Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Implementation of the Kademlia Distributed Hash Table

Semester Thesis Written by

Bruno Spori

Summer Term 2006
SA-2006-19

Advisor: Marcel Baur
Professor: Prof. Dr. Bernhard Plattner

Abstract

This report describes an implementation of the Kademlia DHT written in Java. The intended purpose of the implementation is to become an alternative to the existing DHT in the Celeste automatic storage system. Important design characteristics are the multi-threaded architecture, the representation of messages as usual Java objects which can be converted to `byte[]`s according to a well-defined format and the introduction of buffers for the temporary storage of messages. To ensure the proper functionality of the program, several kind of tests were performed. There remains some future work to be done connected with the implementation, most notably the actual integration into Celeste.

Zusammenfassung

Dieser Bericht beschreibt die Implementation eines Kademlia DHTs, geschrieben in Java. Die Implementation soll später einmal als Alternative zu der bestehenden DHT im Celeste Speicherungssystem eingesetzt werden können. Wichtige Design Merkmale sind die mulit-threaded Architektur, die Repräsentation der Messages als gewöhnliche Java-Objekte, die einem klar definierten Format folgend in `byte[]`s umgewandelt werden können und die Einführung von Buffern um Messages zwischenzuspeichern. Um die Korrektheit des Programmes sicherzustellen wurden verschiedene Arten von Tests durchgeführt. Im Zusammenhang mit der Implementation sind noch ein paar Arbeiten ausstehend, vor allem die eigentliche Integration in Celeste steht noch bevor.

Contents

1 Introduction **6**

1.1 Distributed Hash Table 6

1.1.1 Hash Table 6

1.1.2 Distributed Hash Table 6

1.2 Celeste 7

1.2.1 The Problem of Managing Storage 7

1.2.2 System Concept 7

1.2.3 The Need for a DHT 8

1.3 Motivation and Goal 8

2 Kademlia DHT **9**

2.1 XOR-Topology 9

2.2 k-Buckets 9

2.3 The Node Lookup Procedure 10

2.4 Bootstrapping 11

2.5 Storing and Finding of <key,value>-Pairs 11

2.6 Republishing and Other Recurring Tasks 12

3 Implementation **13**

3.1 General Architecture 13

3.1.1 Representation of the Messages 13

3.1.2 The Pinging-Concept 15

3.1.3 Separate Threads for Receiving Messages and Answering Requests 15

3.1.4 Buffers for Temporary Storage of Received Messages 17

3.1.5 Implementation of the Node Lookup Function 17

3.1.6 Priority Queue For All Periodic Tasks 19

3.2 Messages 19

3.2.1 Hierarchical Structure of the Different Messages 19

3.2.2 Ping Request and Ping Reply Message 19

3.2.3 Reasons For Introducing the PingRepReq-Message 20

3.2.4 The FindNode- and FindValue-Message 21

3.2.5 The Store-Message 21

3.2.6 The NodeReply- and ValueReply-Messages 22

3.3 Repositories 22

3.3.1 k-Buckets 22

3.3.2 KeyValueContainer 24

3.3.3 PublishedValuesContainer 26

3.4 Buffers 27

3.4.1 PingReplyBuffer 27

3.4.2 NodeReplyBuffer 28

3.4.3 ValueReplyBuffer 29

3.5 Threads 30

3.5.1 The Receiving Thread 30

3.5.2 The Replying Thread . . . 30
3.5.3 The Refresher Thread . . . 30

4 Testing **32**

5 Future Work **33**

6 Conclusions **34**

Bibliography **35**

List of Figures

2.1 Binary tree representing the complete 4 bit address space. 10

3.1 General Structure of the messages on the `byte[]`-level 14
3.2 General course of a message exchange 15
3.3 The structure of a `PingRequestMessage` (top) and a `PingReplyMessage` (bottom). 20
3.4 Extraction of ping replies to ensure validity of received messages. 20
3.5 The structure of the `PingRepReqMessage`. 21
3.6 Course of the message exchange triggered by an explicit ping request 21
3.7 The structrue of the `FindNodeMessage` and `FindValueMessage`. 21
3.8 The structure of a `StoreMessage`. 22
3.9 The structure of a `NodeReplyMessage`. 22
3.10 The structure of `ValueReplyMessage`. 22
3.11 Adding the node to the k-buckets to which a STORE_RPC was sent results in a loop. 26

List of Tables

3.1 Two 16 bit ids and the distance between them 24

Chapter 1

Introduction

This report describes an implementation of the Kademlia Distributed Hash Table (DHT). The implementation was performed in a semester thesis at the Computer Engineering and Networks Laboratory at ETH Zurich.

The aim of this first chapter is to explain what a DHT is and to introduce the Celeste system, for which the Kademlia DHT has been implemented. Thereafter, the motivation and the goal of the semester thesis can be stated.

1.1 Distributed Hash Table

Kademlia is a Distributed Hash Table (DHT). To understand the functioning of Kademlia, it is thus vital to know what a DHT is. The following section describes first what a hash table is in general and then introduces the DHT, together with its most important properties.

