the package. When the checking peer gets the response he verifies his IP address with the one seen by the responding client. It knows by a unequal IP address, that it is behind a NAT. Each incoming response increase also the response counter to inssure that the firewalled request does not use too much bandwidth. So the check will be stopped after four incoming responses. Furthermore the responding client will try to make a connection to the checking client. Obviously, only peers that are not `firewalled` can establish a test connection. If the connection is successful, it will send a KADEMLIA_FIREWALLED_ACK. The checking client indicates not firewalled, when it has received more than two positive responses.

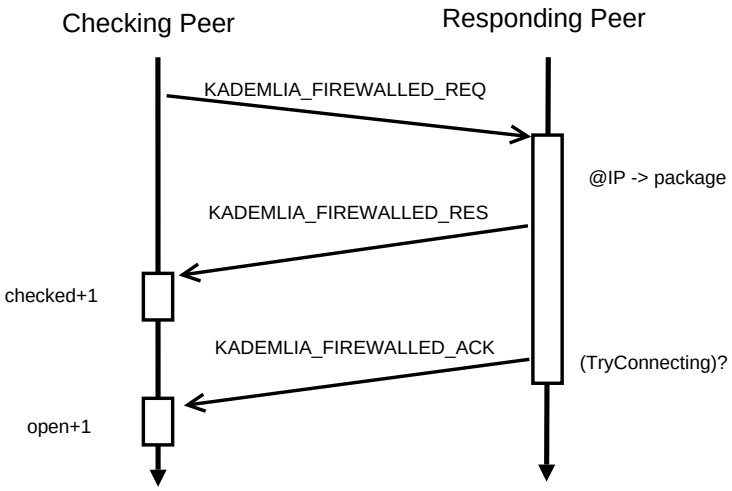

Figure 3.11.: The firewall checking process

3.3.4. Find Buddy

A `firewalled` aMule client has to find a non-firewalled client, called a buddy. A buddy can receive incoming messages for its firewalled buddy though it will manage his communications. Only one buddy is allowed for each firewalled or non-firewalled client. The search for a buddy starts five minutes after the firewalled check and only if the client is still firewalled. The earliest time for a buddy search is five minutes after the connection. This gives time for the client to obtain enough responses for the firewalled check. To start a buddy search, a client sends a "KADEMLIA_FINDBUDDY_REQ" $<$ *serviceport* $>$ to the three nearest peers. If the receiving peer is not firewalled, he will answer with a "KADEMLIA_FINDBUDDY_RES" $<$ *serviceport* $>$ message.

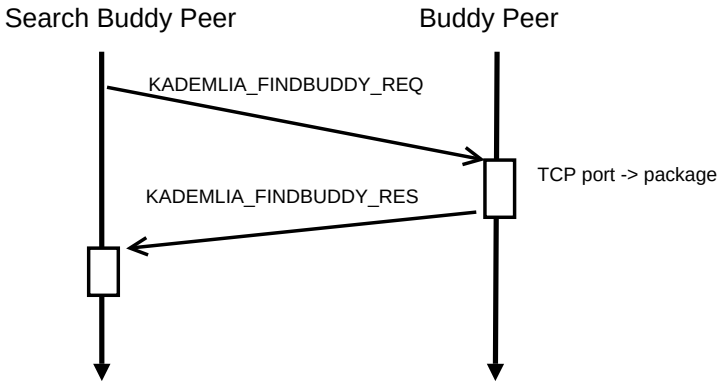

Figure 3.12.: The find buddy process

4. Lookup process

The Kademlia-based protocol Kad has the same lookup processes like the other prefix-matching DHTs: Pastry [51], Tapestry [61] or Toplus [13]. Without knowing all nodes in the network, a peer in the Kad network can find any other active peer. Normally, a peer in a Kad network with 2 million peers, knows only a subset of about 1000 peers. A client chooses these contacts according to a specific position and keeps them in an structured hash table, also called *routing table*. Studies showed that the structured distributed hash tables (DHTs) show a higher performance in locating specific nodes or rare objects than networks with unstructured DHTs.

4.1. The Lookup

The lookup describes the methods and algorithms to find peers, which are close to a certain target in the Kad space. Therefore exists different strategies, which are described in this section.

4.1.1. Object locating

An object in an overlay network can be located with an **iterative** or a **recursive**. These two methods are illustrated in figure 4.1. Each method has its advantages and disadvantages:

1. Recursive routing has the risk, that an intermediate peer with the lookup message is departing the network [7].
2. The recursive lookup has a lower latency than the iterative lookup[58].
3. The iterative routing is easier to implement, because it allows a better debugging and maintenance [55].

In the end the advantages of the iterative methods prevails the disadvantages in the Kademlia protocol. Especially the iteration copes more efficient with the high fluctuations and dynamics of the peers.

Figure 4.2 shows an example of a node location in a simplified 4-bit number space. The point S starts looking for the the node T, which is in another subtree. So he asks an acquainted node from the other subtree for a closer node to the target. Again he demands the recently obtained node for a closer one. Finally, after the iteration through several nodes, the target node will be found by at least $log(n)$ hops, where n are all peers in the network. Each iteration step reduces the metric distance to the target by at least $\frac{1}{2}$ [35]. This corresponds to the limitation of the possible target peers to $\frac{1}{2}$ or even less.

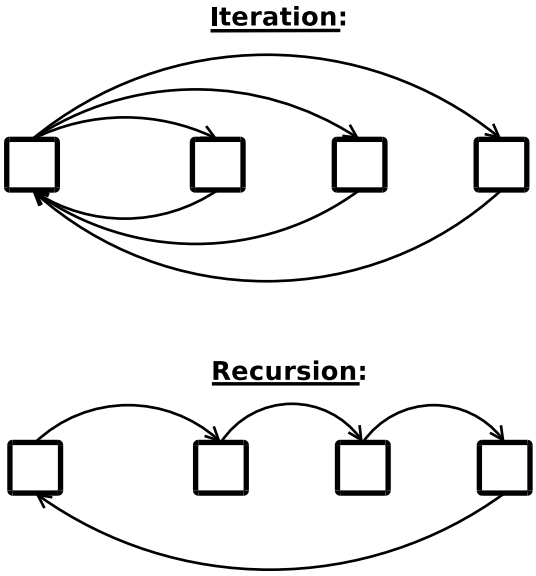

Figure 4.1.: Two different node lookup methods are distinguished: *iteration* and *recursion*. Within the iteration, the initial peer takes the control of the lookup. The other peer in the lookup chain responds only simple requests with some of its closer contacts. In contrary the recursive method sends the address of the initial peer to another. This one redirects independently the message to a closer peer. When the target peer receives the message it will notify the initial peer.

4.1.2. Concurrent lookup

In theory there is one **iterative** lookup process to find a peer, which is described in the previous section. But this has the risk that a single stale client would increase the latency of the whole process. The delay would be the sum of each stale contact in the iteration process. To avoid this, the solution in Kademlia is a concurrent node lookup, also called parallel lookup. That means that a lookup request is sent to α peers at the same time, instead of sending the lookup to only one peer. Now, when one peer in the iteration process is overloaded, there are still other peers which will respond. The same technique is already applied by EpiChord [24] and Accordion [27].

It is important to find an adequate α, because a high number of parallel lookups could accelerate the iteration to the target. But the network overload would increase by the same manner. The Kademlia protocol defines an $\alpha = 3$, which means that there are 3 parallel lookups. Also Stutzbach found out in his study, that the advantage of a faster iteration in relation to the network overload is an α of 3 [55]. Finally, two different lookup methods can be differentiated; a **strict concurrent lookup** and a **loose concurrent lookup**.

Strict concurrent lookup is the original approach of Kademlia [35], which is

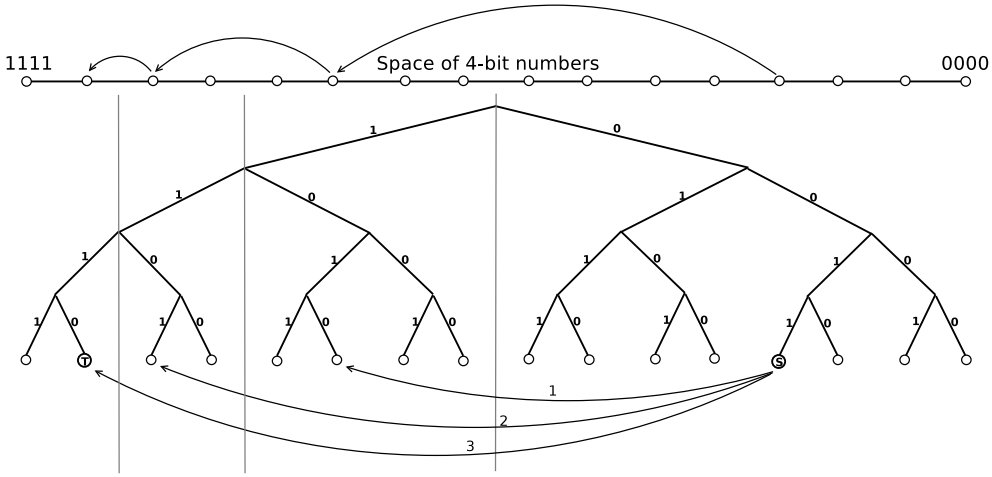

Figure 4.2.: Locating a node in the Kademlia space. The searching peer S with the ID 0011 searches for a target object T with the ID 1110. So the search iterates to nodes which are in a closer subtree.

shown in figure 4.3. There are α parallel lookups, where each process acts like an individual and independent lookup. Without consulting the neighbour processes, each lookup process follows its closest way to the target. This is even the case if another lookup process found closer contacts. When one process finds a stale contact, it must wait until the timeout. Within this method the resulting maximum network overhead is linear to the factor α.

Algorithm 2: Strict concurrent node lookup for one α thread

```
if Incoming response OR last request time out then
    if has a closer node to the target then
        send request for a closer node;
    endif
endif
```

Loose concurrent lookup The Kad algorithm of aMule and eMule implements a from the Kademlia modified and though a looser concurrent lookup explained in figure 4.4. It starts like the strict lookup with $\alpha = 3$ initial requests to the closest **possible** contacts. Possible contacts are known nodes, which are the closest to a target. However, they are only imaginable nodes, because the peer does not know if they are still alive. When a response with several closer contacts arrives, the Kad protocol will sequentially compare these with the already closest ones. By each incoming of a closer contact, the client will immediately send a new request to it. The α assures that there is a maximum of 3 simultaneous request. At the first view, this seems to result in more expensive network traffic. But

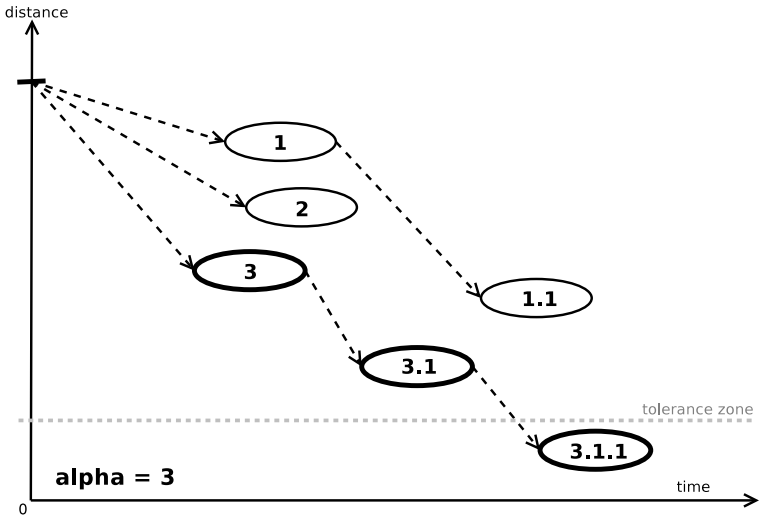

Figure 4.3.: The strict concurrent lookup follows $\alpha = 3$ parallel lookups. This figure shows that thread 2 makes a lookup to node 2. But this one is stale and returns no new contacts. After the timeout the search restarts with the second closest node. If this were only this process it would significantly increase the total lookup latency. Thread 3 is the fastest to reach a node which is in the tolerance zone. After rechecking node 3.1.1 the lookup process is complete and the other threads are abandoned.

when a first response arrives it has already several closer contacts, so that the responses of the other two are no longer relevant.

Algorithm 3: Loose concurrent node lookup

```
possibles ← get 50 contacts from the routing table with minimum distance
to the target;
send to α contacts a request for β closer nodes;
while Incoming response do
    reset timeout;
    for received node < β do
        if Not already tried OR already known then
            if Node is in the α closest contacts to the target then
                send a request for β closer nodes;
            else
                possibles ← node;
            endif
        endif
    endfor
endw
```

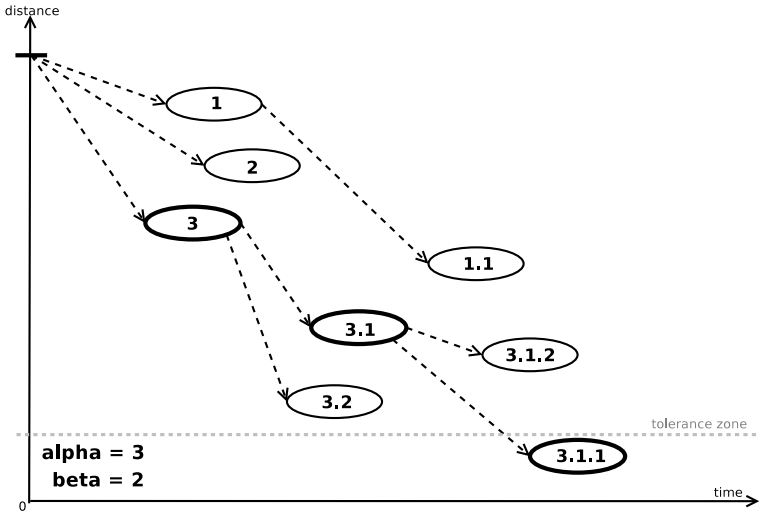

Figure 4.4.: The loose concurrent lookup begins like the strict lookup with $\alpha = 3$ parallel lookups. But within this lookup the contacts try to responds with $\beta = 2$ closer contacts. If the obtained new contacts are among the α closest contacts at that moment, a request will be sent to them.

Generally, if there is a high percentage of stale nodes in the network, the latency of the loose lookup will be significantly smaller. Otherwise there is no notable latency difference between the two concurrent lookup methods [55]. For both methods an $\alpha = 3$ is optimal, which means an efficient and short lookup to find the target without increasing unnecessarily the network overload.

Furthermore an implemented **stale protection** assures that a search does not block, when it meets only stale contacts. At the beginning of a search, a client puts the 50 closest contacts to the *possible list*. During the search process, it still adds new obtained and not already requested contacts to that list. After a timeout of 3 seconds, the peer sends concurrent requests to all contacts in the *possible possible*.

4.1.3. Object ID

The Kademlia space consits of peers and data objects. In accordance the Kad network has the following objects; **clientIDs** of peers and the hash values of **sources**, **keywords** and **notes**.

ClientID Each peer calculates randomly its own unique hash ID. This is neither influenced by its IP address nor by its port number, because the uniqueness of the IP address and port number can not be assured. For example two clients behind a router could have the same IP address and the same port number. After the initialisation the client serialised its clientID. This

is loaded by each starting of the client. So all clients will keep its ID even when they change their IP address.

Source Each file has *location information*, which is published instead of itself. Therefore the first bytes of a file calculates their corresponding hash value. So even after changing the file name, the hash value will remain the same. A popular file, which exists several times on different peers will have the same sourceID.

Keyword is to allow a "human" search for words and to profit of other advantages (compare 5). Each filename is split to several keywords. For example a file named "Kademlia Project" will be split into two case insensitive keywords "kademlia" and "project". Then these will be published in the Kad-space.

Note is a comment and a rating for a file. So it will also be identified with the sourceID like the source. To avoid interferences with sources, another opcode guarantees the strict separation of these two references.

The hash IDs are created by the cryptographic hash function MD4 [50]. The result is a 128-bit hash, which has the same bit size like all Kad-IDs. This method avoids a collision between some encrypted letter sequences. That means if there are more different words, their hash values will consequently be also different. Moreover, it assures that it is impossible to find the real word, knowing only the hash value. Only a single twist in the letter generates a totally different hash value. An Kad-ID, presented in the hexadecimal format, could look like these:

Kademlia C90A12567F3F56870C79889EAF6CA47F

Kadmelia 13941B5DAC38B4966aB8200B1C409CC5

4.2. The Search object

The search is responsible for finding the different objects in the Kad space. This is for example, the classic search for a Keyword after a user's input. But the client also executes internal searches for sources from the *partfiles* or for other peers to maintain the routing table. Furthermore the search object manages the converging to a node, which should at least be situated in the minimum distance of the target. So the Kademlia iteration process is defined in the search object. This process is driven by a timer, which determines the lifetime and controlls the state of the search periodically. The other controlling instances are the responses from other peers who are answering to the searches.

Search Type	Timeout (seconds)	Maximal Responses
FILE	45	300
KEYWORD	45	300
NOTES	45	50
NODE	45	-
NODECOMP	10	10
STOREFILE	140	10
STOREKEYWORD	140	10
STORENOTES	100	10
FINDBUDDY	100	10
FINDSOURCE	45	20

Table 4.1.: The different search type with parameters

4.2.1. Search types

There can be searches with 10 different target types, which are shown in figure 4.1. First of all these are the standart searches for a "file", "keyword" or a "note". But there are also implementations for an internal behaviour like the search for a "node", which is used by the routing table to find a random node. A "nodecomplete" is needed to execute a selflook-up, which reapeats periodically every four hours. This is also a part of the bootstrap mechanism described by [35], because it searches close nodes around the peer itself. The three "store" types have the objective to publish a file, a keyword or a note. For the buddy support the "findbuddy" is responsible as well as the "findsource" is responsible for a callback request.

4.2.2. Search states

Figure 4.5 describes the different states in which a search passes through during its lifetime. At the beginning, an empty search object is created. After that the hash value for the destination target is assigned to the search with one of the ten search types. Though the object obtains the state "started". Then the routing table selects 50 of its, to the target closest contacts. These are called *possible* contacts, because it is not known at this stage if they are still active. Maybe it is a lot to start with 50 possible contacts, but this should avoid the search from being stuck as a result of stale contacts. Anyway the maximum of concurrently sent requests is limited to $\alpha = 3$.

By entering the "tried" state a KADEMLIA_REQ message is send to α closest possible contacts. The search will rest at this state until at least one response (KADEMLIA_RES) arrives and changes the object to the state "responded". When a response package contains new contacts, which are neither among the list of possible contacts nor are they already tried, they will be added to the list of possible contacts. If a new contact has a smaller distance than one of the contacts from the "best list", it will take the peer with the longest distance.