1.1.1 Hash Table

A hash table is a data structure that provides a mapping from keys to values. That is, through specifying the key, the hash table returns the value. To achieve this mapping, a hash function is needed. The hash function calculates out of the key the address of the memory cell where the value shall be stored. A hash function should have certain properties: First, it shall avoid collisions. That is, the mapping from the key space to the address space must be even and random as much as possible. Second, the hash function shall be easy to calculate. This second requirement is because of performance reasons.

1.1.2 Distributed Hash Table

As the name suggests, a DHT is a hash table implemented as distributed system. In the case of Kademlia, it is a peer-to-peer overlay network. Peer-to-peer means that no central server exists where all connections go through. Instead, nodes communicate directly with each other. In an overlay network, each node keeps a list of contacts participating in the network. The contacts for the list are chosen in a structured way such that the desired topology of the network is achieved.

Similar to a hash table, a DHT provides a mapping from keys to values. As opposed to a generic hash table, the values in a DHT are stored on the different nodes in the network and not in a local data structure. Through specifying the key, the value can be stored at an appropriate node or retrieved from the network. In a DHT, the key is calculated from the value. All keys are part of the same address space. For example, the 160 bit address space has become popular in the context of file sharing. To determine on which nodes which values shall be stored, each node must have an unique identifier out of the same address space like the keys. Additionally, a notion of distance between two ids must be introduced (a key can be considered as the id of the value). The rule is then to store the value on that node, whose id has the lowest distance to the key.

To store a value in the network, a node sends a store message for that value to an appropriate contact. This is the contact chosen from the routing table which has the lowest distance to the key. The message will be forwarded in the network from node to node[1] until it reaches the closest possible node. There, the value is stored. To find a value, the procedure is the same, except that the node sends a find message and that the node which has stored the value returns it.

An efficient DHT must fulfill several requirements:

- The key space must be evenly distributed to the nodes.
- When a node joins or leaves the network, the address space must be redistributed as well as the stored values in the whole network.
- It must be self-organizing.
- Because nodes may fail, there must be redundancy in the network. Thus, values are not stored on only one node, but on a number specified by the respective application.
- It must be scalable. That is, a very large number of nodes must be supported.

1.2 Celeste

As mentioned already in the introduction, the implementation of Kademlia described in this report is intended to become a part of the Celeste system. In this section, an overview of Celeste is given. For a more detailed description of Celeste, see [1].

1.2.1 The Problem of Managing Storage

The use of computing systems is still growing. This can be in areas where until recently the tasks had been accomplished without or very limited aid of computers. Or it can be in areas where the use of computers has enabled new possibilities. In all cases, the consequence is the growing need of storage capacity for the newly generated data.

Satisfying these needs is not a problem. Every generation of hard drives offers higher storage capacities to the same or even lower prices than the preceding generation. The result is that the price per gigabyte continuously drops. But increasing storage capacities increase the management needs for the storage as well. In contrast to the price for the storage itself, the costs for storage management are very high. An example, taken from [1], illustrates this circumstance very well: Assuming the current price of approximately 1$ per gigabyte and administrative annual costs of 100K$ per year for an employee, then one employee is equivalent to 100TB of storage every year!

Thus, the need to reduce those management costs is obvious. Celeste proposes a solution to this problem through providing an automatic storage system. That is, users of Celeste can just put their data into Celeste without having to care about the management or even how the system works internally.

1.2.2 System Concept

To achieve this goal, Celeste is based on a distributed system composed of a set of member nodes. Each node contributes to the system through providing storage and computing capacity. Data stored into Celeste are distributed among the nodes. Certain policies to adjust trade-offs between capacity, performance and reliability are provided, but all other management tasks are performed by the nodes themselves.

The nodes can be connected to each other by the Internet. If implemented like this, Celeste can be used as a Public Utility Computing System (PUC). This means that some company maintains a Celeste system to which other companies are connected. The other companies will have to pay for the storage capacity they get, but they are freed from any burdens of storage management. As

[1] Kademlia uses another approach, as explained in section 2.3.

Celeste provides strong security measures, even competing companies could use the same Celeste system.

1.2.3 The Need for a DHT

What Celeste effectively implements is Distributed Object Location and Retrieval (DOLR). That is, a data object provided to Celeste is distributed among the nodes in the system. Based on a policy, it is decided on which nodes certain data objects are stored. Conversely, to retrieve a data object out of Celeste, it must effectively be found in the system and transfered to the place where it is requested. To implement these two functionalities, Celeste needs a DHT.

1.3 Motivation and Goal

Up to now, Celeste uses a DHT similar to "Tapestry" [2], developed by UC Berkeley. The implementation in Celeste is based on TCP-connections between the nodes. This involves some drawbacks regarding performance. To establish a connection, TCP needs an explicit connection establishment phase, during which no user data can be transmitted. Furthermore, each TCP-connection has a state. If a node has connections to many other nodes, it must establish a TCP-connection to all of them. For performance reasons, a cache had to be implemented.