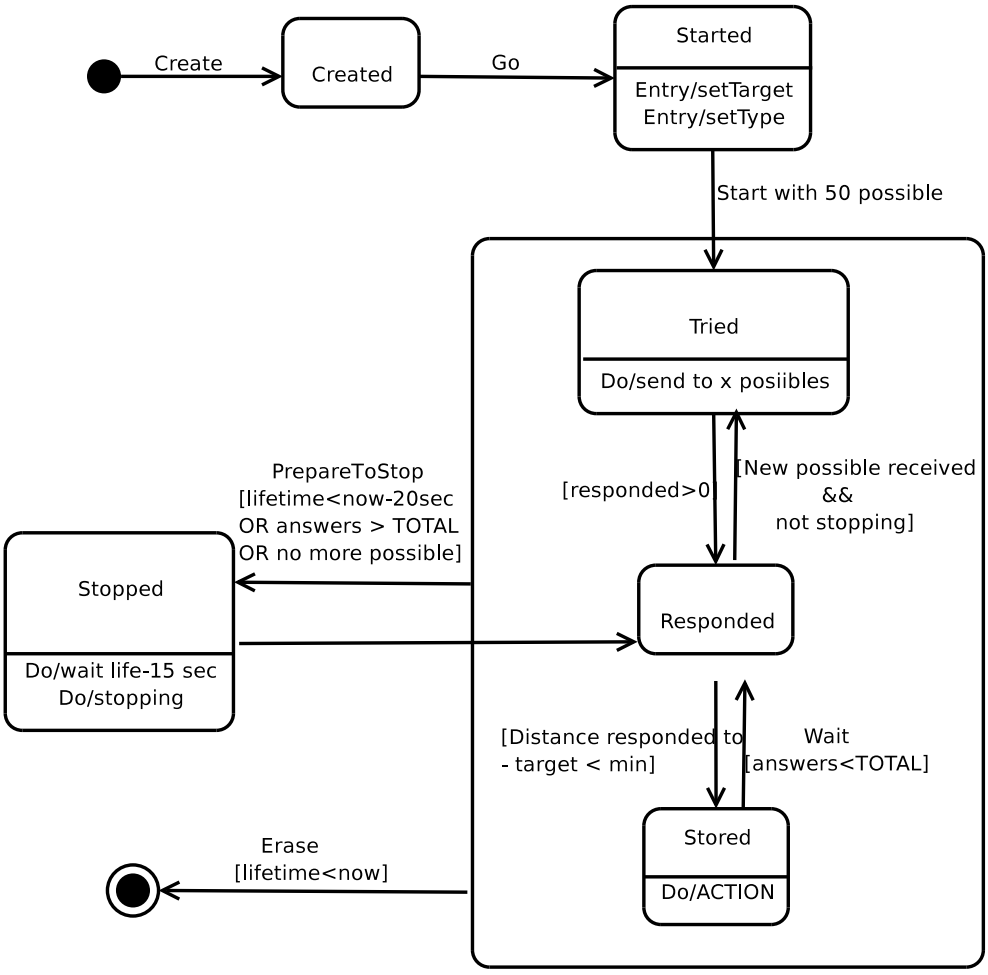

Figure 4.5.: The UML state machine diagram shows the different states of a search object.

The size of the "best list" is defined by ALPHA_QUERY and is three. A request will be send directly to a contact when he belongs to the best list. Otherwise it will be added to the possible list. If it is the closest contact in this list, it will be send a request within the next trigger of the timer.

A timer checks periodically all responded clients in regard to the XOR-distance between them and the target. If the distance is smaller than the predefined search *tolerance*, the search will pass to the state "stored". This will activate a specific action corresponding the search type listed in table 4.1. This is either sending a demand to store a data item or a request for an object. After lancing such an action and if the maximum of 10 allowed answers is not already reached, the search will return to the state "responded".

The "stopped" state can be reached from the "tried", "responded" or "stored" state. This occurs after one of the following cases. The first is that the lifetime, which is calculated by the `creation time` plus the search `lifetime` from table

4.1, minus 20 seconds has passed. Another possibility to change the state to "stopped" is that the maximum answers (compare table 4.1) are exceeded. The answers are the responses to the specific search type executed within the "stored" state and not the normal responses. At least the stop will also be prepared when there are no more possible contacts. To assure that a lot of delayed returning packets are not missed, an action increases the lifetime by 15 seconds. It also activates the stopping variable, which ensures that no more KADEMLIA_REQ message are sent. Then the search will return to the state "responded", where it can continue with the store process. Finally, the search will be erased when the stopping is activated and the lifetime is over.

4.2.3. The iteration process

The Kademlia DHT follows its own algorithm to find an object like it is explained in the sections 4.1.1 and 4.1.2. This section proceeds with the detailed description of the approaching to a target object. Therefore the following figures 4.6, 4.7 and 4.8 illustrate the iteration process with a particular example.

Figure 4.6 illustrates a 128-bit number space. In the example the searching peer is market with its possible contacts. These are the three closest possible contacts from the routing table. They have a different XOR-distance and are still not close enough to the target. The tolerance zone around the target indicates when a contact is close enough to host a reference.

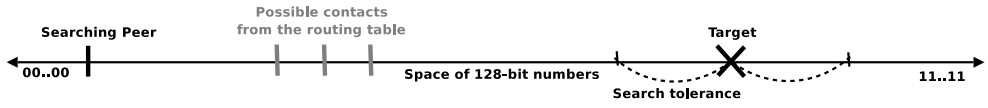

Figure 4.6.: Extraction of a Kademlia space with 128-bit numbers. It shows the searching peer with three of its *closest possible contacts*. Around the target object, the tolerance zone marks when a contact is close enough.

When a peer wants to approach a target, it sends a request to 3 contacts out of a selection with 50 possible contacts. For an easier comprehension, figure 4.7 displays only three possible contacts. If they are still available, they will respond with a certain quantity of new possible contacts. The quantity of replied contacts depends on the search type, which is listed in table 4.2. Here the request have the *type* 2. These possible contacts are known by the responding node and have the smallest distance to the target. As each peer looks extensively for its neighbour peers, a demanded contact knows closer contacts. In case that a peer is the closest node to the target, it will reply with the closest peer of its routing table. These will be further than itself.

The second step in figure 4.7 shows, that the first requests responded with four new possible contacts. Two of them are already situated in the tolerance zone of the target. These nodes will stay in the list with the possible contacts,

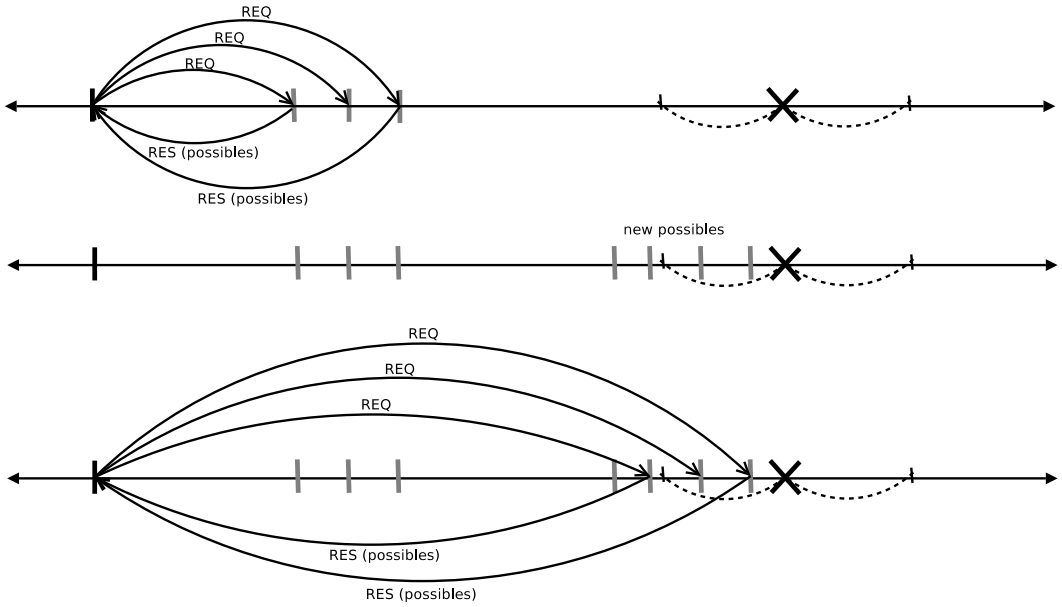

Figure 4.7.: Shows three steps of the approach to the target object. First, the searching peer sends three messages to the closest possibles contacts. As response the searching peer obtains four closer contacts. In this example two contacts are in the tolerance zone of the target. In the last step the searching peer sends a request for closer contacts to the three closest contacts again. But only two peers are available and reply with more closer contacts.

because the client doesn't know yet, if these nodes are still connected. So in the next step it will execute another request to all new obtained possible contacts. Again, they will respond with a list of possible contacts.

Finally, when a contact has found some connected contacts, which are in the tolerance zone of the target, it will send them the **action** request. An action is executing the real search or publish request depending on the search type listed in figure 4.1. They will reply with the type specific response. The counter of the answers will increase by one for each response of an *action* request. So if the counter reaches the maximum, the search for new nodes will be stopped. Otherwise the search will continue like it is shown in figure 4.8.

4.2.4. Kademlia request

The Kademlia request KADEMLIA_REQ $< TYPE : targetID : receiversclientID >$ is strictly for the iteration process and its only purpose is to find closer contacts. It is important to not confuse it with the Kademlia search request, which asks a peer in the tolerance zone for a certain object.

Each Kademlia request message contains a parameter specifying the category of the search, which are shown in table 4.2. There are three different categories

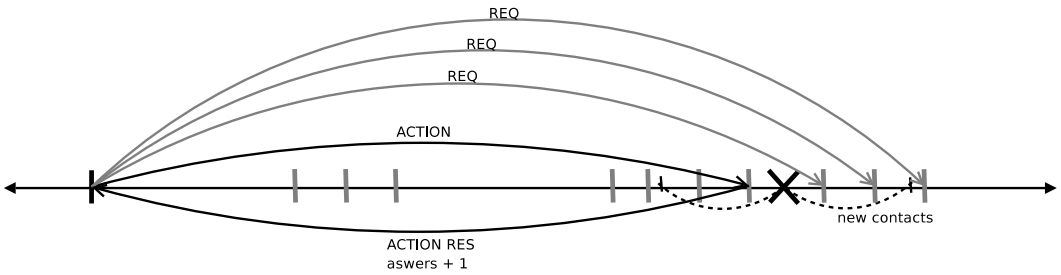

Figure 4.8.: The real request for an search type. This happens when a active contact is found, which is in the tolerance zone of the target.

Protocol	Parameter	Responses	Search type
KADEMLIA REQ [0x20]	FIND NODE [0x0B]	11	NODE NODECOMPLETE
	FIND VALUE [0x02]	2	FILE KEYWORD FINDSOURCE NOTES
	STORE [0x04]	4	FINDBUDDY STOREFILE STOREKEYWORD STORENOTES

Table 4.2.: The Kademlia request possibilities

representing the quantity of responded contacts. For example a response of the category `FIND_NODE` replies 11 closer contacts from the routing table of the responding peer. If the category identifier is higher than 7, the message will be sent within a compressed packet.

In detail this process looks like it is shown in figure 4.9. A peer wants to find another peer which is closer to the target than to itself. So it will send the hash value of the target and the request type to the possible closest peers. The possible peers are at the beginning 50 peers from the routing table, which have the smallest XOR-distance to the target and have a *contact type* smaller than four. There are a lot of possible contacts, but this should be a fallback in case that the search stales due to dead contacts. The requested new peers, which have to be even closer to the target will be added. If one of the three nearest peers receives the message, it will look for its closest nodes to the target of the type smaller than three. The quantity of the nodes which are returned is defined by the parameter of the request. With the response, the client will add the received unknown nodes to his routing table. If the request was of the parameter FIND_NODE the search response wont be processed any further after adding them to the routing table. If it was of another parameter it will add the unknown obtained nodes to the possible peer list for this search. In the

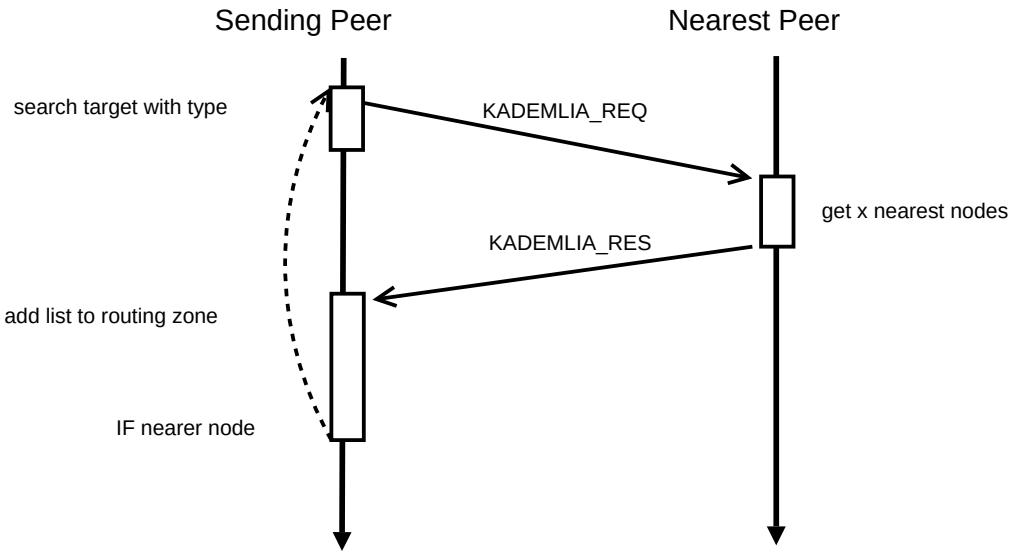

Figure 4.9.: The Kademlia request

next step the client executes the same process to the new possible contacts.

5. File sharing

To assure the filesharing of the P2P applications, the Kad protocol needs a process to publish the identifier of files in the 128-bit number space. This allows a relative rapid lookup for files starting at any peer in the Kademlia network. Only a subset of peers in the network will be responsible for a certain reference. When a peer is in the *tolerance zone* of the metadata, it is allowed to take the responsibility of the reference. Then the peer is called **host peer**. This means that it returns the metadata to other peers, which are searching for that keyword. The host peer is only responsible for the reference for a certain time. This should avoid unnecessary reference at a non-existing file in the network. Furthermore the host peers of the references handle the **overload protection** for popular keywords.

As the publishing of entire files or chunks would cause too much network traffic, the Kad protocol publishes only *references*. Files are efficiently shared by publishing two different references, which has dependencies on each other. The next section describes this system, which is called **2-levels publishing**.

5.1. 2-level publishing scheme

A file in a filesharing application is static and stays at the publishing peer until a download process starts. Each peer takes charge of the publishing of its own released files. The 2-levels publishing scheme divides the files into two reference types: **metadata** and **location information**. While keeping the real files by the releasing peer, only file location information will be sent to other peers. These are on the *1st level* and points to the peer with the real file. A metadata contains information like ID3 tags for MP3 files and is identified by a MD4 hash of a *keyword*. Files have the same number of metadata pointing to their location information, as its filename has valid *keywords*. The metadata is distributed on the *2nd level*.

Figure 5.1 shows a simplified example for the publishing process of the Kademlia protocol. A peer wants to publish the file with the name "Kademlia Project". All relevant reference to the real file are generated in the first step. This contains a unique hash for the location information *source*. The sourceID is computed of the file's bits. Thus the same files can have different metadata tags like for example two different titles. This results in different ripping of parameters of individual users [28]. Thereafter the *keyword* is extracted from each word of the filename and the corresponding hash value is calculated. Now each reference has a unique hash value to distribute it to the Kad-network. The following distribution of the keywords and sources are exemplified in detail in the sections

5.2.1 and 5.2.2. The main intention of this figure is to illustrate the different distribution of pointers and references.

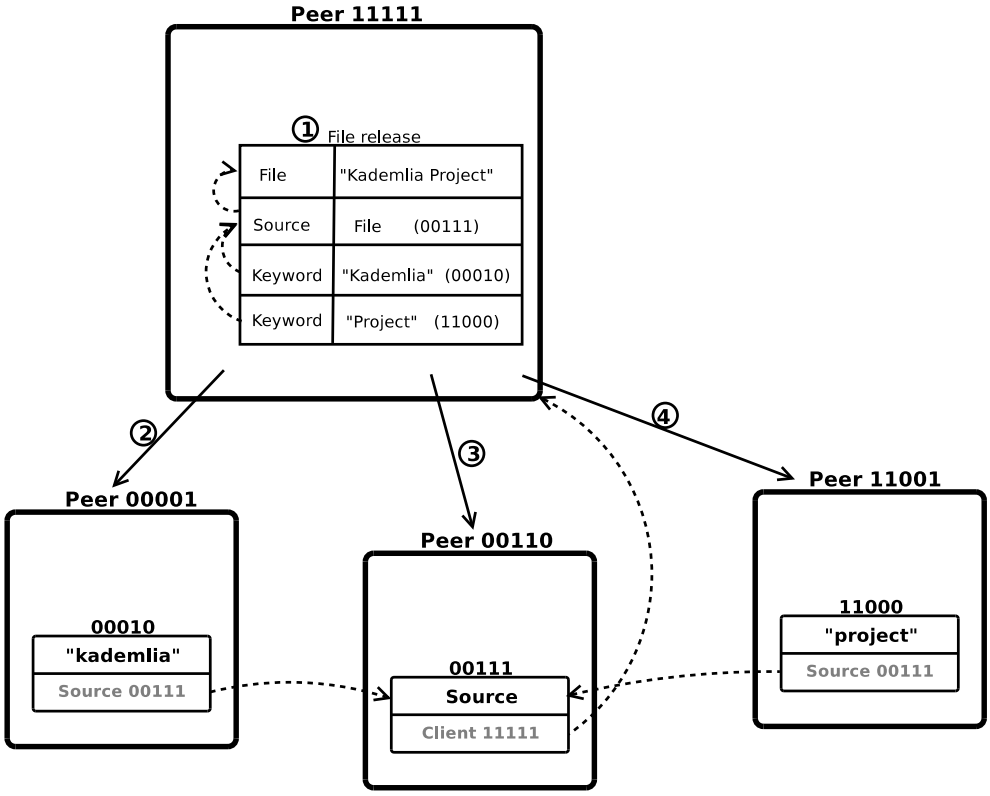

Figure 5.1.: Presents an example for the publishing of a file with simplified hash values. The figure shows a peer generating the references for the file "Kademlia project" in the first step. Then it begins to distribute the keyword reference "kademlia" to a peer with a small XOR distance. After that the location information, is published with a pointer to the peer containing the file. Step four distributes the other keyword "project" to one of its closest peers. Both keyword references contains a pointer to the same source reference.

The 2-level scheme in figure 5.2 has advantages against the 1-level publishing scheme in figure 5.3. Especially in networks with a high rate of file duplication needs the 2-level scheme less references. Also benefits the average keywords per file of 7 the 2-level publishing scheme. But also the separation of the keyword and source balances the network overload on different peers.

A further step would be the transferring of the keyword responsibility to the 1st level. Therefore it is necessary to introduce a ping so that the peer on the 1st level knows if the file is still available. This is described in another study [12].

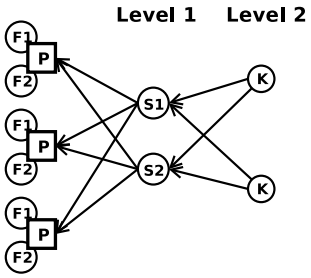

Figure 5.2.: This example of a 2-level publishing scheme shows three peers, which have all the same two files *f1* and *f2*. This scheme needs 10 references for this scenario: $distinctfiles * replicationperfile + distinctfiles * keywordsperfile = 2 * 3 + 2 * 2 = 10$.

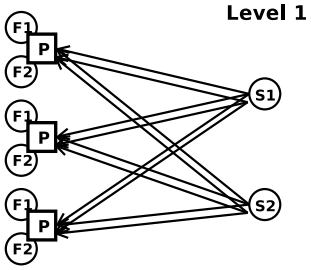

Figure 5.3.: This example of a 1-level publishing scheme shows three peers, which have all the same two files *f1* and *f2*. This scheme needs 12 references for this scenario: $distinctfiles * replicationperfile * keywordsperfile = 3 * 2 * 2 = 12$.

5.2. Object publishing

As already previous sections described, the Kad-protocol differentiate between three object types, which can be published; a metadata, a location information and a note. These objects must be situated in a tolerance zone like previous sections also described. Now, this section explains the publishing process in detail.

All references are distributed several times, which is called **content replication**. This is necessary, because of the churn of the peers. Publishing a reference to a single peer has a high risk. The reference would be lost, when afterwards the peer disappears or is stale. So the Kad-protocol sends the same reference to at least 10 different peers. An optimal value for the content replication will be analysed in the chapter measurements.

5.2.1. Metadata publishing

The metadata publishing is on the 2nd level of the publishing scheme. So it references to the 1st level with the location information. But it already contains the information about the real file like table 5.1 shows. These details make the source selection for a user easier. The publishing process starts right after the Kad-status "firewalled" transferred to "connected". Otherwise the process starts, when the client has found a buddy to handle incoming messages. Of course, the client must have a file to release to start the publishing process.

Name	Type	Description
`File ID`	128-bit number	A Hash ID of the file
`Client ID`	128-bit number	Hash ID of the publishing client
`FILENAME`	text	The whole name of the file
`FILESIZE`	number	The byte size of the file
`FILETYPE`	number	The type of the file (audio, video, ...)
`FILEFORMAT`	text	Format of the file (mp3, avi, ...)
`MEDIA_ARTIST`	text	Artist of a media file
`MEDIA_ALBUM`	number	Album name of a media file
`MEDIA_TITLE`	text	Title of a media file
`MEDIA_LENGTH`	text	Length of a media file
`MEDIA_BITRATE`	number	Bitrate of a media file
`MEDIA_CODEC`	number	Codec to read the media file
`SOURCES`	number	Indicates the availability

Table 5.1.: List of the tags for the metadata

If one of the keys has never been published or the republishing time of 24 hours has passed by since the last publishing, a new publishing process will be started. The started publishing process blocks the distribution of the other references because the `KADEMLIATOTALSTOREKEY` allows only one simultaneous metadata publishing. When the metadata is sufficiently published according to the content replication, it will also block the next publishing for 2 seconds.

The keywords are distributed metadata, which reference sources. So pointers to sources are packed into the keyword publishing message. A client will publish maximal 150 sourceIDs for a keyword. If a keyword references more sources, the client will rotate the list to publish different source IDs the next time. To avoid large messages, the publish request for a keyword is divided in parts with maximal 50 sourceIDs. Figure 5.4 shows the maximum of three possible KADEMLIA_PUBLISH_REQ messages in the format $<$ `Keyword_hash` : $n*$ $<$ `source_hash` : $m*$ $<$ `metatag` $>>>$. All messages are sent autonomously to a peer in the *tolerance* zone, found with the iteration process.

Each incoming message to the receiving peer will be treated separately. First it will check if the maximum of 60000 keywords is already exceeded. If this is the case, it will respond with a KADEMLIA_PUBLISH_RES $<$ `Keyword_hash:load` $>$

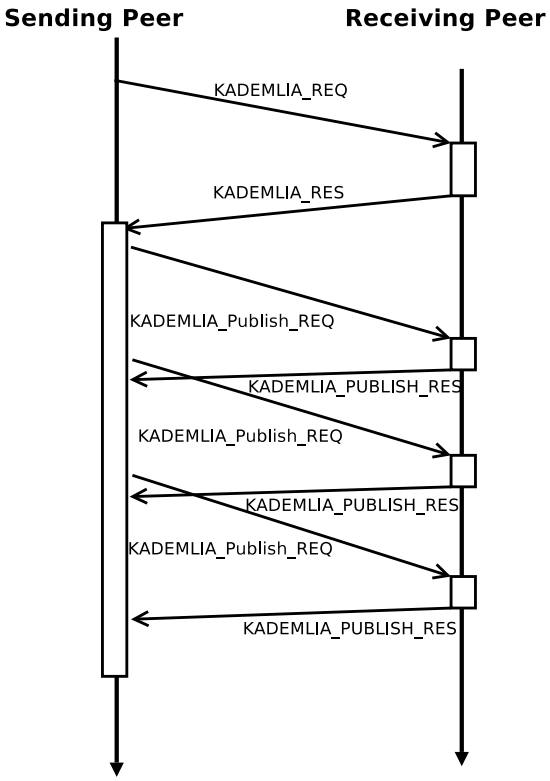

Figure 5.4.: The Kademlia publish process for a keyword

including the published keyword ID hash and a load of 100. It is important to mention that a positive response is sent anyway. In order to avoid a "hot node" from indexing only popular files, when a new source for a keyword is added, the response message will include a load, which is described in detail in section 5.3.

5.2.2. Source publishing

Sources are the location information pointing directly to the peer, which holds the real file. For each file, exactly one location information is published. However, the content replication induces the publishing on several different peers. The publishing process is similar to the metadata distribution. But the maximum number of synchronous source publishing processes is two. Again, the first publishing of a source starts when the client obtains the status "connected" or when a buddy is found. The republishing starts periodically when the standard republishing delay of 5 hours has passed by.

A source entry has the attributes listed in table 5.2. These are all included in the publishing message. This is of course the sourceID of the file and the contact information about the peer releasing that file. Wheather the buddy information is published or not, is defined by the `SOURCETYPE` described in

table 5.3. `HighID sources` are non-firewalled peers and they send nothing for the buddy information. Sources of type 3 are published from *firewalled* peers and though they contain the information about the buddy. So a peer, that wants to download a file of type 3 must first contact the buddy. This one will inform the peer releasing the file. Now, the peer with the file contacts the peer which wants to download and its firewall is passed.

Name	Type	Description
`SOURCE ID`	128-bit number	A Hash ID of the file
`CLIENT ID`	128-bit number	Hash ID of the publishing client
`SOURCETYPE`	number	Type of the source (firewalled, ...)
`SOURCEIP`	number	IP address of the publishing peer
`SOURCEPORT`	number	Service port number of the publishing peer
`SOURCEUPORT`	text	Message port number of the publishing peer
`SERVERIP`	text	The IP address of the buddy
`SERVERPORT`	number	The port number of the buddy
`CLIENTLOWID`	number	Clients Hash ID when it is firewalled
`BUDDYHASH`	number	Hash ID of the buddy

Table 5.2.: Properties list for sources

Source type	Description
1	highID sources
2	used by older clients [depreciated]
3	firewalled Kad source

Table 5.3.: A source can have three different types, which influence the distribution

When a source is supposed to be published, the peer will start a Kademlia request lookup (see section 4.2.4). This will find a peer in the tolerance zone of the sourceID. Then the publishing peer will send the KADEMLIA_PUBLISH_REQ message $<$ `sourceID` : $n*$ $<$ `clientID` : $m*$ $<$ `sourcetag` $>>>$. If the client is firewalled accordingly it will also send the data of his buddy.

As a peer receives a publish request for a source, it will also record its information like the incoming IP address. When a peer receives a new source it will add the location information to its sources list and responds with a KADEMLIA_PUBLISH_RES message $<$ `sourceID:load` $>$ including a load of 1. Otherwise it will add the source of the file (Client ID, client IP address, port number, etc.) to the existing source. In this case it will respond with a load, which is described and calculated in detail in section 5.3.

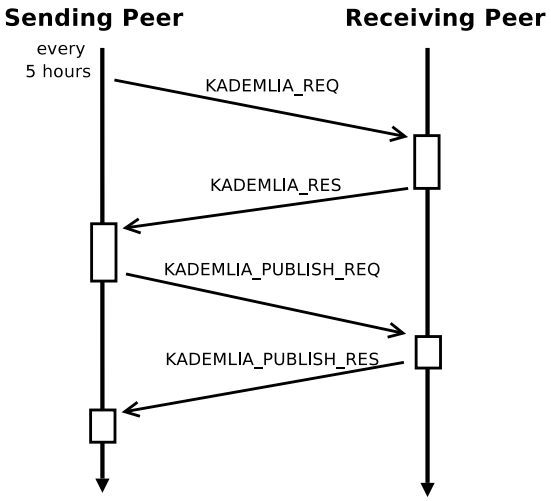

Figure 5.5.: The Kademlia publish process for a source

5.2.3. Publish note

The study of [28] shows, that in 2004 50% to 80% of copies of popular files have been found to be polluted. When somebody wants to prevent the sharing of a certain file, he will start a *pollution attack*. Therefore he corrupts a file to make it unusable or to change its content [9, 41]. The other users continue to download the file in belief that the file is correct [29]. When the download is finished they will note the bogus. But in the meantime they have already offered chunks of that file to other users, who will offer it again to other users. So the note with a comment and a rating for a file informs the users about the content at the beginning of the download. So he can stop publishing the corrupted file.

But the *note* can also be used for all other kinds of information storing. The noteID is in the Kad-protocol the ID of the file to which the note belongs. But the noteID can be the MD4 hash value of any other identifier or key. Then information can be stored on peers in the Kad-network and later on they can be retrieved with the key. This allows an other frame to use the infrastructure of the already existing Kad-network.

As the principal objective of the note is to evaluate the contents in the network, the `FILERATING` is evaluate a file with a rating between 1 and 5. The rating 5 is the best and 1 is the worst. The rating should only notify when a file is corrupt or evaluate the technical quality like sound or video. But sometimes the user also validates the content of the file and not the file itself. No rating is done when it is 0, then the file is only commented. The `DESCRIPTION` is an optional comment for a file. Here a user can indicate that a file does not have the indicated content, when the file is a victim of the pollution attack.It is important to indicate the `FILENAME` to which a note corresponds, because a file can have different filenames in the Kad-network.

Name	Type	Description
`File ID`	128-bit number	A Hash ID of the file
`Client ID`	128-bit number	Hash ID of the publishing client
`SOURCEIP`	number	The IP address of the publishing peer
`SOURCEPORT`	number	The port number of the publishing peer
`FILENAME`	text	The name of the commented file
`DESCRIPTION`	text	is a comment about the content quality
`FILERATING`	number	signifies the quality of a file
		in a scale between 1 and 5

Table 5.4.: Properties list for file notes

It has to be assumed that a user has created a note for a file. A user can only create a note for his *partfiles* or files which are released. Then when the client is either connected or has a buddy, the note will be published to other peers with the same content replication as the keyword and source. The publishing process is similar to the source publishing, but it uses a separated opcode shown in figure 5.6. First the publishing client sends a KADEMLIA_PUB_NOTES_REQ message $<$ `noteID` : $n*$ $<$ `clientID` : $m*$ $<$ `metatag` $>>>$. The nodeID is the fileID of the corresponding file and the clientID is from the sending client. The other attributes like the comment or the rating are in a metatag list.

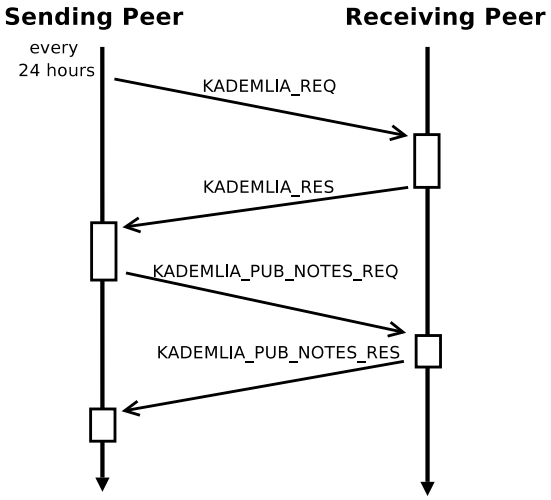

Figure 5.6.: The Kademlia publish process for a note

The receiving client adds the note to a sourceID when the maximum of 50 notes per file is not exceeded. Then it calculates the load, explained in section 5.3. In any case it respond with a KADEMLIA_PUB_NOTES_RES message $<$ `noteID:load` $>$.

5.3. Overload protection

The closest peers to a popular keyword receive more keyword or source references like peers with a higher XOR-distance. This is prooven in the chapter measurement. Consequently, when they are the closest node to popular references, they will receive nearly all references. This causes a high network overload for these peers. Furthermore popular keywords displace unknown keywords without a protection. So the overload protection controls the equal distribution of very common references.

5.3.1. Reference limitation

Each client has a certain limit of network traffic. As the access to the Kad-network is not constrained, it has to be assumed that some clients have slow internet connections or a less performance computer. To assure that slow clients do not disturb the processes, a client can only be responsible for maximal 60000 **metadata** references. Actually, the keyword variety of files is not proportionally distributed. Mostly a few **popular** keywords like "mp3" represents the most files in P2P networks [12]. When a client is a **hot spot** for a popular keyword, the 60000 references are nearly used only for that popular keyword. So *rare* and *unknown* metadata would be lost.

Algorithm 4: Adding incoming keywords

```
Input: Metadata identified by a keyword
if total indexed keywords ≤ 60000 then
    if Keyword has already sources then
        if Keyword has already more than 50000 sources then
            return a load of 100;
        else
            if SourceID does not exist OR total sources for this keyword ≤
            45000 then
                add source to Keyword;
                total of indexed keywords + 1;
                return load = (sources per Keyword*100)/50000;
            else
                return a load of 100;
            endif
        endif
    else
        add source to Keyword;
        total of indexed keywords + 1;
        return a load of 1;
    endif
else
    return a load of 100;
endif
```

To avoid this loss, algorithm 4 shows the overload protection implemented by the Kad-protocol. Each client can hold a maximum of 50000 references to sources for an individual keyword. Furthermore, if a client already has a keyword with 45000 sources, it will stop to publish popular sources. By reaching that limit, only references to totally unknown sources are published. However, when a client has 60000 keyword entries, even unknown references are drawn.

An important protection to prevent the mass distribution of popular keyword is carried out by the publishing response. For example when a very busy node receives a publishing request, but it is already responsible for the maximum of keywords. The peer responds that it published the keyword successfully, even if he had reached his maximum and rejected the request. This avoids the publishing peer spreading the popular keyword to all sourrounding peers. This would unbalance the published keyword, when it is stored on too many peers. Periodically, a peer cleans its references, than it will store incoming references again.

The overload protection for the sources follows a different objective. Here it is important to provide as many download clients as possible, because they are the bottleneck of a filesharing application. The algorithm 5 shows that the inserting of source references is also constrained to a maximum of 300 sources per file. But in contrast to the keyword reference the source inserting follows a *rotation* principle. When an incoming source has exceeded the maximum sources per sourceID, it will replace the oldest source references.

Algorithm 5: Adding incoming location information

```
Input: Incoming sources publish request identified by the sourceID
if Peer has already some sources then
    if a source with the same sourceID exists already from same peer then
        overwrite existing source;
        return load = (sources per sourceID*100)/300;
    else
        if Source has already 300 location information then
            remove oldest location information;
            add new source;
            return a load of 100;
        else
            add source;
            total of indexed sources + 1;
            return load = (sources per sourceID*100)/300;
        endif
    endif
else
    add sources;
    total of indexed sources +1;
    return a load of 1;
endif
```

The insert of a note reference and the calculation of it's load, described in algorithm 5, is similar to a source inserting. However, notes are less commonly published than sources. The noteID is the identifier for a node. The Kad-protocol concepts a note as a valuation for the shared files. So a note has the same identifier like the valuated file. If a user wants to change its valuation for a certain file, the old note will be overwritten. Even if the total load is already reached the note actualisation is assured.

Algorithm 6: Adding a note reference with the load calculation

```
Input: incoming note publish request identified by the noteID
if peer has already existing note references then
    if note with a identical noteID and < IPaddress : port > then
        overwrite the existing note with the new note;
        return load = (notes per noteID*100)/50;
    else
        if NoteID has already 50 notes then
            remove oldest note;
            add the new note;
            return a load of 100;
        else
            add the note;
            total notes + 1;
            return load = (notes per noteID*100)/50;
        endif
    endif
else
    add note;
    total notes +1;
    return a load of 1;
endif
```

5.3.2. Republishing delay

The Kad-protocol implements another method to reduce the network traffic. This is also based on the load, calculated by the host peer of the reference. As the previous algorithms showed, a load exist between 1 and 100 for the *metadata*, *location information* and *note* reference. A peer publishing a reference always receives a load value from the host peer. According to this value, the client calculates a delay or waiting time, before it starts the republishing for this reference. At the moment aMule has only developed this process for the publishing of a keyword. Transferring this method to the source and note publishing would reduce even more the network traffic. But this influences the availability of the sources, which are still a bottleneck.

After each termination of a publish process, a clients computes the delay before the next publishing of the same reference. Therefore it calculates an *average load* of all loads returned by the host peers. The quantity of the responding

host peer depends on the *content replication*. When the average load is under 20, the delay keeps the standard value of 24 hours. Otherwise it continues with the following calculation:

$$\textbf{republishing delay} = 7days * \frac{average \quad load}{100}$$

For example when a peer receives an average load of 30 for a published keyword, the responding clients have on average 15000 sources per keyword. The published keyword is common and the peer reduces the network traffic by delaying the next publishing. In this case an average load of 30, results in a delay of 50 hours.

Instead of only increasing the republishing delay it is also possible to reduce the content replication. This could reduce the network overflow in the same manner, but it keeps the availability more efficient. This avoids the not-available effect of the republishing delay, which is; $calculated \quad delay - standard \quad delay$. In extreme cases, a file is not referenced by a keyword for 6 days.

5.4. Object retrieving

All published objects contain data, which has to be retrieved. This can be induced by the user, searching for a keyword or by the application, searching for sources or notes.

5.4.1. Keyword search

To start a search for a file in the Kad-network, a user needs at least one *keyword* of the filename for which he is looking for. A keyword must consist of at least three letters. A client searches only for the first keyword in the Kad network. However, more keywords are packed in form of a *search tree* into the request message. So the requested peer can filter its keyword entries by these keywords.

To start a search, the client will first calculate a 128-bit hash value for the searching keyword. Then the peer will continue to search nodes which have a hash ID close to the hash of the keyword. Therefore the client utilises the search iteration approach described in section 4. When it knows a node in the tolerance zone, it will send a KADEMLIA_REQ shown in figure 5.7. This checks if the node is still online, but it will also return further close nodes, they can even be closer to the target. When a node responds with a list of contacts, the client will send him the real search request KADEMLIA_SEARCH_REQ in the format; $< keywordID : extention* < SEARCH \quad TREE >>$. The *extention* indicates with the values 1 or 0 if there are more keywords, packed in a *search tree*. But furthermore the whole search parameters such as type, extension, minimum, maximum and availability are packed in the search tree.

If the client has some entries, he will send them back with the KADEMLIA_SEARCH_RES $< keywordID : n* < sourceID : m* < METATAGS >>>$. Here, n defines the quantity of send sources and m counts the information tags per source. As

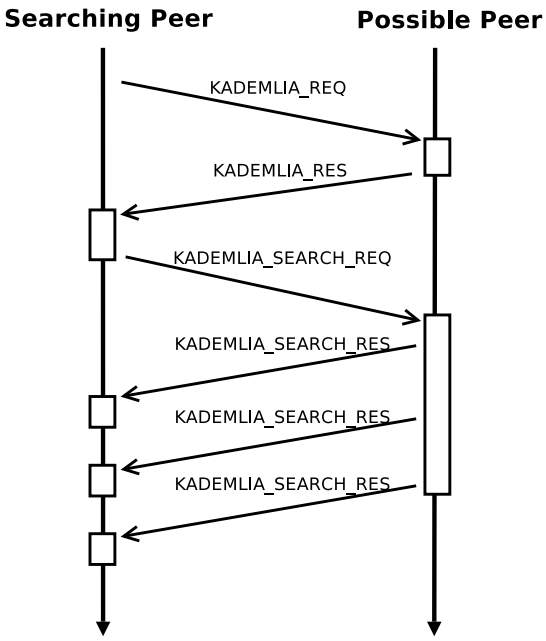

Figure 5.7.: Shows a search for a keyword or a source. The searching peer checks first if the possible peer is connected. Then the real search request is sent with optionally search parameters. The KADEMLIA_SEARCH_RES includes at most 50 entries and though can be split up to a maximum of six messages.

the limit of keyword entries for one message is 50, the entries can be divided to several massages. However, the total maximum of 300 entries will limit the massage to maximal six responses. Also the search options and parameters will be filtered before sending the results. Finally, all the received keyword entries from the various nodes, are presented with all details to the user.

5.4.2. Source search

After searching for a keyword of an existing file, a client receives a list of sources containing this keyword. Then the user selects one or more sources, which are added to the download list. Then the client searches constantly for sources of this list, where the download process is incomplete. As a downloading process includes only a part of a file. Many searches for sources must be executed per file.

The method for the source search is the same as for the keyword search so figure 5.7 is also valid for the source request. The only difference is that there are no search constrains, though the client searches only for the hash ID of the target source. It sends the KADEMLIA_SEARCH_REQ message $< sourceID : extention >$, where the extension is 0. Again, the packages contain maximal 50 source entries and there are maximal 300 source entries returned. If there

are any source entries found a KADEMLIA_SEARCH_RES $< sourceID : n* <$ $clientID : m* < METATAGS >>>$ message is replied. Here, n defines the quantity of returned peers having that file and m defines the quantity of information tags. The new obtained sources are added to the download queue of the file with that sourceID.