It would be beneficial if both, the long connection establishment phase and the cache for TCP-connections could be avoided, because they signify overhead. Thus, another DHT should be implemented for Celeste.

The choice was made for the Kademlia DHT, because of the following reasons:

- Kademlia is based on UDP and thus does not have the disadvantages stated above. To deal with the unreliability of UDP, Kademlia makes use of concurrent requests to several nodes (see section 2.3).
- Kademlia is very popular at the moment. Many well known file sharing clients such as "eMule" [3] are based at least in part on Kademlia.
- Kademlia uses a new approach for locating contacts in the network. To find an id, a node iteratively queries other nodes until it has reached it. Most of other DHTs route query messages from node to node (see section 2.3).

The goal of this semester thesis was thus to implement a Kademlia DHT for the Celeste system. As Celeste is written in Java, a constraint was that the implementation has to be in Java as well. The implementation is based on [4].

Chapter 2

Kademlia DHT

In this chapter, the functioning of the Kademlia DHT is explained. The description follows [4].

2.1 XOR-Topology

In Kademlia each node and each key has a 160 bit id. A key in Kademlia is the hash value of a data object (for example a file). The value corresponding to a key is a pointer to the host that has stored the data object. Specifically, a value consists of the IP-address and the port number where the host storing the data object can be contacted.

The distance between two ids is defined as their bitwise XOR. That is, if x and y are two ids, then $d(x, y) = x \oplus y$. Note the following properties of XOR: $d(x, x) = 0$, $d(x, y) > 0$ if $x \neq y$, $\forall x, y : d(x, y) = d(y, x)$ (thus, d is *symmetric*) and the triangle propery: $d(x, y) + d(y, z) \geq d(x, z)$. The symmetry of the distance is an important advantage of the XOR-metric. It enables Kademlia to learn contact information from ordinary queries it receives. Furthermore, XOR is unidirectional. That is, for a given distance Δ and an id x, there exists exactly one id y sucht that $d(x, y) = \Delta$. This property ensures that lookups for a certain id converge all along the same path, regardless of the originating node. Kademlia profits from this property through caching of values along the path to the hosting node. Lookups for an id are then likely to hit a cached entry.

2.2 k-Buckets

The routing table of Kademlia consists of k-buckets. A k-bucket is a list to store contact information. "k" designates the capacity of the buckets, the authors of [4] propose a value of 20. There is a k-bucket for every $0 \leq i < 160$, where i is the index of the k-bucket. Obviously, there are in total 160 k-buckets. A k-bucket with index i stores contacts whose ids have a distance between 2^i and 2^{i+1} to the own id. Thus the 160 k-buckets cover the whole 160 bit address space. It is clear that nodes with a small distance are put in a k-bucket with small index and that for contacts with small distances more space is reserved in the k-buckets. Another way of defining the index of a k-bucket is to say that a node with distance d will be put in the k-bucket with index $i = \lfloor \log d \rfloor$.

Upon reception of any message, the sender's contact information is tried to be added to the appropriate k-bucket. Adding a node to a k-bucket includes the following steps: If the corresponding k-bucket stores less than k contacts and the new node is not already contained, the new node is added at the tail of the list. If the k-bucket contains the contact already, it is moved to the tail of the list. Should the appropriate k-bucket be full, then the contact at the head of the list is pinged. If it replies, then it is moved to the tail of the list and the new contact is not added. If it does not, the old contact is discarded and the new contact is added at the tail.

The description above makes clear that the k-buckets use a least-recently-seen eviction policy, with the exception that nodes still participating in the network are never removed. There are two reasons why old contacts are preferred: First, it was shown empirically [5] that the longer a node

has been up in a peer-to-peer network the more likely it will remain online another hour. Second, it is not possible to flush a node's routing table through sending of faked messages. This provides some resistance against denial of service attacks.

2.3 The Node Lookup Procedure

The node lookup is the central function of Kademlia. As subsequent sections will reveal, most procedures of Kademlia are based on the node lookup. The task of the node lookup is to find the k closest nodes in the network to the id which is looked up.

The course of operations in a node lookup are as follows: The node that performs the lookup must first get α nodes out of its k-bucket where the id to be looked up would fit in. α is a system-wide concurrency parameter, the authors of [4] suggest a value of 3. If the corresponding k-bucket has less than α entries, the node takes the α closest nodes it knows of. Once the node has located those α entries, it sends FIND_NODE-RPCs to them. A node receiving a FIND_NODE-RPC must return the k closest nodes to the id contained in the RPC-message it has stored in its k-buckets.

After the responses of the queried nodes have been received, the nodes contained in the reply messages are inserted in a list. This list is sorted according to the distance between the new nodes and the looked up id. Depending on whether the last round of FIND_NODE-RPCs revealed a new closest node or not, different actions are taken. If it has, then the node chooses α nodes among the k closest already seen but not already queried and sends them FIND_NODE-RPCs. Otherwise, it picks all among the k closest it has not already queried and sends them FIND_NODE-RPCs. The node lookup is terminated when the node has queried all of the k closest nodes it has discovered and gotten responses from them.

In order to speed up the node lookup when a node does not answer, the looking up node can begin a new round of FIND_NODE-RPCs before all of the α nodes of the last round answered. When an answer was only delayed, then it shall be considered nevertheless.

The node lookup can also be illustrated graphically. Figure 2.1 shows the complete 4 bit address space (the 160 bit address space used by Kademlia would have been far too large for the drawing).

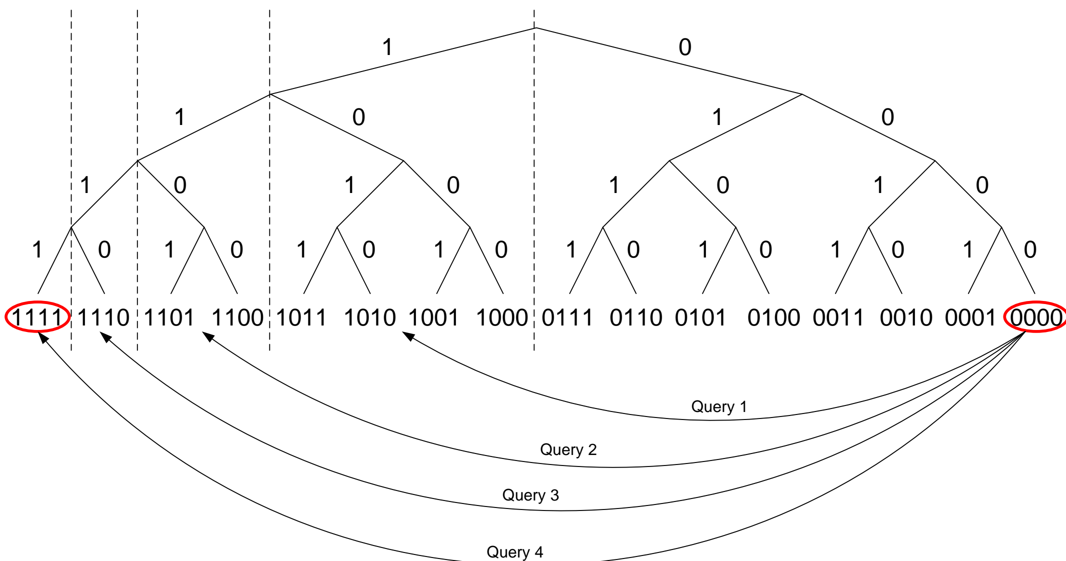

Figure 2.1: Binary tree representing the complete 4 bit address space. The node with id "0000" performs a node lookup for id "1111".

Assume for example that the node with id "0000" performs a lookup for id "1111". As stated

above, this node must first retrieve the α closest nodes out of its k-buckets. Notice that in figure 2.1 α=1 was chosen in order to simplify the drawing. In the corresponding k-bucket, all of the ids beginning with a "1" can be stored, because $\lfloor \log d \rfloor$ yields the same result for all of them (cf. section 2.2). In the worst case, the node contains in this k-bucket only contacts of the subtree "10". This is the worst case because now the first FIND_NODE-RPC will only reach a contact in this subtree and thus approach the target-id with the least far reaching possible step.

The contacts receiving the RPC will return their k closest nodes they know of. Constructing again the worst case, they have only contacts in the subtree "110", which they return to "0000". In the next step, "0000" will contact nodes out of this subtree. This continues until in the fourth step it has found "1111".

This example shall clarify that in the worst case a node lookup can take up to 160 steps in the 160 bit address space of Kademlia. Fortunately, it is highly unlikely that this happens. In [4], it is proved that with high probability a node lookup terminates in $\lfloor \log n \rfloor + c$, where n is the number of nodes in the network and c a small constant.

Another interesting fact is that the node lookup works iteratively. That is, queries are always returned to the node performing the node lookup. This node will in turn contact further nodes in subsequent rounds. Many other DHTs follow a different concept, where the messages are routed directly from subtree to subtree.

2.4 Bootstrapping

A node that wants to join the network must either know a gateway or it has to become a gateway itself. In the latter case, the node will be the first and only node in the network and thus has to wait until other nodes join before it can perform any meaningful operations. In the case where a gateway is known, the node does the following:

1. The gateway is inserted in the appropriate k-bucket.

2. A node lookup for the own id is performed. Of course, the only node that will be contacted initially is the gateway. Through the node lookup for the own id, the node gets to know its closest neighbors.

3. Node lookups in the range for all k-buckets with a higher index than the one of the lowest non-empty are performed. This fills the k-buckets of the joining node as well as communicates the arrival of the new node to the existing nodes. Notice that node lookups for k-buckets with index lower than the first non-empty would be useless, as there are no appropriate contacts in the network (otherwise, the lookup for the own id would have revealed them).