5.4.3. Note search

For each released file or file in the download list a search for a note is executed. This also includes the partfiles and files preview for download. Constantly, the client sends search requests for note of these files. That process is similar to the keyword and source search process. However, the request has another opcode to separated the notes and sources. The client sends the KADEMLIA_SRC_NOTES_REQ $< noteID >$ to peers in the tolerance zone of the nodeID. These respond with a KADEMLIA_SRC_NOTES_RES $< noteID : n* < clientID : m* < METATAGS >>>$ containing all its notes for that source. These are identified by the peer, which created the note and contains in the metatags the comment, rating and further information. In contrary to the search response of the keyword or source, there is only one note response because the maximum notes per file is 50.

5.5. The file transfer

This section gives an overview of a transaction for a file transfer. This consist of several subprocesses, where most are allready described. First, a keyword search returns a choice of different files. When the user selects one, the client will start a search for sources and notes for that file. If the clients has received some sources, it will start the real download process.

5.5.1. Download process

The download process starts when the downloading peer obtains a slot from the peer, which releases the demanded file. To get a slot, a peer demands all peers or their buddies of the obtained location information. Then the peer is added to a waiting queue. The position depends on the rating, which is described in the next section. When it is the turn of the peer it will start to download a part of a file also called *chunk*. These chunks allows that peers can allready download file from peers, which has still not the complete file. As the downloading is longer process it will be transacted over the TCP port and so it is not integrated into the Kad-protocol.

5.5.2. Credit system

This section describes briefly the credit system applied by the clients aMule and eMule. The reason for this system is to incite users, especially free riders to

upload more files. The obtained credits from a upload will determine the waiting time of user A in the download queue of user B. To avoid manipulations of the credits, they are saved locally at the client of user B, which owes the credits. So there is neither a global credit transfer among the clients nor can own credits be displayed.

The credit of client A is calculated with a *ratio* at client B. The ratio is the minimum of two different results, which are calculated the following way:

1. Ratio $= \frac{Totalupload[MB] * 2}{Totaldownload[MB]}$
2. Ratio $= \sqrt{Totalupload[MB] + 2}$

However these two equations have some constraints. If the *total upload* does not exceed 1MB, it will be set to 1. Also the *total download* will be set to 10, when it is 0. Furthermore the ratio is limited between 1 and 0. The minimum of the two ratios is included into the calculation of the final rating:

- Rating $= \frac{Ratio * waitingtime[s]}{100}$

Finally, the client with the highest rating will get the download slot. However, there are also some specialties that the rating of peers with an older application version is divided by two. Moreover banned clients are rated with 0.

6. Analysis framework

This chapter describes the setup and methodologies of the development infrastructure. This is above all the code modification of the open source application aMule. But a big part of the development was also the integration of the MySQL [39] interface and the creation of the database. This infrastructure builds a framework for a wide spread analysis and measurement. The final measuring of the obtained data is done with the aid of Perl scripts.

6.1. Development

The main part of the development is the modifying of the aMule client, the creation of the database and its interface and the evaluation of the results with scripts. Besides the functional development also maintenance and general improvements are implemented.

6.1.1. aMule

aMule is like eMule an open source software with the GNU licences [14], written in C++. At the beginning these two clients were strictly designed for the eDonkey protocol. Later on, the Kad protocol was added to the official source code. Even when the Kad protocol is separated from other functions, some important code is mingled with the original eDonkey code. Moreover the Kad uses implementations, which are designed for eDonkey, like the real downloading process or the socket implementation. This complicates the understanding and modifying of the Kad-protocol.

The wxWidgets [59] have an important role in the source code. They assure for example the support for Unicode, which guarantees the utilisation in different languages and special characters. Besides the handling of the strings, the wxWidgets library is also responsible for the socket, thread, IO file or timer functionality.

The principal changes for the understanding and measurement of the Kad protocol have been made in the exclusive files for the Kad protocol. Four sections for the main functionality are differentiated: **main file**, **routing table**, **network interface** and **search object** and **index**.

Main file

All the main tasks are managed and hold in the class **Kademlia**. Here all the other Kad classes are managed. A timer executes periodically the different processes.

Routing table

The routing table is represented by the **RountingZone**. These build a tree structure, where each RoutingZone is a node. The tree will be iteratively parsed through starting with the root RoutingZone. It handles the inserting, removing, splitting and merging of the nodes and leafs. Each leaf of the RoutingZone has a **RoutingBin**. The RoutingBin is a bucket, which holds a maximum of K **Contact**s.

Network interface

All outgoing and incoming messages have to pass the **KademliaUDPListener**. This network interface reads the incoming packets and redirects them to the target object. Also the outgoing messages are here created and transferred to the socket, which sends them to the target peer.

Search object

The two files **SearchManager** and **Search** are handling the search object. The manager is responsible for the whole lifecycle of the search, which includes; create, start, update, stop and delete. Furthermore, it allocates a search object to an incoming response. The search object described in figure 4.5 is responsible for the whole lookup and iteration process.

Index

The references are managed in the file **Indexed**. There the incoming metadata, location information and notes are managed. That consists of calculation of the *load*, the serialisation and the finding of reference for an incoming search.

It is important to mention that many other functionality or dependencies are implemented in the standard aMule files. For example all objects are managed central with the **amule** object. So the publishing of them is done in one file, where the Kad and the eDonkey protocol are implemented together. For the creation of a search, preliminary formats are done in standart aMule files, which were designed for eMule.

6.1.2. The database

The aMule client uses a MySQL database to make the obtained data persistent. Also has it the advantage that the obtained data of the parallel clients can be easily centralised. However, the most important aspect is that the analysis of the results is easier and faster with a optimised database.

For the optimisation, the tables in the database use the engine `InnoDB` [17]. This allows a faster writing into the tables, because several parallel aMule instances

cause a bottleneck by their simultaneous writing to the database. Another improvement is to put indexes on the attributes of the tables after the crawling is terminated. These speed up the SQL queries for the analysis.

As the aMule client is written in C++, it needs an interface to access the MySQL database. Therefore mysql++ [40] is used, which supports the major SQL queries. Finally each instance initiates one connection to the database over which it executes all the queries.

The whole structure of the tables in the database are illustrated in the appendix D. But each table and their main attributes will be described in this section. The tables *Words* and *Instances* are helper tables until now, which takes no direct influence into the measurements. However, the table Words is very useful to find a word, only having its hash value.

The tables **Routing table** and **Messages** describes the *contacts* and the *interacting peers*. It can be said how long a peer will stay in a relation with an instance and which types it passes through the time.

Routing table This table represents the routing table of an aMule client with all the contacts and their attributes (see table 3.2). By each change of an attribute, the actualised contact will be written into the database. So it can be followed the **lifecycle** of every contact in the routing table. For example the table contains the information about the creation, the differnet types during lifetime and when the contact is deleted. Furthermore, the actual time as the expiration time is notated by every change.

Messages is the table, which saves all incoming messages to the database. This allows to analyse not only the contacts in the routing table, but also to see all other peers, which are sending messages to the instances.

The *passive* listening is done with the tables **keywords**, **sources**, **notes** and **incoming searches**. These filter all incoming messages which another peer has initiated and sent to one of the instance, which is connected to the database.

Keywords holds the information about the metadata and their attributes (see table 5.1). Especially the `FILENAME` allows to analyse the file contents in the Kad network. For example it can be calculated the average keyword per file in the Kad network. But it can also be analysed the `FORMAT`, `TYPE` and so on. Anyway, to analyse all these attributes it is necessary to run a large number of instances to cover the Kad space. Otherwise a concentration on very popular keywords could distort the results. The `LOAD` shows the capacity of the instances for a certain metadata.

Sources is the table with the incoming publish requests for the *location information* and all its attributes shown in table 5.2. The possibility to distinguish between sources from a firewalled and a not firewalled peer is probably the most important. For example the sources, which have a `SERVERIP` and a `SERVERPORT` are "firewalled". Again, the `LOAD` gives new insights to the quantity of saved sources for each instance.

Notes holds all the incoming notes related to a certain source with a `COMMENT` and a `RATING` (see table 5.4). A note message contains also the `FILENAME`, but for the analysis it is less interesting than the `FILENAME` of the *Keywords* because the *notes* are sent only rarely.

Incoming searches are requests of other peers for references. When the attribute `isSource` is 0 the request is for a metadata. Therefore a `keywordID` and optionally a list with further keywords in Unicode is send. Otherwise a `source ID` is send to search for a location information.

The **publish responses**, **search responses** and **hello responses** are tables containing all responses for a request to a selected peer. These are especially for the analysis of the publishing and lookup process.

Publish responses are the responses from a peer after sending a publish request. This messages indicates the peers on which a reference was successfully published. However, when the `LOAD` is 100 the *overload protection* prevent the indexing of the reference on the host peer. As the KADEMLIA_PUBLISH_RES does not contain the clientID, it will be search in a list with all peers on which it was published. Seeing that the corresponding clientID is searched with $< IPaddress : messageport >$, peers reappearing with a new IP address will be assigned a `CONTACT ID` of 0.

Search responses is the table, were the search object is saved after an incoming search response. A search consists of r search responses, which are numbered by the `LAST_RESPONSE` attribute.

Hello responses contains all the *hello ping* responses: KADEMLIA_HELLO_REQ. A list with $< IPaddress : messageport >$ is created of all peers, which replied with a positive publish response. A hello request is sent after every 15 minutes to see the minimum of still connected peers.

The tables **request search** and **request result** describes the *iteration* process. The hops needed to find a close target can be deducted with these two tables. Also the jumped bits by each hop can be calculated.

Request search contains the outgoing messages for the iteration process. The attribute `FROM CONTACT` is the peer which sent the `CONTACT ID` of the peer to which the new request is send.

Request result are the incoming messages concerning the iteration process. The main attributes of this table is the `TARGET` object, the `CONTACT ID` and the XOR-`DISTANCE` between these two peers.

6.1.3. Word list

Actually there are millions of keywords in the Kad network. All distinct shared files and partfiles of the users, has to be multiplied by 7, the average keywords per filename. Most of these keywords were written in a official language like English or Chinese. But there are also some keywords which consist of just a sequence of numbers or letters. These words have to be searched, to get an overview of them and to analyse them. There are three principal ways to find the keywords in the Kad-network.

Movie list A search for words obtained from the internet movie database [8] and the music chart list. These are popular lists with nearly all existing film and music titles. These cover a large range of the existing keywords in the Kad network. But the disadvantage is the numerous of repeating words in the title like for example "the".

Dictionary A normal dictionary in the same major languages would also cover the most of the keywords in the Kad network. The advantage is that there are only rarely duplicated words. But it neither contains many pseudonyms nor proper nouns.

Iteration Another method is to lance a search for keyword, which is in the Kad network. The responses for that request will return the total filename containing the keyword. So the whole filename can be parsed and a new request can be send to the new obtained keywords.

Passive listening means that some spy clients are positioned as the nearest peer to a keyword. So this client will propably receive the most popular filenames containing that keyword. This effect can even be increased by choosing a popular keyword like "the", "www", "avi" or "mp3".

The implementation is a combination of the mentioned methods. The words out of the dictionary execute rapidly a vast search to the basic words. The passive listening will also gives the possibility to get rare keywords, which can be the starting point of the iteration in another language. Theoretically with this method nearly all possible keywords of the Kad network can be *crawled*. However, this is very time and resource intensive, so a realistic snapshot of some minutes is impossible.

6.1.4. General improvements

As the standard client is developed for the customary use, it has to be made some improvements of the aMule client. As it was necessary to install a client without a GUI. But the command client was only basic and it had to be expanded by other functionality. Some of these new implementations like the download via the command line were integrated in the official aMule version

[2]. The others like printing the actual routing table (compare appendix D), was only for the investigation of the Kad behaviours.

The firewalled state is disabled for the investigations, after testing that the clients were not behind a firewall or proxy server. Otherwise a client needs about 15 minutes to find out that it is not firewalled. This could distort the results of the first minutes. Especially the incoming requests would be blocked.

Another change is the disabling of saving the contacts at each shut down of the client, because a client saves only contacts with a type better than 4. So when a client is only after a few minutes restarted, it would be too short to validate the contacts and good contacts would be lost. Now, the client bootstraps each time from a self created list with about 200 contacts. This list has to be actualised continuously.

6.2. System infrastructure

The investigations followed the two main methods: **active** and **passive**. The active investigation was to send specific and well-directed messages to peers. This allows to find out specific behaviour of the Kad-network and its peers. The passive investigation is done by **spy clients**, which filter all incoming messages and write them into the database. These spy clients are well-positioned in the Kad space to span a **satelite system**.

6.2.1. Active investigation

The infrastructure of the active investigation exists of two components; the **publisher** and the **retriever**. Generally the publisher distributes references to selected peers. Afterwards the retriever starts a search for exact these references. These active components are controlled by a timer. This means that the publisher is executed every hour or every 24 hours and the retriever is executed for example every 15 minutes.

Publisher

The publisher is primarily for the distribution of references into the Kad-network. In this case it is used to publish unknown or very popular keywords to other peers close to the keywordID. Therefore it follows the standart aMule iteration process. This component is purposed for the following two measurements:

Publishing analysis To analyse the publishing 1000 references are published. This allows to messure the average publishing and iteration time and total messages. Furthermore with the help of this implementation, the republishing to same clients is measured.

Reference distribution Another purpose is the distribution of absolutely unknown keywords. Therefore the keywords are composed by three words. Between these words were the number keyword which increases from 1 until 50.

- word1word1word1instanceI
- word2word2word2instanceII
- word3word3word3instanceIII

Distributed load was investigated by sending publish requests to very popular keywords. So peers will respond with their load for the target: KADEMLIA_PUBLISH_RES $< target : load >$.

Some in the source code implemented constants had to be transferred to variables, which are given within the starting parameters. So it was possible to vary without recompilation, the tolerance zone and content replication for the publishing.

Retriever

The retriever has to locate the previously published references. Therefore it starts every 15 minutes an iteration process to find the published keywords. The timeout of the search is increased from 45 seconds to 5 minutes. Otherwise some results which takes longer than 45 seconds would be lost.

Furthermore the retriever pings periodically the clients on which a reference was published. This is done by a KADEMLIA_HELLO_REQ $< IP : messageport >$. As the availability is checked with the IP and not with a Kad-lookup process, the *aliasing effects* described by [3] is not considered. So the results depicts the minimum of available peers. The advantage of this method without an iteration is that it is more reliable and faster.

6.2.2. Passive investigation

The passive investigation consists of a *spy client* which is positioned in a *satelite system*. Generally the passive peers have only to listen to incoming messages. However, they will make all the behaviours like a standart client. This is for example the selflookup every four hours or the responding of incoming messages.

Spy client

A spy client is a peer, which listen to the incoming messages. It will filter them and write them to the database. Of course, it responds with a corresponding message, which assures that the peer shows the same behaviour like the other peers. This guaranties that the results are not distorted and that the spy client remains in secret.

The most import messages for spying are the metadata publishing requests. This information gives an overview of the file contents in the Kad network and of the host peers. But also the incoming *location information*, *notes* or *searches* are of great interest. So it can be seen that the incoming publishing requests for keywords are a multiple of the incoming searches for keywords. Finally, an optimisation of the performance is better done on the publishing of metadata.

Another possibility of the spy clients would be to manipulate the received data an respond with bogus data. It could also respond to specific data to make a DDOS attack.

Satellite system

The satelite system positionates the spy clients in the Kad-space. For the measurements, explained in the following chapter, three different satelite scenarios are used; whole space, tolerance zone and picked target. Each of them will be used for specific results.

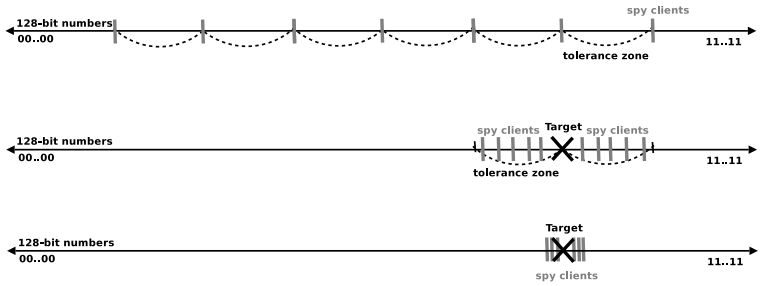

Figure 6.1.: The three satelite scenarios: whole space, tolerance zone and picked target

Whole space This structure positionate the client all over the Kad space in the same distance. The best way is to keep the same distance between the individual clients. For example, the clients can be dispersed that in every tolerance zone is one spy client. This scenario needs 256 peers and is for an overview of the whole network.

Tolerance zone is the satelite system, which has several clients in one tolerance zone. This carries out the investigation for any or a special keyword or source. This scenario allows us to analyse the distribution or searches for references.

Picked target The purpose is to place several clients so that they are the closest peers to the keyword. The measurements of the results indicated that the closest peers has about 20 first identical bits to a keyword. So here the clients were placed with at least 30 identical bits. Then it is possible to get all or the most messages concerning a certain target.

7. Measurements and Analysis

The setup and the implementations described in the previous section permit the analysis and measurements of the Kad processes. Particularly, the analysis of the keywords, iteration process, contact availability, publishing process and metadata distribution was measured as well active as passive.

7.1. Analysis of the Keywords

Mainly, the passive investigation of the Kad network made the analysis of the metadata's keywords possible. The spy clients were placed allover the 128-bit Kad-space. Finally, they received about 100000 incoming metadata publishing requests. These metadata had sometimes the same targets but they referenced to different locations. Moreover the evaluated filenames were written in divers languages.

The number of keywords per filename in the Kad-network is shown in figure 7.1. Here a keyword is interpreted as a sequence of at least **three** number or letter characters. So the keywords are separated by special characters like white spaces, hyphens, points, etc. The absolute values show that about 15% of the filenames contain five keywords. Only 3% of filenames have more than 20 keywords. Generally, the CCDF of the keyword distribution shows the same characteristics like the CDF of eDonkey files analysed by [12]. However, now in the Kad network the filenames exists on average of 2 more keywords.

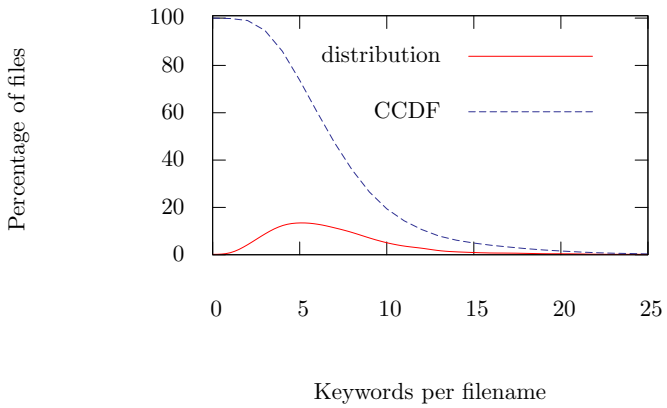

Figure 7.1.: Shows the distribution of keywords per filename.

7.2. Analysis of iteration process

In this section the iteration process is analysed and measured. It is difficult to make an exact measurement, which are statistically significant. This is because the RTTs of a request vary between the large scale from 5 until 30 seconds. When an iteration process has to execute more requests sequentially, the difference between each iteration process will be even more significant. However, the following diagrams will give a first impression of the needed time and messages for a lookup.

7.2.1. Iteration time

To analyse the time for one iteration process, 1000 different metadata have been published. Then it was measured the total time to publish one keyword. In this experiment unknown or popular keywords make no difference. Figure 7.2 shows that after about 20 seconds, a peer in the tolerance zone with 8 until 14 bits is found. From then on it needs 200 seconds to find 10 other peers in the tolerance zone. So, the most time of a publishing process is used to fullfill the content replication. The high iteration time of the tolerance zone with 16 bits is caused by the fact that there exist only a few peers. In a tolerance zone of 17 bits are even less peers, so after a shorter time all of them will be found.