2.5 Storing and Finding of <key,value>-Pairs

For both to store and to find a <key,value>-pair, a node lookup must performed. If a <key,value>-pair shall be stored in the network, a node lookup for the key is conducted. Thereafter, STORE-RPCs are sent to all of the k nodes the node lookup has returned. A STORE-RPC instructs a node to store the <key,value>-pair contained in the message locally.

To find a <key,value>-pair, a node lookup for the key has to be performed as well, but with a small modification. Instead of FIND_NODE-RPCs, FIND_VALUE-RPCs are sent. The difference is that a node receiving a FIND_VALUE-RPC will return the <key,value>-pair if it has stored it. Otherwise it will return the k closest nodes to the key it has in its k-buckets, as if it had received a FIND_NODE-RPC. A node lookup for finding a <key,value>-pair stops immediately when a node returns the value.

Should a node lookup for a value be successful, that is, some node returns the value, then the <key,value>-pair is stored at the closest node to the key that did not return the value during the lookup. In other words, the value is cached at the closest node that did not return the value.

2.6 Republishing and Other Recurring Tasks

The specification of Kademlia [4] defines several recurring tasks:

- Every stored <key,value>-pair must be republished every hour.
- The original publisher of a <key,value>-pair must republish it every 24 hours.
- The lifetime of a stored <key,value>-pair is exponentially inversely proportional to the number of nodes between the own id and the node whose id is closest to the respective key. Thus, the expiration time is dependent on the structure of the k-buckets and must be recalculated every time a new node is added.
- When a new node is added to the k-buckets, it must be checked whether this node is closer to some of the stored <key,value>-pairs. If it is, the corresponding <key,value>-pairs must be replicated to that node.
- If in the range of a k-bucket no lookup has been performed for an hour, this k-bucket must be refreshed. This means that a node lookup is performed for an id that falls in the range of this k-bucket.

Chapter 3

Implementation

The aim of this chapter is to describe the actual implementation. It explains the concepts used to realize the specifications of the Kademlia peer-to-peer system stated in chapter 2. But the intention is not to give details on the methods and fields the various classes consist of. For this purpose, the program has been documented extensively with Javadoc [6].

3.1 General Architecture

In this section, the basic set-up of the implementation is introduced. Some of the concepts stated here are explained more detailed in the subsequent sections of this chapter.

3.1.1 Representation of the Messages

Separate Objects for Messages

The messages are special parts of a networking application. On the one hand, messages are like usual objects. They must be constructed and the data contained must be extractable. On the other hand, messages are sent and received over a network. This imposes them the necessity of a special representation accepted by the networking interface. At the beginning of the implementation work, the best form to represent messages was not clear.

From a technical point of view, the messages exchanged between the nodes across the network are just bit strings. Thus, the use of `byte[]` inside the program to represent the messages can be seen as a reasonable approach. For every type of message, a clearly defined format would have to be defined. That is, every message's byte sequence is split up in portions representing some specific data object which is special for the message. The interface to the classes used for networking is a second argument in favor of the `byte[]` representation. The constructor of the class `DatagramPacket`, which represents a packet to be sent over a `DatagramSocket`, accepts only `byte[]`s for the data representing the payload of the packet. Conversely, the data from a received packet can only be extracted in form of a `byte[]`.

But the operation with arrays is highly inconvenient. As `byte[]`s, every message has the same type, regardless of the function it has. That is, it is not possible to decide through type checks what specific kind of message a `byte[]` represents. Even worse, no special methods can be defined for the operations that must be carried out for the different types of messages. For example, for a message carrying the nodes found in a node lookup (see 2.3) there must be a means to get those nodes, whereas for most other messages it suffices to extract the id of the sending node. With the messages as `byte[]`s, all operations where something needs to be extracted must be performed through iterating over the array.

The desired properties just stated point to the solution to define separate objects for every kind of message. In this case, every message type could have its custom methods. But the problem is then how to send the messages over the network. A solution would be to make the classes

representing the messages implement the `Serializable`-interface. When doing so, the task of creating a suitable representation out of the object for sending over the network is performed by the Java Runtime Environment (JRE). In the application, one has thus only to operate with objects.

Unfortunately, serialization has some disadvantages:

- The message format is completely defined by the JRE.
- It can only be controlled to some extent (through protecting variables with the `transient`-statement) what is actually sent.
- As a consequence, the possibility exists that messages sent do not have their smallest possible size. This results in decreased performance.
- It cannot be guaranteed that all versions of the JRE perform the serialization in exactly the same way. Thus, instances of the program running on JREs of different versions may be incompatible.

The approach taken in this implementation takes the best out of both possibilities mentioned above. All messages are implemented by separate objects, but the conversion to a `byte[]` is not delegated to the JRE, but performed after a self defined pattern. This enables convenient handling of the messages insided the application, as they can be treated as normal objects. Through the custom built serialization, a clearly defined message structure is used and the maximal possible performance can be reached.