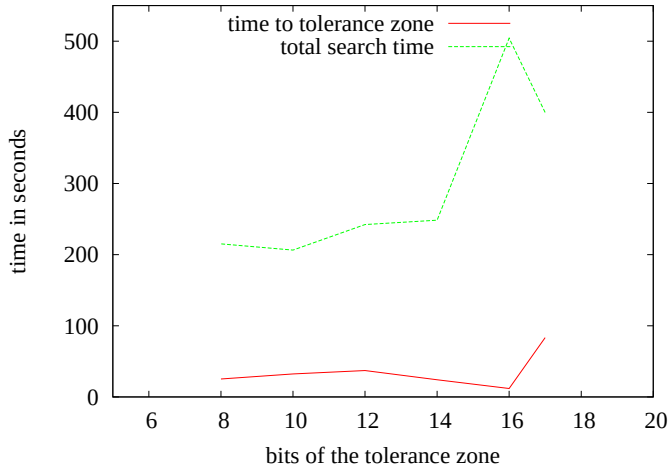

Figure 7.2.: The publishing times are illustrated for different tolerance zones. It shows the iteration time until the first positive publish response and also the total time for reference publishing with a content replication of content replication factor **r**=11.

7.2.2. Iteration messages

For the analysis of the iteration messages, it was used the same basis like from the iteration time. Again figure 7.3 illustrates a same result, that there is

no significant difference of a tolerance with 8 or 14 bits. About 22 iteration requests are needed to place 11 references in the tolerance zone. By subtracting the request to find the responding target peer, 1 effective publishing request needs 1 iteration request. From a tolerance zone with 16 bits, the needed messages increase rapidly.

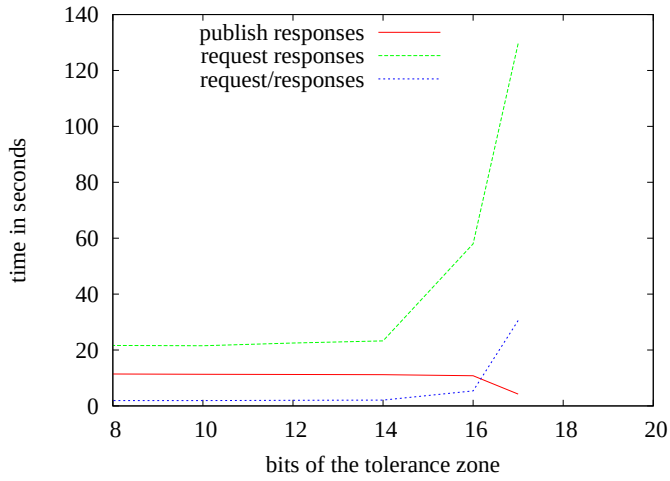

Figure 7.3.: Shows the average quantity of iteration messages which are required for 11 content replications.

7.3. Analysis of peers

The analysis of the contacts investigates mainly the availability of the peers. This information will determine the publishing and lookup performance in a real DHT network, where the peers have a considerable fluctuation, also called *churn*.

7.3.1. Firewalled peers

Probably the only way to find out the proportion of firewalled peers in the Kad-network is with passive results. Each incoming publish request for a source indicates if the peer, holding the files has an buddy. So the distinct peers having a buddy indicates the proportion of firewalled peers. The analysis of 100000 incoming sources over the analysing time shows that only 56.2% of the peers have the status "connected". 44% of the peers are either within a NAT or blocked by a firewall. The other 0.4% are peers which changed their state, for example by deblocking their firewall.

7.3.2. Peers with the aliasing effect

The *aliasing effects* described by [3] is responsible that peers disappear and afterwards reappear with a new location $< IP : messageport >$. Figure 7.4

gives an extract of the influence of this effect to the availability of the peers. For this experiment, 50 unknown keywords were published on 550 peers. Afterwards they have been searched by the publishing peer. The percentage of peers with a new location among the found peers is nearly negligible for the first 10 hours. But after 24 hours it increases constantly up to 20%.

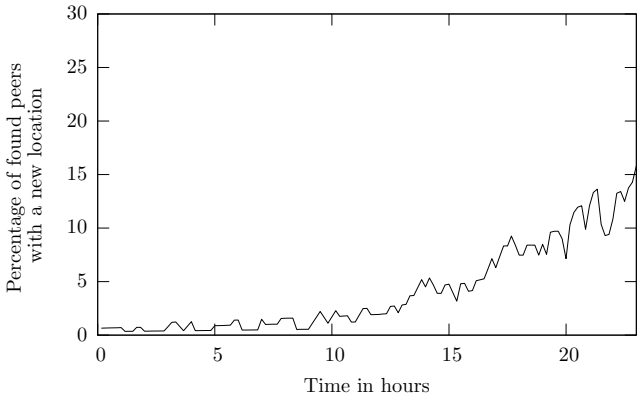

Figure 7.4.: The curve represents the percentage of found peers with our published reference, which have changed their location $< IP : messageport >$.

However, it is important to mention that this effect has no influence to the publishing process. When a client is disconnecting, it will save all his obtained references, not older than 24 hours. By rejoining the Kad network with the same ID, it will provide all his references as before. The fact of a different location only increase the latency and not the availability.

7.3.3. Host availability

This section analyses the availability of the **host**, a peer containing the published metadata. This experiment publishes 50 random and unknown metadata to 11 peers each. Every 15 minutes a *Kademlia Hello Request* was sent to the initial location $< IP : messageport >$ of these 550 peers. So obviously within this ping, the aliasing effect was not considered. Anyway the results specify the minimum of available peers. The total availability can be estimated in comparison with the results of section 7.3.2.

Figure 7.5 describes that for the first 12 hours 2 peers stay for nearly 100 % connected to the Kad network. Furthermore it can be said that at least six clients will be available after 12 hours with the propability of 50 %.

A smaller tolerance zone has no significant influence to the availability of peers, as the map on figure 7.6 depicts. So the *quality*, time to stay alive, of the peers selected from a small tolerance zone is the same as peers from a large tolerance zone.

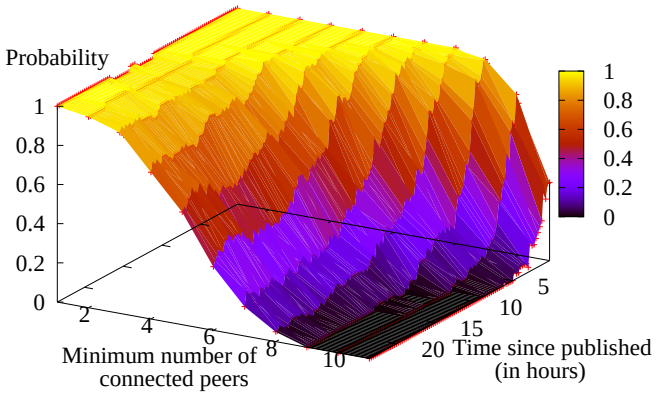

Figure 7.5.: Shows the availability of peers depending on the time. The peers are a selection of 550 peers which participated in a publishing process in a tolerance zone of **8** bits

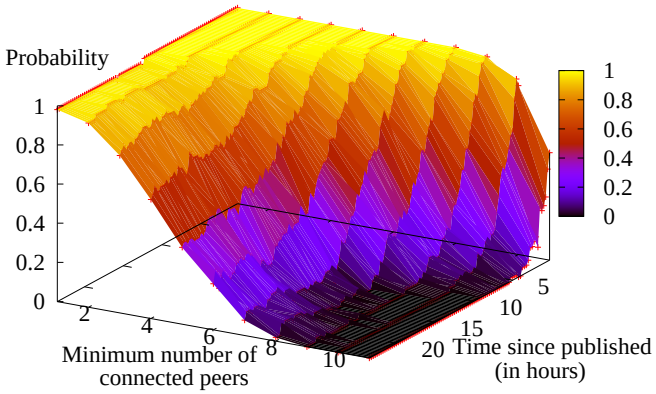

Figure 7.6.: Shows the availability of peers depending on the time. The peers are a selection of 550 peers which participated in a publishing process in a tolerance zone of **14** bits

7.4. Publishing

In this section, the publishing performance is measured. Therefore metadata is published under different constraints, principally biased on the tolerance zone and on the content replication. But it is also analyses with different contacts in the routing table. Afterwards the published metadata is retrieved with the standard search algorithm of Kad-protocol.

7.4.1. Standard Publishing Performance

First, standard publishing process, implemented by aMule and eMule is analysed. Therefore a client publishes 50 random and unknown keywords. It will be assured that each metadata is sent to exact 11 other peers. After the publishing process is finished, a client sends periodically every 15 minutes a Kad search for the previously published metadata. As the lookup process depends on the contacts in the routing table, it will be differentiated three different states: **consecutive contacts**, **identical contacts** and **random contacts**.

consecutive contacts is a routing table, which keeps its initial contacts and adds new incoming peers to the table. Primarily, these are obtained within the publishing or the lookup process. Obviously, the contact list contains new obtained *close* peers to the target, after the publishing process. So the second iteration will generally find close neighbour peers of the target. Figure 7.7 illustrates the lookup performance with consecutive contacts.

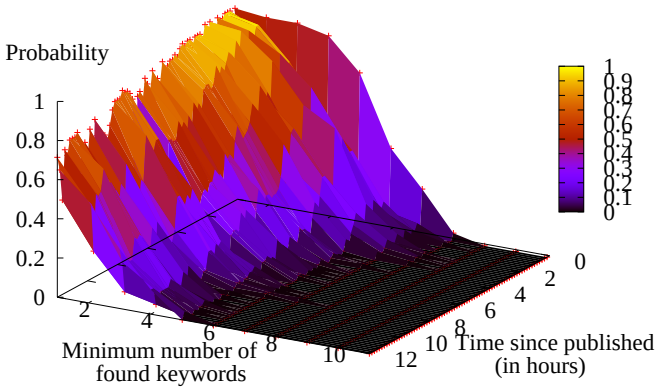

Figure 7.7.: The propabilities to relocate a least $50 * 11$ published metadata. The keywords are searched with the standard Kad process and with **consecutive contacts**. (Tolerance zone of 8 first bits and r=11)

identical contacts means, that the publishing and the lookup process are started with the same contacts. Therefore the same list with 160 contacts is loaded before each iteration process. Figure 7.8 shows the search performance when the contacts are initialised before each search.

random contacts are 20 randomly chosen contacts. These peers are obtained within a *bootstrap* request to a peer connected to the Kad network. This guarantees totally independent searches with different starting peers.

Most remarkably it can be seen that a lookup process with *identical contacts* or *random contacts* is better than a lookup with consecutive contacts. This is

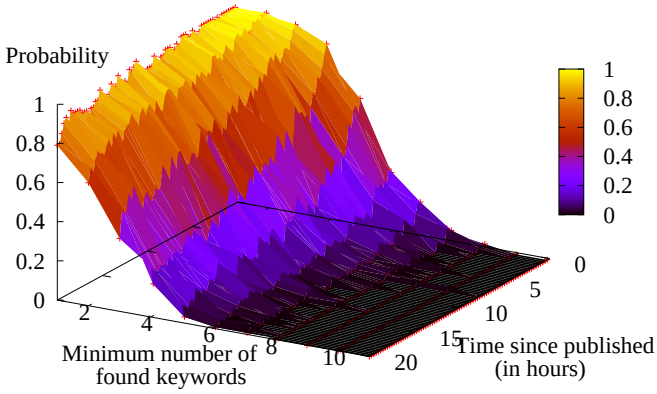

Figure 7.8.: The propabilities to relocate a least 50∗11 published metadata. The standard Kad lookup process used a list of 160 **identical contacts** for each iteration. (Tolerance zone of 8 first bits and $r = 11$)

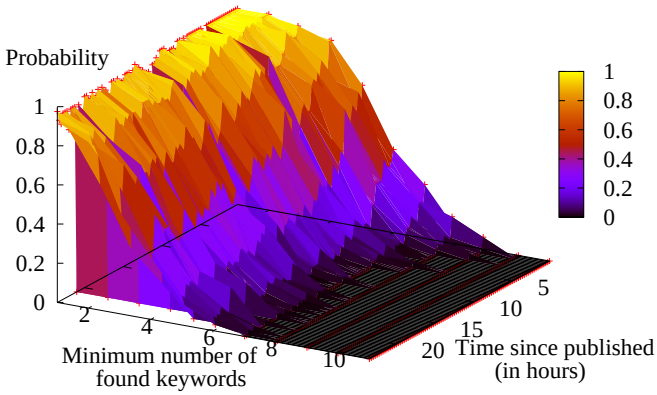

Figure 7.9.: The propabilities to relocate a least $50 * 11$ published metadata. The standard Kad lookup process used a list of **random contacts** for each iteration. (Tolerance zone of 8 first bits and $r = 11$)

caused by the fact that the consecutive contacts are too close to the target and though it is impossible to find published metadata, which is published on peers with a higher XOR distance. This highlights that a search to **once published** keywords is more successful with less contacts. Also shows this, that executing a second lookup would decrease the propability to found a published metadata.

7.4.2. Variation of the size of the tolerance zone

The next figure 7.10 has the same setup then the previous figure 7.9. But now the tolerance zone is reduced from 8 to 14 first bits. So, it can be seen that the probability to find peer is only a slightly higher.

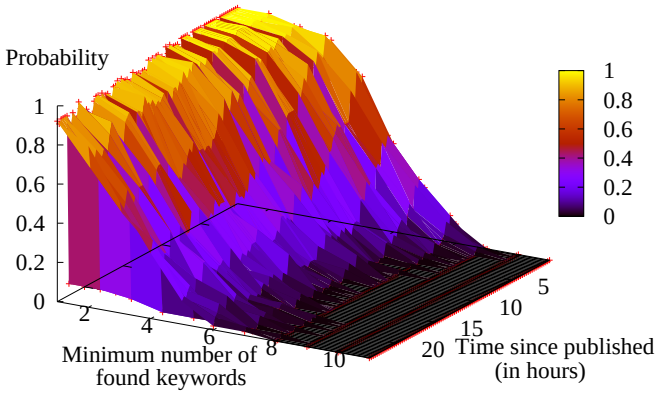

Figure 7.10.: The propabilities to relocate a keyword within the Kad search. The tolerance zone is defined to the 14 first bits and $r = 11$.

The results are even better when the tolerance zone is reduced further on to 16 bits. But they are no more representative because it is difficult to find sufficient peers. With a tolerance zone of 17 bits, it was even absolutely impossible to find exact 11 peers.

7.4.3. Variation of the content replication factor

This section shows the influence of the content replication factor to the location of published keywords. Therefore 50 random and unknown keywords have been published for 3, 5, 8, 11 and 16 times. Figure 7.11 illustrates the probability to retrieve at least one of these keywords with a tolerance zone of 8 bits. After 18 hours it can be said that $r = 8$ finds a published metadata with a propability over 0.9. In figure 7.12 the tolerance zone is is narrowed to 12 bits. It illustrates that for the first 12 hours the content replication can be divided in half to 5. A metadata is nearly always retrieved. But after 12 hours the retrieving of published metadata with a content replication lower than 11 cannot be insured.

7.5. Search time

This section analyses the time to retrieve a published metadata. Figure 7.13 shows the necessary time to retrieve a metadata with the aMule standard tolerance zone of 8 and $r = 11$. A metadata is retrieved with the probability

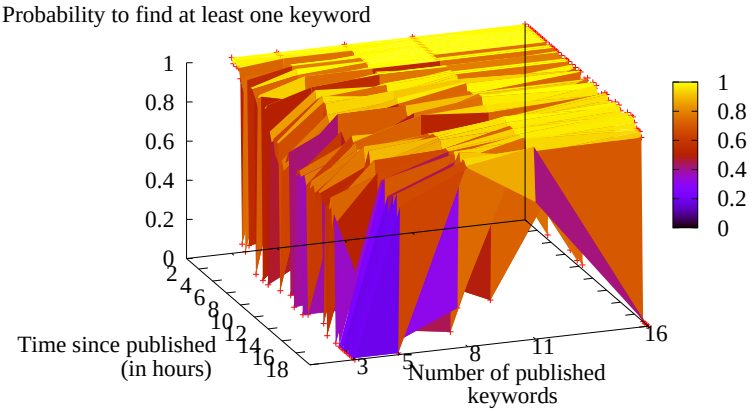

Figure 7.11.: The probability to find at least one keyword in a tolerance zone of **8** bits.

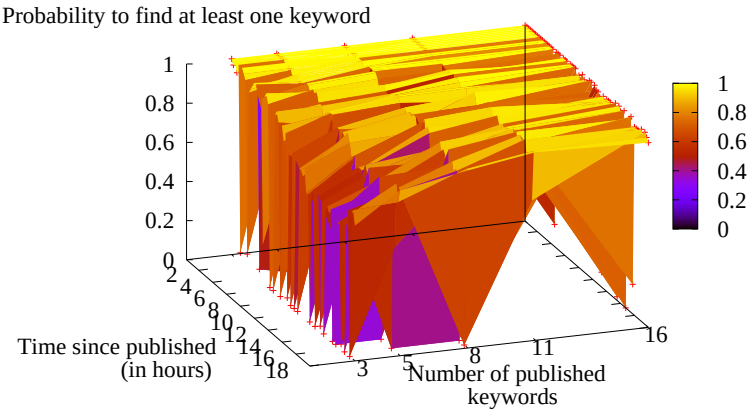

Figure 7.12.: The probability to find at least one keyword in a tolerance zone of **12** bits.

of 60% before 15 seconds. It is insured that a metadata is found before 45 seconds. The retrieving time of one published metadata decreases slightly over the time. So 24 hours after the publishing, a client needs more time to retrieve that metadata.

7.6. Metadata distributions

For the measurement of the metadata distribution several clients have been started with different parameters like the size of the tolerance zone and the content replication. Each of them published 1000 different and unknown keywords. The amount of contacts in the routing table has been constantly between

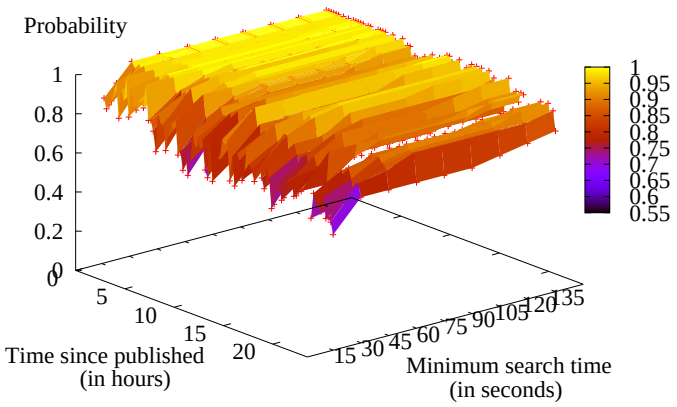

Figure 7.13.: The probability to find the first peer containing a published keyword within x seconds.

230 and 270 during the publishing process. As the metadata were published sequentially with one second delay, the whole publishing took about 30 minutes. The tolerance zone has 8 first identical bits and $r = 11$, if there is nothing else specified.

7.6.1. Distribution in different tolerance zones

The publishing process iterates until a peer close to the target and then searches its neighbours. Though it can be assured that the keywords will be published on peers with a small XOR distance even in a large tolerance zone. In figure 7.14 the tolerance zone of 8 bits starts the publishing on peers which only have the first 8 bit identical. But the most keywords are published at peers, which have at least 15 identical bits.

Algorithm 7 defines the position of the host peer. If the distance is negative, the host peer is between the target and the publishing peer. Otherwise it is behind the target. The following figures depict that the distance on the left and on the right side is, apart of small differences, symmetric.

7.6.2. Distribution with different content replications

Figure 7.15 presents the distribution of metadata based on different content replication. The effect can be seen that a smaller amount of content replication can be published on the few closest peers. But when the metadata is published more often, the client must also publish the metadata also at same peers with a larger XOR distance to the target.

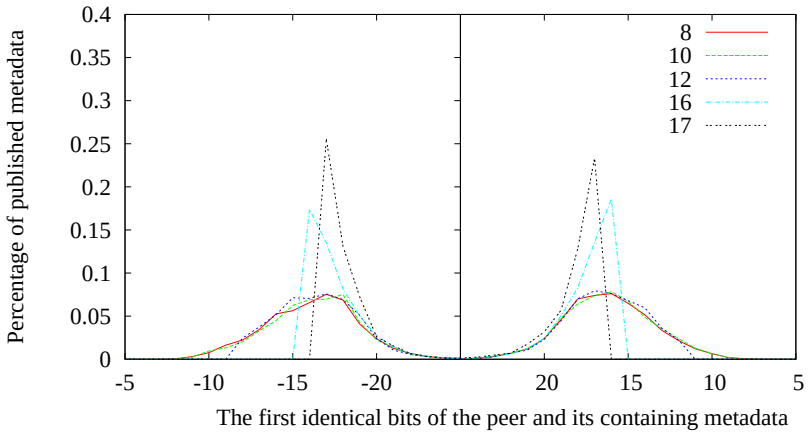

Figure 7.14.: Shows the percentage of metadata, which is published on a peer with x identical bits. The different curves illustrates the varying tolerance zone from 8 until 17.

Algorithm 7: Algorithm to assign a positive or a negative distance to the contact.

```
if publishing peer > target then
    if contact > target then
        distance = distance * 1;
    endif
else
    if contact < target then
        distance = distance * −1;
    endif
endif
```

The main conclusion is that peers publish most references at the closest peers to its hash ID. This is an important effect, which allows to relocate the references within a lookup process.

7.6.3. Distribution of republished metadata

This section describes the republishing of data item. The aMule client publishes a metadata every 24 hours and a source every 5 hours. The next tests republish 1000 data items every hour. Afterwards, 20 publishing for the same 1000 data items are compared to each other.

For the analysis of the republished metadata distribution, the same keywords were periodically published. After each publishing, the XOR distance was calculated of the published keywords and their host peers. Astonishingly, the results in figure 7.16 illustrates that there is nearly the same metadata distri-

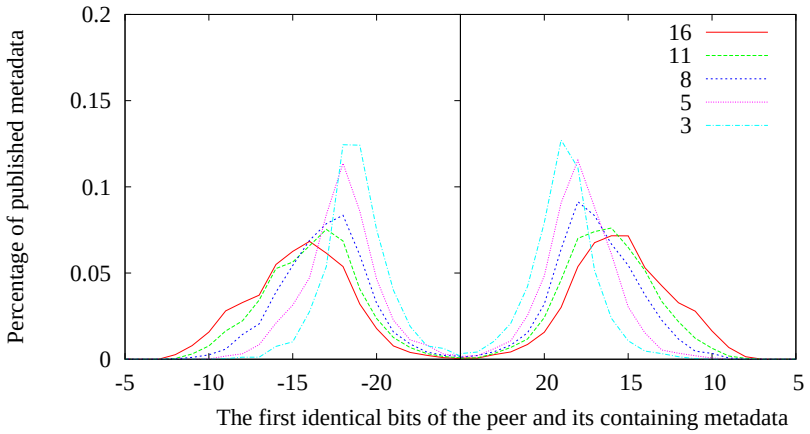

Figure 7.15.: Shows the percentage of metadata, which is published on a peer with x identical bits. Each curve represents a different value for r.

bution after each republishing. So at each republishing the iteration process has the same shape in regard to the XOR distances.

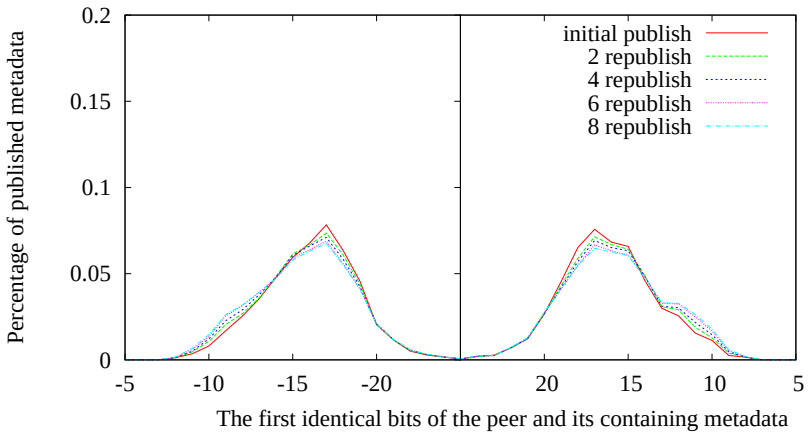

Figure 7.16.: Shows the percentage of metadata, which is published on a peer with x identical bits. Each curve represents a n-th republishing, whereas a publishing is executed every hour. In this case the tolerance zone has 8 bits and $r = 11$.

The next step is to examine the propability that a republishing will find the same peers on which they published the first time. Figure 7.17 shows the republishing for different content replications. The content replication of 3 republishes the most on same peers. This indicates in comparison with figure 7.15 that most of these three peers are in a small subset with the closest peers.

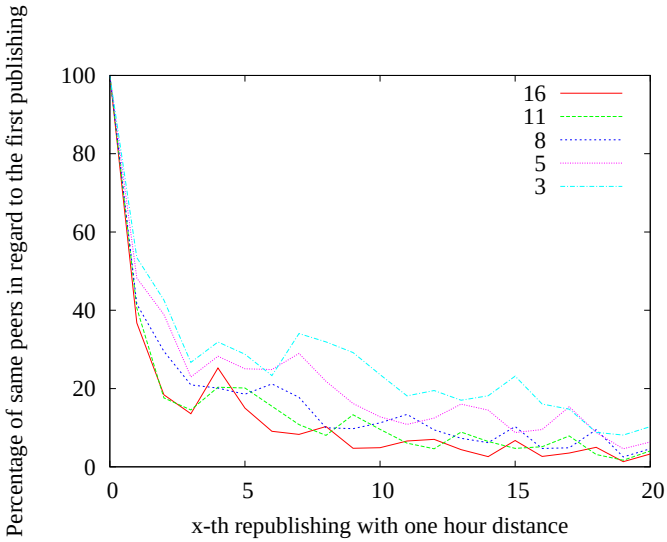

Figure 7.17.: Illustrates the percentage of republishing on same peers like the first publishing. The republishing is executed every hour to the same targets with a varying content replication from 3 until 16 and a tolerance zone of 8 bits.

Figure 7.18 shows that the second publishing after an hour finds barely 40% of the peers from the first publishing then it will decrease continuously under 20 %. However in small tolerance zone when there are not enough alternatives, the republishing serves more often of the same peers. The curves of smaller tolerance zones shows a zigzag because there are only a few peer and though it is more sensible to the churn.

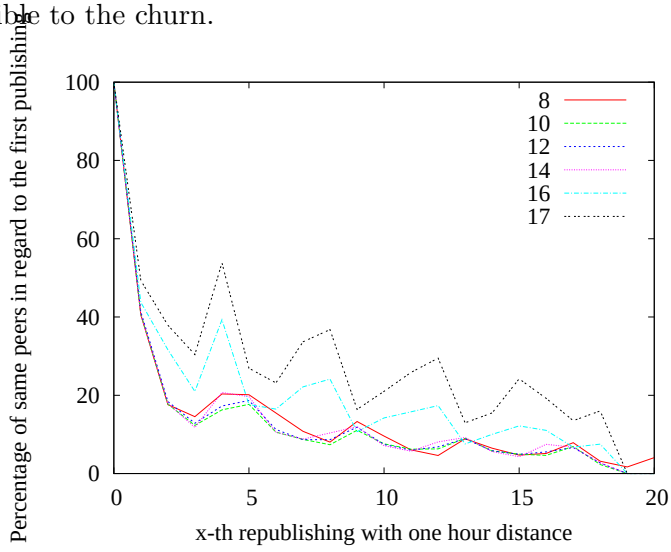

Figure 7.18.: Depicts the percentage of republishing on same peers like the first publishing. The republishing is executed every hour to the same targets with a varying tolerance zone from 8 bits until 17 bits and $r = 11$.

Especially peers with a smaller XOR-distance to the target have a higher propa-

bility to receive a second publishing request (Figure 7.19). This is caused by the iteration process, which does not start at the closest peers to the target. So there are many possible clients to publish on. Also will each iteration takes a different sequence on peers to approach the target. But the closer the iteration comes to the target, the higher is the density of clients. When there exits only 5 or less clients, the propability of publishing at a same client will increase.

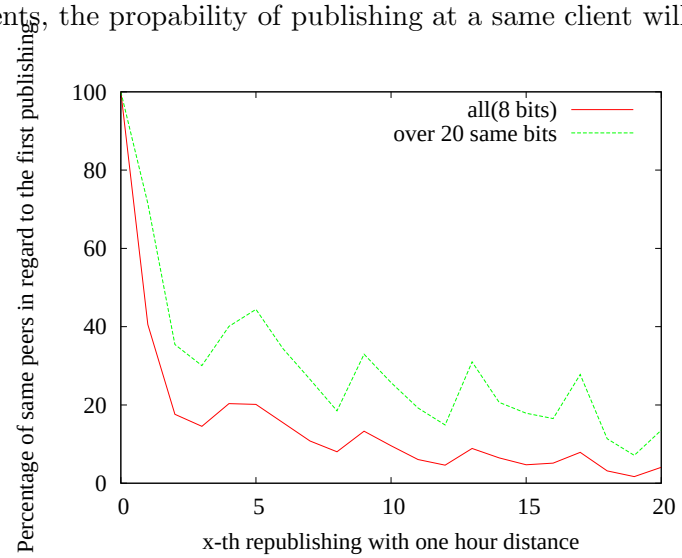

Figure 7.19.: distance

7.6.4. Cumulative publish distribution

The publish distribution CDF in figure 7.20 describes the distances of the targets to their host peers. The targets have been published in the Kad network with over 1 million peers. So 60% of the metadata is published on peers which have at least the first 15 bits identical. Furthermore it can be seen that some metadata is published on peers with more than 20 identical bits. As the following calculation describes exists approximately 1.43 peers on average with 20 first identical bits. So it can be seen that the iteration process find direct neighbours of the metadata.

$$\text{Peers with same 20 first bits: } \frac{1500000}{2^{20}} = 1.43$$

7.6.5. Popular keyword distribution

Figure 7.21 shows the distribution of metadata with the popular keywords "lucas", "movie" and "lucky". The data was obtained within a satelite system spanned over its tolerance zones. The results confirm the insights of the active analysis of the distribution, that the publishing of the keywords is mostly done on peers close to the target.

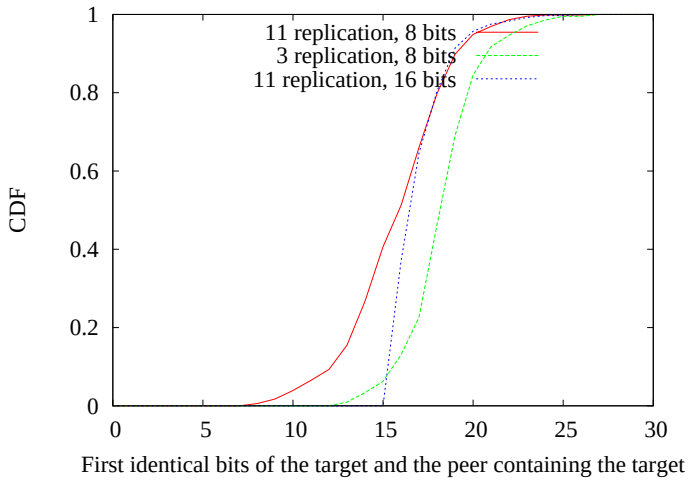

Figure 7.20.: Publish distribution CDF in different tolerance zone and different content replication.

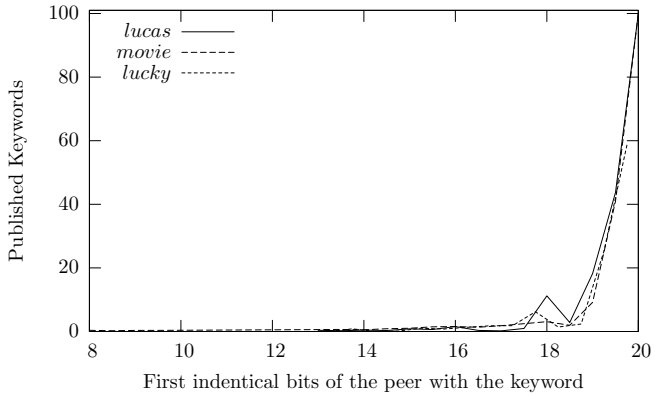

Figure 7.21.: Shows the quantity of incoming metadata for a peer. The ID of the closest peers has 80 identical first bits to the keyword hash.

7.7. Peer distribution

This section analyses the distribution sequence and distance of the target peers. Therefore aMule instances with varied parameters publishe each 1000 metadata. A tolerance zone of 8 bits limits the publishing of data item to peers with at least 8 identical bits. But the distribution of the peers with their distance is not indicated in detail. So figure 7.22 shows the average identical bits of the host peers to the target. The furthest host peer has on average 12.2 identical bits in a tolerance zone of 8 bits. This is a lot further than the permitted 8 bits.

By each iteration *hop* to peers in the tolerance zone an action is executed. When this action is a content publishing, peers with a larger distance will host the

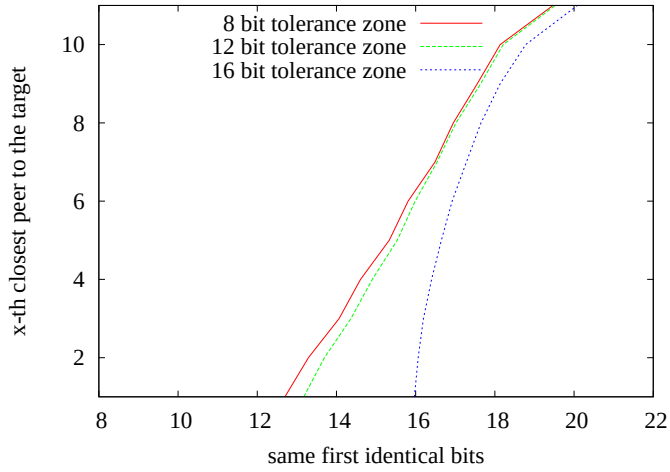

Figure 7.22.: The peers are sorted by their XOR-distance to the target. So it can be seen the average distance of the peers.

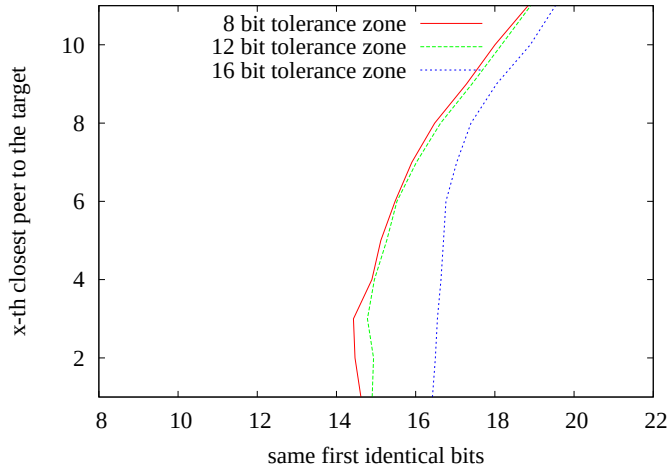

Figure 7.23.: It shows the average distance of the peers, which are sorted by the time when they were published.

data item. So figures 7.24 and 7.25 show the propability of retrieving data items in dependence to the host peer distance. They clarify that the metadata on the furthest peer is only found with the propability of 10 %. After some time it will even worsen. One of the 4 furthest bits is found with the propability of under 80 %. So the keywords are primarily managed by the first few close peers. In figure 7.22 it is already shown that the 4 furthest peers have on average between 12 and 14 identical bits.

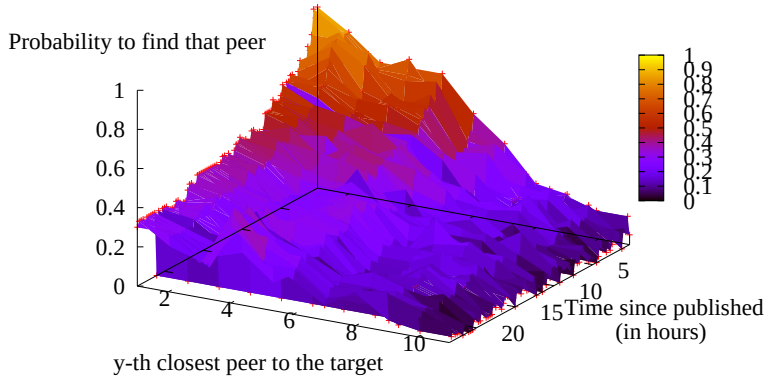

Figure 7.24.: It is shown the propability to find the closest peer with the XOR distance, the second closest peer and so on. The 11th peer on which a metadata is published, is the furthest peer and has the lowest probability to be retrieved. The tolerance zone is **8** bits and r is 11.

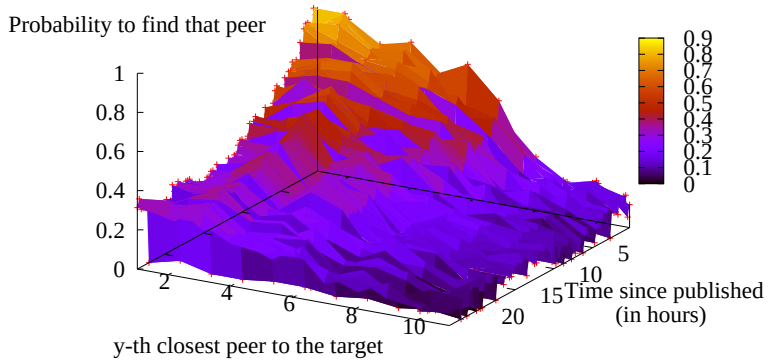

Figure 7.25.: It is shown the propability to find the closest peer with the XOR distance, the second closest peer and so on. The furthest peer on which a metadata was published is the 11th. The tolerance zone is **14** bits and r is 11.

7.8. Publish load

The analysis of the load permits to know more details about popular files. Primarily, the most popular file was searched, therefore a list of top 50 keywords was created. This was generated with the help of incoming messages and with test searches on a normal aMule client. Afterwards several aMule instances pub-

lished metadata with this keywords. Each response to these requests includes the load for the keyword on the individual peers.

Figure 7.26 shows the keywords "the", "www", "com", "mp3" and "net" as the top 5 Keywords in the Kad network. The keyword "the" is clearly recognised as the most popular identifier for a metadata. The average load is calculated for peers with the same XOR-distance:

- $average \quad load = \frac{\sum_{k=1}^{distinct \quad peers} (load)}{distinct \quad peers}$

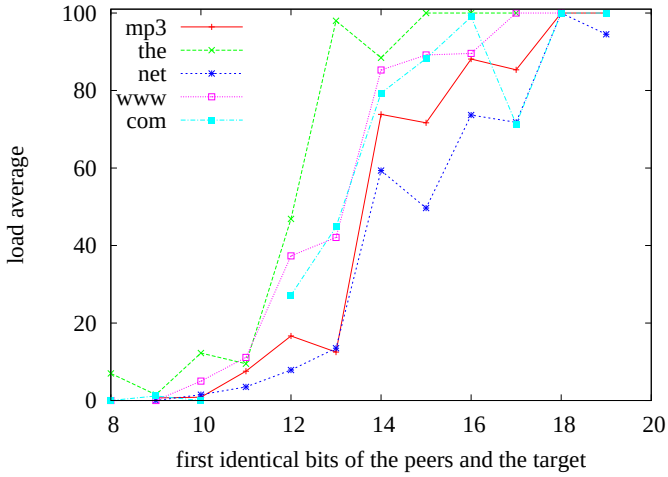

Figure 7.26.: The average load around the different keywords shows that the keyword "the" is the most published.

The next figure 7.27 presents the occurrence of peers with their load. There are many peers with a distance less than 13 bits and a load smaller than 20. This cluster is caused by the first iteration hop into a subset with over 100 peers. In contrary peers with at least 15 identical bits have mostly a load of 100, because there is only a small subset of peers. For example there are 3 peers with 18 identical bits and a load of 100. The peers in this cluster are found by many publishing peers.