General Structure of the Messages

On the `byte[]`-level, the following partition of the messages has been chosen:

- The first byte is a demultiplexer key. Each kind of message has its own identifier. Upon reception of a message, this identifier enables to detect what message type has just been received.
- The next 20 bytes represent the id of the sender.
- In case of a message that answers some request, the subsequent 20 bytes represent the echoed random id. The echoed random id was the random id of the request message that forced the receiver to send the reply message (see below).
- As fourth item, a variable number of bytes representing the information the message carries follows. For example, in case of a message answering a request in a node lookup, the information are the nodes found.
- Always the last 20 bytes contain the random id chosen by the sender. As required by the specification of Kademlia [4], every request message must contain a randomly generated id that the receiver must echo. With this concept, *"some resistance to address forgery"* [4] is provided. Because of the concept explained in 3.1.2, not only request messages carry a random id, but also reply messages.

Figure 3.1 shows the structure stated above graphically.

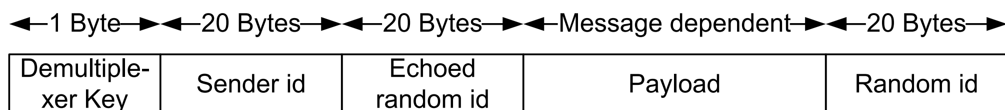

Figure 3.1: General Structure of the messages on the `byte[]`-level

For the specific structure of some type of message, see section 3.2.

3.1.2 The Pinging-Concept

As described in [4], ping requests can be piggy-backed on reply messages *"to obtain additional assurance of the sender's network address"*. This is important because the contact information of every node from which a message has been received is added to the k-buckets. In case a node sent a request message and received a correct reply message in turn, that is, the reply message contains the same random id as the request message, it can be quite sure that the inquired node is indeed the intended one. The situation is different when the first message a node received is a request message. In this case, the receiver cannot be sure whether the sender's contact informations are correct. It could be that the request was faked. To determine this, the piggy-backed ping is used. The effect of the piggy-backed ping is that the original sender of the request must send a ping reply upon receiving the reply message. Thus, the receiver of the request message is able to determine the correctness of the sender as well.

The result of the concept is that the sending of an RPC leads to the exchange of three messages (even in the case of pings, see 3.2.3). Figure 3.2 shows the exchanged messages in a message sequence chart.

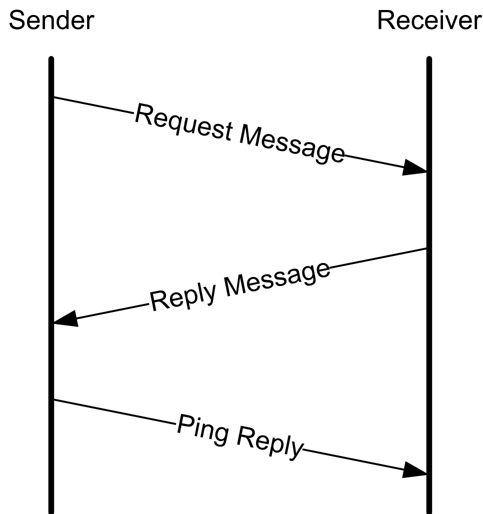

Figure 3.2: General course of a message exchange

3.1.3 Separate Threads for Receiving Messages and Answering Requests

According to the specifications given in chapter 2, it is clear that several tasks must be executed concurrently. To achieve concurrency, threads must be introduced. In the subsequent two paragraphs, the design choices behind the `ReceivingThread` and the `ReplyingThread` are explained. The third thread, `RefresherThread`, is described in section 3.1.6.

The ReceivingThread - Quick Reaction to Received Messages

The need for a thread that listens on the socket for incoming messages is obvious. As the `receive`-method of the `DatagramSocket`-class is a blocking operation, there must be a dedicated thread. The tasks the `ReceivingThread` must perform are very simple (for a description of them, see 3.5.1). In fact, it was the main goal to keep the `ReceivingThread` as simple as possible.

The reason for that is the limited size of the buffer assigned to the used socket. When the buffer is not emptied frequently, there is a possibility that it may overflow. Should that happen, information sent by other nodes gets discarded. Obviously, such a scenario must be avoided.

As the `ReceivingThread` does not have to perform expensive operations, it is ready to continue to listen on the socket for a new packet very quickly after the receipt of a packet. Thus, due to the fast reaction time, the danger of information loss caused by a buffer overflow is alleviated.

That the problem just explained is not of only theoretical nature but can occur in the real operation is shown the following example: Imagine that a sufficiently large number of hosts participate in the Kademlia network, i.e. the k-buckets are mostly full. Furthermore, a node performs a node lookup for an id. The id shall be such that a host inquired in the first round is the closest. As a consequence, the contacts the nodes inquired in the first round return all have a greater distance. In the node lookup procedure, this causes the executing node to send FIND_NODE-RPCs to all nodes among the k closest already seen that have not been inquired yet (see 2.3).

As assumed, the one of the nodes queried in the first round is the closest in the network to the looked-up id. It is thus likely that the other nodes of the first round are close as well, because they all probably come out of the same k-bucket. Therefore, it is reasonable to assume that the first round revealed α=3 of the k closest nodes. For the continuation of node lookup, this means that in the next round, assuming the standard value of 20 for k, $20 - 3 = 17$ nodes are inquired. Each of them will return a message that contains k=20 contacts (because of the assumption that there is a large number of nodes in the network).

Each such reply message has a size of 580 bytes (see 3.2.6). 17 messages thus have a size of $17 \cdot 580$ bytes = 9860 bytes. Because all nodes were inquired in the same round, those messages will arrive shortly after each other. On the computer used for the implementation, the size of the buffer is 8192 bytes. Hence, information will be lost if arrived messages are not removed quickly from the buffer.

A Thread-Pool for Concurrent Answering of Requests

As explained in section 3.1.2, every received message except ping replies causes the sending of an answer. In a first thought, one could be tempted to let the `ReceivingThread` perform the sending of reply messages as well. But this does not work. First, in case the received message was a request, e.g. a FIND_NODE-RPC, the corresponding reply message would contain the closest nodes to a specified id the inquired node knows of. But this comprises the searching of closest nodes in the k-buckets, which is not a trivial operation, as section 3.3.1 will reveal. Thus the principle of keeping the `ReceivingThread` simple would be violated.

Second, and much worse, the replying of messages is a short term blocking operation, as it must be waited for a ping reply after sending a reply message (see 3.1.2). Thus, at most for the duration of the timeout defined for waiting on a ping reply, the program would be blocked. Such a situation cannot be tolerated. The solution is again the introduction of a dedicated thread for answering requests. In the implementation, the thread is called `ReplyingThread`. The design of the `ReceivingThread` is explained in section 3.5.2.

During the execution of the program, several requests may arrive in short intervals. As a consequence, the requests need to be answered concurrently, because the sending hosts have assigned a timeout to the receipt of the corresponding reply message. It is thus required that several `ReplyingThread`s can be active at the same time. A first and straight forward solution would be to create a `ReceivingThread` every time a request message is received to answer it. In fact, in a preliminary version of the program, this was handled in this fashion. But such a solution is not optimal. It can push heavy load on the virtual machine in times of high activity in the network. The solution proposed implies the creation of a new thread every time a request is received. Furthermore, after finishing the reply, the thread has fulfilled its task. Therefore, the garbage collector must frequently remove old `ReplyingThread`s.

A better solution is to maintain a thread pool of `ReplyingThread`s, reusing threads in times of high activity. Because threads in a thread pool have a limited lifetime, unused `ReplyingThread`s are removed in times of low activity. That is why no memory is waisted at such times. Exactly the concept just described is used in the implementation.

3.1.4 Buffers for Temporary Storage of Received Messages

Together with the threads, buffers were developed. As described in 3.1.3, the `ReceivingThread` removes quickly the received messages from the I/O-buffer of the system. Right after that, the just received message needs to be temporary stored somewhere. Otherwise, the `ReceivingThread` would have to wait until another thread takes over the message for further processing. But this would violate the goal of keeping the `ReceivingThread` as simple and thus fast as possible (see 3.1.3).

Hence, buffers for temporary storage must be supplied. Every type of reply message has specialties regarding the relative importance and the way they must be treated. Notice that request messages need not be buffered. Those specialties influence the design of an optimal buffer for every message type. Therefore, a dedicated buffer for every type of reply message (with the exception of the `PingRepRepMessage`, see 3.2.3) has been implemented. Furthermore, the separating of the messages according to their types facilitated the programming work. It allows to search for a specific message type without the need for type checks.

The particularities of every type of buffer are explained in section 3.4.

3.1.5 Implementation of the Node Lookup Function

As stated in section 2.3, the node lookup function is vital in the Kademlia system. Out of this reason and because there arose some challenges for the implementation, its design is explained in this section.

Problem of Distinguishing the Rounds

The specification of the node lookup function [4] includes several requirements which are difficult to fulfill at the same time:

- The node lookup needs to be split up in rounds.
- If nodes fail to respond quickly, they must be removed from consideration until and unless they do respond.
- A next round can begin before all of the nodes queried in the current round have answered.

For the implementation, those requirements imposed some practical problems:

- How to distinguish between answers of different rounds?
- When should a round be considered to have finished?
- How to treat delayed answers?

To distinguish rounds, a data structure named `nodesEnquired` has been introduced that keeps track of the nodes queried in the current round. It is implemented as a `HashMap` that provides a mapping from ids to special objects exclusively used in the node lookup. These objects basically consist of the objects used for representing a contact (i.e. providing id, IP-address and port) in the network, augmented with flags to determine whether a contact has already been queried or not and whether an answer has already been received or not. After every round, bookkeeping must be performed. This means that the central data structure in the node lookup, a `TreeMap` providing a mapping from ids to the special objects introduced above, must be updated. The `TreeMap` keeps track of the contacts already discovered during the node lookup. The reason to choose a `TreeMap` comes from the requirement that it must be sorted according to the distances between already discovered and the id to be looked up. Otherwise, it could not be determined when the k closest nodes have been found in the network and thus when the lookup can be finished.