The load depends besides the distance on the target also on the up-time of a peer. A recently connected peer needs some time to accumulate the incoming metadata. In figure 7.28 the difference between the maximum and the minimum load of peers is significant.

7.9. Controlling a data item

This section evaluates the possibility to control a data item by placing fraud peers as close as possible to the data item. The iteration process finds these

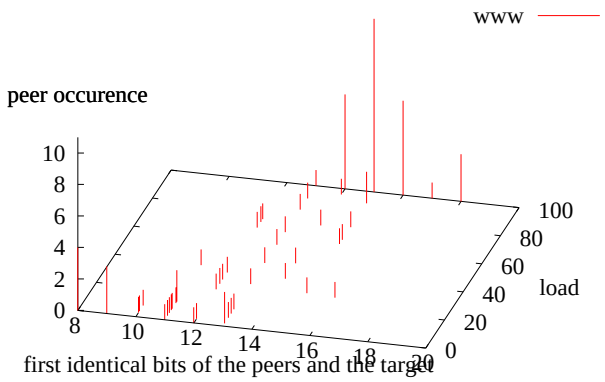

Figure 7.27.: Illustrates the absolute load distribution on peers around the MD4 hash of "www".

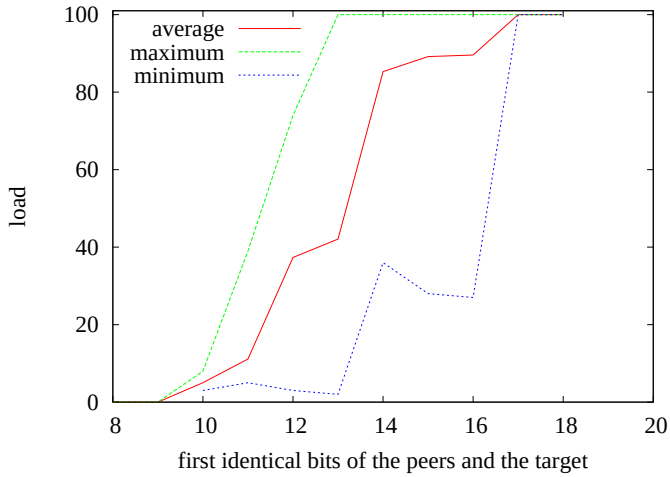

Figure 7.28.: Shows the minimum, average and maximum load of peers that have the first x bits identical with "www".

peers because they are the closest peers to the target. A fraud peer can manipulate a data item or let it disappear once it is the host peer. If a subset of fraud peers receives all published content of a data item ID, they would have total control over them. Figure 7.24 shows that the furthest 5 peers are rarely found. This leads to the fact that controlling the 6 closest peers could be sufficient to manipulate data in the Kad-network. Some film or music institutions have a large interest in this attack, as they have already started the pollution attacks.

In this experiment 18 aMule instances are started with at least 30 identical bit to the target "the". Choosing a popular target insures that there are enough incoming publishing messages. A load of 100 was reached on the closest peers during the measurements in less than one hour. Table 7.1 shows information

running time (hours)	time GMT	messages
28	17:20:00	808589
31	20:20:00	743446
37	02:20:00	231868
43	08:20:00	461607

Table 7.1.: Shows the details about the representant closest instance

about the running time and about the number of incoming publish requests. It can be seen that the instances receives most messages in the evening, because "the" is represented mostly within Europe.

Figure 7.29 indicates the possibility to take partial control of the metadata for the keyword "the". It is supposed that an iteration process reaches one of the 18 closest peers and executes a publish request. Four identical metadata items from the same publisher peer can be controlled with the probability of 80%. These are the crucial peers, which are found by the retrieving process during the first 12 hours (figure 7.24). However, some modifications can even improve these results:

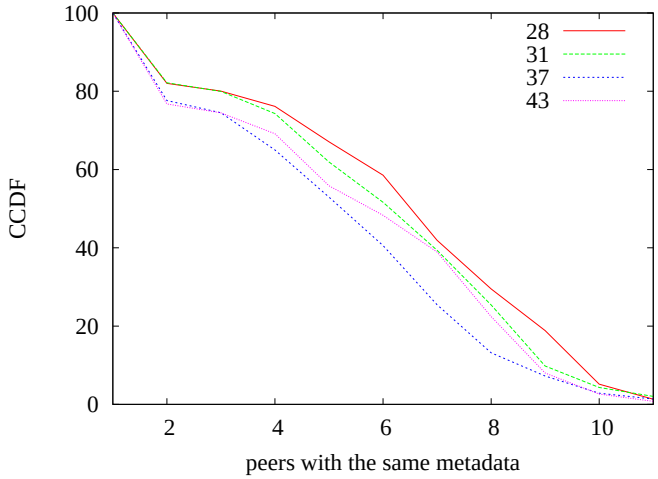

Figure 7.29.: Probability that a same metadata from the same peer is published on x peers.

- Some of the competitor peers on which it was published will leave the network, as a consequence of the churn. (see figure 7.5)
- The longer the instances are connected the better they will be known and the more they will attract messages.
- A structured architecture and the interconnection among the instances can also increase the control to a certain data.

Figure 7.23 shows the first five contents are published on peers with not more than 15 identical bits. To control this subset of peer a lot of more instances are necessary. The tolerance zone has the advantage that it is easier to implement, than the finding of the exact r closest peers. But it is also necessary to protect the data item against attacks.

8. Conclusion

Until now there have been papers describing a theoretical approach to DHT-based overlay networks only. But the Kademlia-based file sharing applications which recently appeared with about 1.5 millions users enabled new possibilities. So this research report is the first to describe the structure and processes of the Kad protocol in detail. Furthermore, this Master Thesis studied empirically the behaviours of the Kad-protocol implemented by the aMule and eMule file sharing application.

The description of the Kad protocol is based on the examination of the aMule client. Therefore the client was indispensably modified to obtain exact information about the structure and behaviour of the routing tree, the contacts, the lookup algorithm and the publishing and retrieving process. Some of the modifications are now integrated in the official version of the aMule client. Moreover the description of the behaviours was experimentally prooven by running several tests.

A more extensive framework with spy clients, a satellite system and an integrated database, enabled the analysis and measurement of the Kad-protocol. The main objective was the exploration of the publishing process. This included the examination of the tolerance zone and the content replication, as well as the analysis of the content distribution. But also the possibility to control a keyword is tested.

8.1. Discussion

The current parameters and processes of the Kad-protocol assure the successful publishing and retrieving of data items. However, after 12 hours the churn of the peers is responsible for one half of the peers leaving the network. This has a high impact on the search results, because the retrieving of published metadata is not anymore insured. So a reduction of the publishing frequency by a half to 12 hours would improve the data retrieving in the network.

As the measurements illustrates, a tolerance zone of 8 identical bits is to large for a network with over 1 million peers. A data item has to be searched in a subset of over 4000 peers, whereby only the closest peers are retrieved. Reducing the tolerance zone to at least 14 bits, increases the probability to retrieve more data items.

After reducing the republishing time to 12 hours and the tolerance zone to 14 bits, the content replication can also be adapted. A dividing of the content replication by half counterbalances the increased network traffic caused by more

republishing. By doing so a retrieval of at least one data item is more probable than with the standard parameters after 12 hours.

However, the optimisation of the Kad-protocol parameters also causes a danger of attack. With the reduction of the tolerance zone, less peers are responsible for a data item. This makes it easier for fraud peers to take control of a certain metadata.

8.2. Future work

The Kad-protocol is relatively recent in application and therefor there are only a few research studies about it. This opens further work on different parts of the protocol:

- The created framework already has the infrastructure for further measurements. This allows for example a study about the lifecycle of the peers in the routing table. Already the working implementation exists to record all attribute changes of the routing table contacts to the database.
- While the publishing and distribution performance is analysed in detail, the search performance rest to be analysed. The next step is to optimise the retrieving process for published data items.
- The possibility to take control of a data item is already shown in the end of the measurements chapter. An attack to a popular keyword was already analysed. Further on, the results can even be increased by changing and structuring of the peers among each other.
- Another future work is the analysis and optimisation of the *load* and its expansion of new feasibility. Therefore the tradeoff between varying the republishing frequency versus reducing the content replication of popular data items can be studied.
- An more extensive work is the change of the 2-level publishing scheme. At the moment the published metadata of a file is under the control of the host peer. That responsibility can be entrusted to the 1st level peer, which checks the availability of the file with a ping [12].

A. aMule definitions

Constante	Value	Description
KADEMLIAASKTIME	SEC2MS(1)	1 second
KADEMLIATOTALFILE	7	Total files to search sources for.
KADEMLIAREASKTIME	HR2MS(1)	1 hour
KADEMLIAPUBLISHTIME	SEC(2)	2 second
KADEMLIATOTALSTORENOTES	1	Total hashes to store.
KADEMLIATOTALSTORESRC	2	Total hashes to store.
KADEMLIATOTALSTOREKEY	1	Total hashes to store.
KADEMLIAREPUBLISHTIMES	HR2S(5)	5 hours
KADEMLIAREPUBLISHTIMEN	HR2S(24)	24 hours
KADEMLIAREPUBLISHTIMEK	HR2S(24)	24 hours
KADEMLIADISCONNECTDELAY	MIN2S(20)	20 mins
KADEMLIAMAXINDEX	50000	Total keyword indexes.
KADEMLIAMAXENTRIES	60000	Total keyword entries.
KADEMLIAMAXSOUCEPERFILE	300	Max number of sources per file
KADEMLIAMAXNOTESPERFILE	50	Max number of notes per entry
KADEMLIABUDDYTIMEOUT	MIN2MS(10)	10 min to receive the buddy

Table A.1.: The kademlia time variables

Constante	Value
SEARCHTOLERANCE	16777216
const unsigned int K	10
KBASE	4
KK	5
ALPHA_QUERY	3
LOG_BASE_EXPONENT	5
HELLO_TIMEOUT	20
SEARCH_JUMPSTART	2
SEARCH_LIFETIME	45
SEARCHFILE_LIFETIME	45
SEARCHKEYWORD_LIFETIME	45
SEARCHNOTES_LIFETIME	45
SEARCHNODE_LIFETIME	45
SEARCHNODECOMP_LIFETIME	10
SEARCHSTOREFILE_LIFETIME	140
SEARCHSTOREKEYWORD_LIFETIME	140
SEARCHSTORENOTES_LIFETIME	100
SEARCHFINDBUDDY_LIFETIME	100
SEARCHFINDSOURCE_LIFETIME	45
SEARCHFILE_TOTAL	300
SEARCHKEYWORD_TOTAL	300
SEARCHNOTES_TOTAL	50
SEARCHSTOREFILE_TOTAL	10
SEARCHSTOREKEYWORD_TOTAL	10
SEARCHSTORENOTES_TOTAL	10
SEARCHNODECOMP_TOTAL	10
SEARCHFINDBUDDY_TOTAL	10
SEARCHFINDSOURCE_TOTAL	20

Table A.2.: The kademlia variables

B. Kad opcodes

Identifier	Opcode	Parameter
KADEMLIA_BOOTSTRAP_REQ	0x00	$< PEER(sender)[25] >$
KADEMLIA_BOOTSTRAP_RES	0x08	$< CNT[2] >< PEER[25] > *(CNT)$
KADEMLIA_HELLO_REQ	0x10	$< PEER(sender)[25] >$
KADEMLIA_HELLO_RES	0x18	$< PEER(receiver)[25] >$
KADEMLIA_REQ	0x20	$< TYPE[1] >< HASH(target)[16] >$ $< HASH(receiver)16 >$
KADEMLIA_RES	0x28	$< HASH(target)[16] >< CNT >$ $< PEER[25] > *(CNT)$
KADEMLIA_SEARCH_REQ	0x30	$< HASH(key)[16] >< ext0/1[1] >$ $< SEARCHTREE > [ext]$
KADEMLIA_SEARCH_RES	0x38	$< HASH(key)[16] >< CNT1[2] >$ $(< HASH(answer)[16] < CNT2[2] >>$ $< META > *(CNT2)) * (CNT1)$
KADEMLIA_SRC_NOTES_REQ	0x32	$< HASH(key)[16] >$
KADEMLIA_SRC_NOTES_RES	0x3A	$< HASH(key)[16] >< CNT1[2] >$ $< (< HASH(answer)[16] >< CNT2[2] >$ $<< META > *(CNT2)) * (CNT1)$
KADEMLIA_PUBLISH_REQ	0x40	$< HASH(key)[16] >< CNT1[2] >$ $(< HASH(target)[16] >< CNT2[2] >$ $< META > *(CNT2)) * (CNT1)$
KADEMLIA_PUBLISH_RES	0x48	$< HASH(key)[16] >$
KADEMLIA_PUB_NOTES_REQ	0x42	$< HASH(key)[16] >$ $< HASH(target)[16] >< CNT2[2] >$ $< META > *(CNT2)) * (CNT1)$
KADEMLIA_PUB_NOTES_RES	0x4A	$< HASH(key)[16] >$
KADEMLIA_FIREWALLED_REQ	0x50	$< TCPPORT(sender)[2] >$
KADEMLIA_FINDBUDDY_REQ	0x51	$< TCPPORT(sender)[2] >$
KADEMLIA_CALLBACK_REQ	0x52	$< TCPPORT(sender)[2] >$
KADEMLIA_FIREWALLED_RES	0x58	$< IP(sender)[4] >$
KADEMLIA_FIREWALLED_ACK	0x59	$(null)$
KADEMLIA_FINDBUDDY_RES	0x5A	$< TCPPORT(sender)[2] >$
KADEMLIA_FIND_VALUE	0x02	
KADEMLIA_STORE	0x04	
KADEMLIA_FIND_NODE	0x0B	

Table B.1.: The opcodes of the Kad-protocol

C. Extract of a routing table

A for the investigation modified version of aMule made an extract of an routing table with 510 contacts. It illustrates all containing nodes of a binary tree which are internal nodes or leafs with the quantity of their containing contacts. Each node or leave has a unique position in the tree which is indicated by the level (L) and the position (Pos).

```
ShowRoutingTable!!!
Node        L: 0       Pos.: 00000000000000000000000000000000
Node        L: 1       Pos.: 00000000000000000000000000000000
Node        L: 2       Pos.: 00000000000000000000000000000000
Node        L: 3       Pos.: 00000000000000000000000000000000
Node        L: 4       Pos.: 00000000000000000000000000000000
Node        L: 5       Pos.: 00000000000000000000000000000000
Node        L: 6       Pos.: 00000000000000000000000000000000
Node        L: 7       Pos.: 00000000000000000000000000000000
Node        L: 8       Pos.: 00000000000000000000000000000000
Node        L: 9       Pos.: 00000000000000000000000000000000
Node        L: 10       Pos.: 00000000000000000000000000000000
Node        L: 11       Pos.: 00000000000000000000000000000000
Node        L: 12       Pos.: 00000000000000000000000000000000
Node        L: 13       Pos.: 00000000000000000000000000000000
Node        L: 14       Pos.: 00000000000000000000000000000000
Node        L: 15       Pos.: 00000000000000000000000000000000
Node        L: 16       Pos.: 00000000000000000000000000000000
Node        L: 17       Pos.: 00000000000000000000000000000000
Node        L: 18       Pos.: 00000000000000000000000000000000
Node        L: 19       Pos.: 00000000000000000000000000000000
Node        L: 20       Pos.: 00000000000000000000000000000000
Leaf   BinSize: 2   L: 21       Pos.  : 00000000000000000000000000000000
Leaf   BinSize: 8   L: 21       Pos.  : 00000000000000000000000000000001
Leaf   BinSize: 0   L: 20       Pos.  : 00000000000000000000000000000001
Leaf   BinSize: 2   L: 19       Pos.  : 00000000000000000000000000000001
Node        L: 18       Pos.: 00000000000000000000000000000001
Node        L: 19       Pos.: 00000000000000000000000000000002
Node        L: 20       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 10   L: 21       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 0   L: 21       Pos.  : 00000000000000000000000000000009
Leaf   BinSize: 10   L: 20       Pos.  : 00000000000000000000000000000005
Leaf   BinSize: 2   L: 19       Pos.  : 00000000000000000000000000000003
Node        L: 17       Pos.: 00000000000000000000000000000001
```

```
Leaf   BinSize: 3   L: 18       Pos.  : 00000000000000000000000000000002
Leaf   BinSize: 9   L: 18       Pos.  : 00000000000000000000000000000003
Node        L: 16       Pos.: 00000000000000000000000000000001
Leaf   BinSize: 5   L: 17       Pos.  : 00000000000000000000000000000002
Node        L: 17       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 10   L: 18       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 18       Pos.  : 00000000000000000000000000000007
Leaf   BinSize: 4   L: 15       Pos.  : 00000000000000000000000000000001
Node        L: 14       Pos.: 00000000000000000000000000000001
Node        L: 15       Pos.: 00000000000000000000000000000002
Leaf   BinSize: 10   L: 16       Pos.  : 00000000000000000000000000000004
Leaf   BinSize: 10   L: 16       Pos.  : 00000000000000000000000000000005
Node        L: 15       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 7   L: 16       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 16       Pos.  : 00000000000000000000000000000007
Node        L: 13       Pos.: 00000000000000000000000000000001
Node        L: 14       Pos.: 00000000000000000000000000000002
Leaf   BinSize: 4   L: 15       Pos.  : 00000000000000000000000000000004
Leaf   BinSize: 10   L: 15       Pos.  : 00000000000000000000000000000005
Node        L: 14       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 7   L: 15       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 15       Pos.  : 00000000000000000000000000000007
Leaf   BinSize: 5   L: 12       Pos.  : 00000000000000000000000000000001
Node        L: 11       Pos.: 00000000000000000000000000000001
Node        L: 12       Pos.: 00000000000000000000000000000002
Node        L: 13       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 4   L: 14       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 10   L: 14       Pos.  : 00000000000000000000000000000009
Leaf   BinSize: 4   L: 13       Pos.  : 00000000000000000000000000000005
Leaf   BinSize: 2   L: 12       Pos.  : 00000000000000000000000000000003
Node        L: 10       Pos.: 00000000000000000000000000000001
Leaf   BinSize: 2   L: 11       Pos.  : 00000000000000000000000000000002
Node        L: 11       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 10   L: 12       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 12       Pos.  : 00000000000000000000000000000007
Leaf   BinSize: 1   L: 9       Pos.  : 00000000000000000000000000000001
Node        L: 8       Pos.: 00000000000000000000000000000001
Node        L: 9       Pos.: 00000000000000000000000000000002
Node        L: 10       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 10   L: 11       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 10   L: 11       Pos.  : 00000000000000000000000000000009
Leaf   BinSize: 10   L: 10       Pos.  : 00000000000000000000000000000005
Node        L: 9       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 10   L: 10       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 10       Pos.  : 00000000000000000000000000000007
Node        L: 7       Pos.: 00000000000000000000000000000001
```

```
Node        L: 8       Pos.: 00000000000000000000000000000002
Node        L: 9       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 10   L: 10       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 10   L: 10       Pos.  : 00000000000000000000000000000009
Leaf   BinSize: 10   L: 9       Pos.  : 00000000000000000000000000000005
Node        L: 8       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 10   L: 9       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 9       Pos.  : 00000000000000000000000000000007
Node        L: 6       Pos.: 00000000000000000000000000000001
Leaf   BinSize: 5   L: 7       Pos.  : 00000000000000000000000000000002
Node        L: 7       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 10   L: 8       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 8   L: 8       Pos.  : 00000000000000000000000000000007
Node        L: 5       Pos.: 00000000000000000000000000000001
Leaf   BinSize: 1   L: 6       Pos.  : 00000000000000000000000000000002
Node        L: 6       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 10   L: 7       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 3   L: 7       Pos.  : 00000000000000000000000000000007
Node        L: 4       Pos.: 00000000000000000000000000000001
Node        L: 5       Pos.: 00000000000000000000000000000002
Node        L: 6       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 10   L: 7       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 3   L: 7       Pos.  : 00000000000000000000000000000009
Leaf   BinSize: 10   L: 6       Pos.  : 00000000000000000000000000000005
Node        L: 5       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 10   L: 6       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 6       Pos.  : 00000000000000000000000000000007
Node        L: 3       Pos.: 00000000000000000000000000000001
Node        L: 4       Pos.: 00000000000000000000000000000002
Node        L: 5       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 9   L: 6       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 10   L: 6       Pos.  : 00000000000000000000000000000009
Leaf   BinSize: 9   L: 5       Pos.  : 00000000000000000000000000000005
Node        L: 4       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 6   L: 5       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 6   L: 5       Pos.  : 00000000000000000000000000000007
Node        L: 2       Pos.: 00000000000000000000000000000001
Node        L: 3       Pos.: 00000000000000000000000000000002
Node        L: 4       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 10   L: 5       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 10   L: 5       Pos.  : 00000000000000000000000000000009
Leaf   BinSize: 10   L: 4       Pos.  : 00000000000000000000000000000005
Node        L: 3       Pos.: 00000000000000000000000000000003
Leaf   BinSize: 6   L: 4       Pos.  : 00000000000000000000000000000006
Leaf   BinSize: 10   L: 4       Pos.  : 00000000000000000000000000000007
Node        L: 1       Pos.: 00000000000000000000000000000001
```

```
Node        L: 2       Pos.: 00000000000000000000000000000002
Node        L: 3       Pos.: 00000000000000000000000000000004
Leaf   BinSize: 10   L: 4       Pos.  : 00000000000000000000000000000008
Leaf   BinSize: 10   L: 4       Pos.  : 00000000000000000000000000000009
Node        L: 3       Pos.: 00000000000000000000000000000005
Leaf   BinSize: 10   L: 4       Pos.  : 0000000000000000000000000000000A
Leaf   BinSize: 4   L: 4       Pos.  : 0000000000000000000000000000000B
Node        L: 2       Pos.: 00000000000000000000000000000003
Node        L: 3       Pos.: 00000000000000000000000000000006
Leaf   BinSize: 10   L: 4       Pos.  : 0000000000000000000000000000000C
Leaf   BinSize: 10   L: 4       Pos.  : 0000000000000000000000000000000D
Node        L: 3       Pos.: 00000000000000000000000000000007
Leaf   BinSize: 6   L: 4       Pos.  : 0000000000000000000000000000000E
Leaf   BinSize: 10   L: 4       Pos.  : 0000000000000000000000000000000F
```

D. Database structure

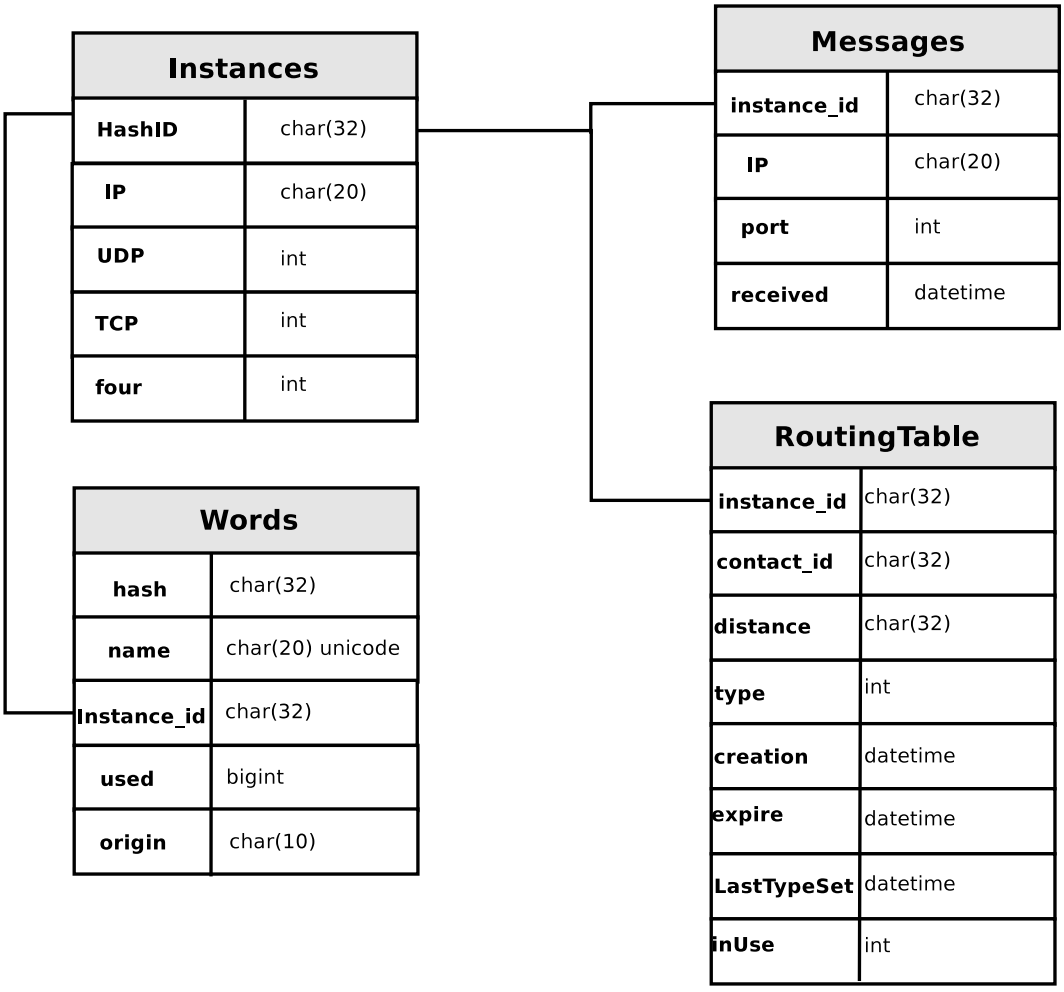

Figure D.1.: The helper tables and the tables for the contact lifecycle

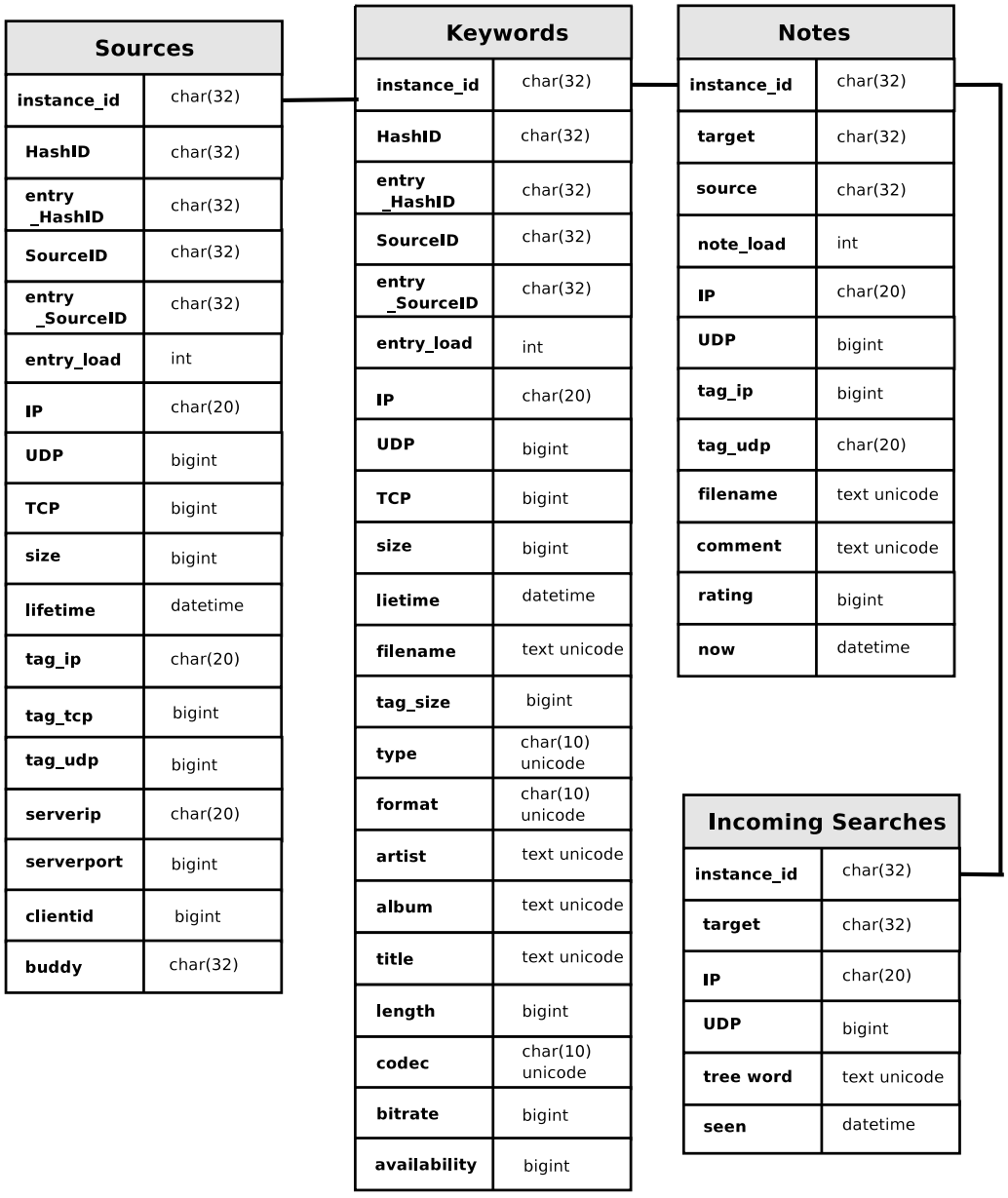

Figure D.2.: Database tables for the passive meassurement

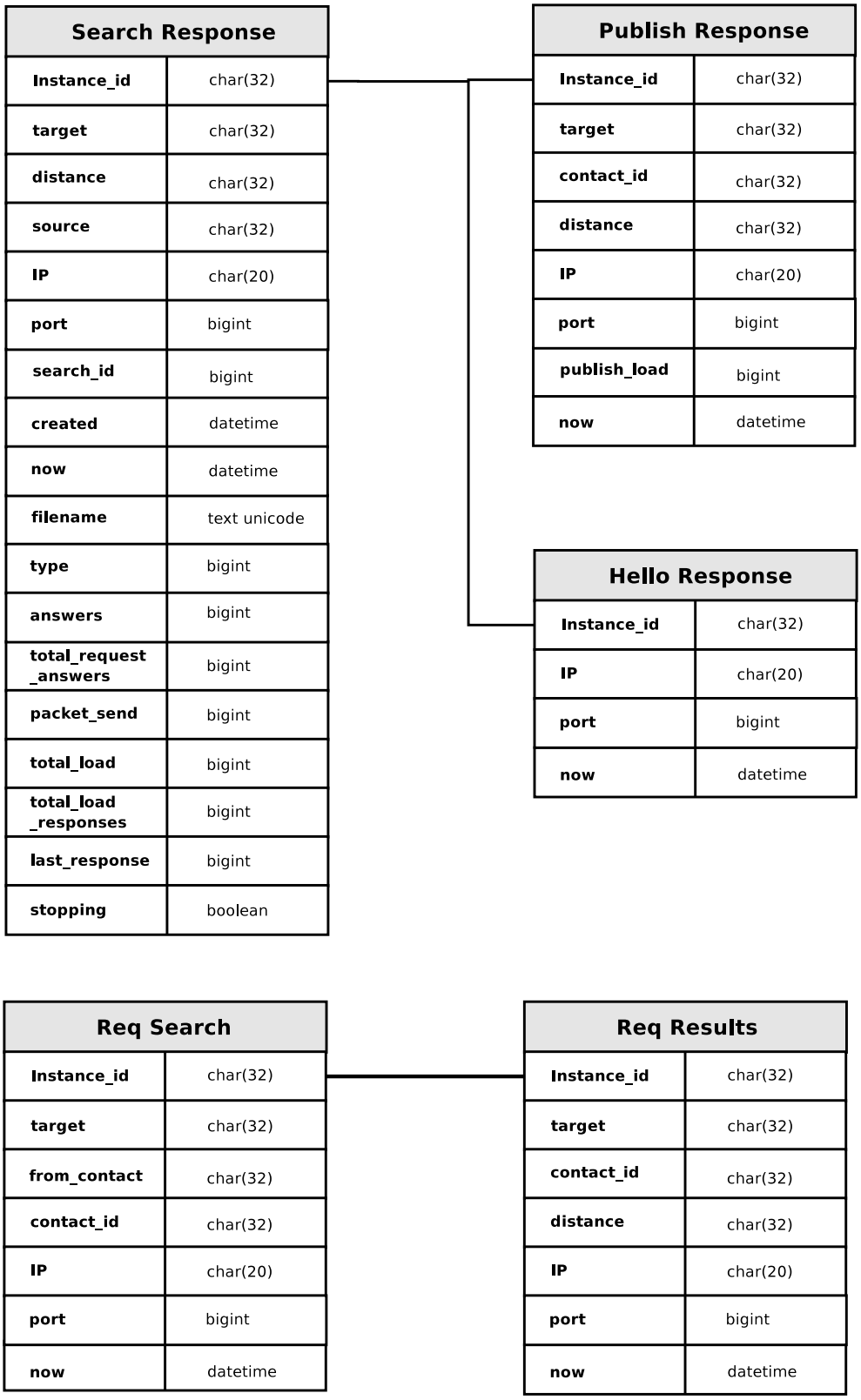

Figure D.3.: Database tables for the passive meassurement

Bibliography

[1] aMule. http://www.amule.org/wiki/. 2006.

[2] aMule Forum. http://forum.amule.org/thread.php?threadid=10168. 2006.

[3] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS-03)*, pages 256–267, 2003.

[4] BitTorrent. http://www.bittorrent.com/. 2006.

[5] CacheLogic. Peer-to-peer in 2005, 2005. http://www.cachelogic.com/home/pages/research/p2p2005.php.

[6] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. In *2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, 2005.

[7] David D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM*, pages 106–114, Stanford, CA, August 1988. ACM.

[8] Internet Movie Database. http://www.imdb.com/. 2006.

[9] D. Dumitriu, E. Knightly, A. Kuzmanovic, I. Stoica, and W. Zwaenepoel. Denialofservice resilience in peertopeer file sharing systems. In *Proceedings of SIGMETRICS 2005*, 2005.

[10] Edonkey2000. http://www.edonkey2000.com/. 2006.

[11] eMule. http://www.emule-project.net. 2006.

[12] Anh-Tuan Gai and L. Viennot. Optimizing and balancing load in fully distributed p2p file sharing systems. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 111–111, 2006.

[13] Luis Garcés-Erice, Keith W. Ross, Ernst W. Biersack, Pascal A. Felber, and Guillaume Urvoy-Keller. TOpology-centric Look-Up Service. In *Proceedings of COST264/ACM 5th International Workshop on Networked Group Communications (NGC)*, LNCS, pages 58–69, Munich, Germany, September 2003. Springer.

[14] GNU. http://www.gnu.org/licenses/. 2006.

[15] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329, New York, NY, USA, 2003. ACM Press.

[16] Oliver Heckmann and Axel Bock. The eDonkey 2000 Protocol. Technical Report KOM-TR-08-2002, Multimedia Communications Lab, Darmstadt University of Technology, December 2002.

[17] InnoDB. http://www.innodb.com/. 2006.

[18] Mikel Izal, Guillaume Urvoy-Keller, Ernst W Biersack, Pascal A Felber, Anwar Al Hamra, and Luis Garces-Erice. Dissecting bittorrent: Five months in a torrent's lifetime. In *Proceedings of Passive and Active Measurements (PAM)*, 2004,.

[19] Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and Kc claffy. Transport layer identification of p2p traffic. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134, New York, NY, USA, 2004. ACM Press.

[20] Patrick Kirk. http://rfc-gnutella.sourceforge.net/index.html. 2003.

[21] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. A statistical theory of chord under churn. In *Proceedings of The 4th Annual International Workshop on Peer-To-Peer Systems (IPTPS 05)*, Ithaca, NY, USA, Jan 2005.

[22] Yoram Kulbak and Danny Bickson. The emule protocol specification. Technical report, School of Computer Science and Engineering, Hebrew University of Jerusalem, January 2005.

[23] Kendy Kutzner and Thomas Fuhrmann. Measuring large overlay networks - the overnet example. In *Konferenzband der 14. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005)*, Kaiserslautern, Germany, 2005.

[24] Ben Leong, Barbara Liskov, and Eric D. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. In *Proceedings of the 12th International Conference on Networks 2004 (ICON 2004), Singapore*, November 2004.

[25] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and M. Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, February 2004.

[26] Jinyang Li, Jeremy Stribling, Robert Morris, Frans M. Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating dht design tradeoffs under churn. In *Proceedings of the 24th Infocom*, Miami, FL, 2005.

[27] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 1–11, Boston, Massachusetts, May 2005.

[28] Jian Liang, Rakesh Kumar, Yongjian Xi, and Keith W. Ross. Pollution in p2p file sharing systems. In *Proceedings of IEEE INFOCOM 2005*, 2005.

[29] Jian Liang, Naoum Naoumov, and Keith W. Ross. The index poisoning attack in p2p file sharing systems. In *Proceedings of IEEE INFOCOM 2006*, 2006.

[30] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 233–242, Monterey, CA, July 2002.

[31] Skype Limited. P2P telephony explained, 2006. http://www.skype.com/download/explained.html.

[32] Eng K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. 7(2):72–93, 2005.

[33] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, February 2003.

[34] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[35] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-peer informatic system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65, Cambridge, MA, USA, March 2002. Springer.

[36] Enzo Michelangeli and Arto Jalkanen. Kadc, 2006. http://kadc.sourceforge.net/.

[37] Alper Tugay Mizrak, Yuchung Cheng, Vineet Kumar, and Stefan Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *Proceedings of the Third IEEE Workshop on Internet Applications (WIAPP'03)*, pages 104–111, San Jose, California, 2003.

[38] MLdonkey. http://mldonkey.org/. 2006.

[39] MySQL. http://mysql.com/. 2006.

[40] MySQL++. http://tangentsoft.net/mysql++/. 2006.

[41] J. Chuang N. Christin, A. Weigend. Content availability, pollution and poisoning in peer-to-peer file sharing networks. In *Proc. ACM E-Commerce Conference (EC'05)*, June 2005.

[42] Nodes.dat. http://www.overnet2000.de/html/index.php?module =contentexpress&func=display&ceid=3. 2006.

[43] Andrew Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[44] Overnet. http://www.overnet.com/. 2006.

[45] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The Bittorrent P2P file-sharing system: Measurements and analysis. In *4th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, volume 3640. LNCS, Feb 2005.

[46] Yi Qiao and Fabián E. Bustamante. Structured and unstructured overlays under the microscope. In *USENIX Annual Technical Confrence*, 2006.

[47] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[48] RevConnect. http://www.revconnect.com/. 2006.

[49] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *In Proceedings of USENIX Annual Technical Conference*, 2004.

[50] Ronald L. Rivest. The md4 message digest algorithm. In *Advances in Cryptology - Crypto '90*, pages 303–311, 1991.

[51] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale Peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, LNCS, pages 329–350, Heidelberg, Germany, November 2001. Springer.

[52] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 137–150, New York, NY, USA, 2002. ACM Press.

[53] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Second Annual ACM Internet Measurement Workshop*, November 2002.

[54] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, Frans F. Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.

[55] Daniel Stutzbach and Reza Rejaie. Improving lookup performance over a widely-deployed dht. In *Proceedings of IEEE INFOCOM 2006*, May 2006.

[56] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM Press.

[57] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Should internet service providers fear peer-assisted content distribution? In *Internet Measurement Conference (IMC '05)*, pages 49–62, 2005.

[58] Chunqiang Tang, Melissa J. Buco, Rong N. Chang, Sandhya Dwarkadas, Laura Z. Luan, Edward So, and Christopher Ward. Low traffic overlay networks with large routing tables. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 14–25, New York, NY, USA, 2005. ACM Press.

[59] wxWidgets. http://www.wxwidgets.org/. 2006.

[60] Member-Li Xiao, Member-Yunhao Liu, and Fellow-Lionel M. Ni. Improving unstructured peer-to-peer systems by adaptive connection establishment. *IEEE Trans. Comput.*, 54(9):1091–1103, 2005.

[61] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr 2001.