Updating means that answers received from nodes contained in `nodesEnquired` are added to the `TreeMap`. Furthermore, the entries in the data structure for the nodes that replied must

be modified (they were added to the `TreeMap` in a former round). Because they answered, the corresponding flag must be set to `true`. To conclude a round, `nodesEnquired` is cleared, new nodes are chosen from the `TreeMap`, queried and added again to `nodesEnquired`.

For the problem of deciding when to declare a round for finished, a solution based on a timeout has been chosen. After queries have been sent to the nodes of the current round, they are allowed a certain amount of time to answer. If all of them answer earlier, the node lookup continues. Otherwise, it is waited until this time has expired. Thus, in a stable environment, most of the times the answers of all inquired nodes can be considered without waiting for the timeout to occur. In fact, no other feasible solution was found. For example, the approach to declare a round for finished after a certain number of the inquired nodes answered is not favorable. If such a solution was used, then *always* some answers would be delayed. Clearly, this cannot be the goal.

The Losers-Concept

There are essentially two options how to deal with delayed answers: First, simply discard them. Second, consider them as well. Besides the fact that option one would violate a requirement of the node lookup (see the second point in the first enumeration above), it could have annoying consequences. As every answer provides valuable information, it would be a pity not to consider it simply because the message carrying it arrived too late. Imagine that the node lookup is used to find a value in the network. Maybe it is stored only on one node. If it should be the answer of this node that arrives too late, the lookup would fail although a node returned the value!

Thus, option two was chosen. The challenge in the implementation was to enable the consideration of the delayed messages without abandoning the division of the node lookup in rounds. This has been accomplished through the introduction of another data structure. As `nodesEnquired`, it is a `HashMap` providing a mapping from ids to the special objects explained above. The purpose of the new map is to store the contacts which failed to respond timely. That is why it was named "losers".

The usage of `losers` is the following: If it is detected that not all nodes responded within a provided time slice, the corresponding contacts are extracted from `nodesEnquired` and copied to the `losers`-map. After a round is finished, it is checked if a node contained in `loosers` answered in the meantime. Very importantly, this time *no* timeout is assigned! If none of the contacts in `losers` replied, the node lookup is continued. Conversely, if one or more nodes of the `loosers`-map replied, they are added to the data structure responsible for the course of the node lookup in the same manner as described above. With this concept, the division of the node lookup into rounds can be maintained, while delayed answers are considered nevertheless.

Integration of Value Lookups

Another requirement the node lookup must fulfill is that it must support both ordinary node lookups (that is, where a list of closest nodes is requested) and value lookups, where a value stored at some nodes in the Kademlia network is searched. The main difference between the two modes is that in the former case FIND_NODE-RPCs must be sent while in the latter FIND_VALUE. Furthermore, in the ordinary node lookup, only messages carrying lists of nodes are expected, while in the value lookup, messages containing values will be received in addition.

To support both operation modes, two separate function could have been implemented. Instead, it has been relied on a single, parameterizable implementation. The reason for this decision was that the two tasks have more similarities than differences. It has turned out that a simple flag is sufficient for differentiation and that only few distinction of cases need to be performed. This retains the readability of the code. Implementing two different methods would have lead to redundant code.

3.1.6 Priority Queue For All Periodic Tasks

According to section 2.6, there are several periodic tasks which must be performed. Soon it was clear that again one or more threads should be charged with those tasks. Threads provide the possibility to suspend themselves for a specified amount of time - a clearly desirable feature. Not clear was how to best distribute the tasks among threads. At the extremes, there are two solutions. Either a thread is assigned to each periodic task. For example, in the case of keeping the k-buckets fresh, this would mean that for every index a separate thread would be created. Or a single thread is responsible for *all* periodic tasks. Between those extremes, arbitrary combinations of merging tasks and assigning them to threads can be imagined.

Clearly, the first solution is not favorable. It produces a huge number of threads what cannot be efficient. For the intermediate solutions, no advantages compared to a single thread could be found. Thus, the solution with only one thread for all periodic tasks has been chosen. In the source code, the thread has been named `RefresherThread`.

A concept of how to schedule the different periodic tasks had to be developed. For this purpose, a priority queue based on a `TreeMap` has been implemented. The keys of the `TreeMap` are the points in time when a specific task must be performed. The values are lists of various objects all implementing the common interface `RefresherEntry`. For every periodic task, there exists a class which implements `RefresherEntry`. This interface specifies the three methods `update`, `getTimestamp` and `updateTimestamp`. Central among those methods is `update`, as its execution causes the periodic task, which the class stands for, to be performed.

Now the functioning of the priority queue can be understood. Being a `TreeMap`, it can be sorted. Reasonably, the sorting is in ascending scheduled execution time. Thus, the periodic action to be performed next is the first in order. If the time has come to execute a task, the first entry in the map is removed and its `update`-method performed. As there may be several actions scheduled for the same point in time, every key of the map points to a *list* of classes implementing the `RefresherEntry`-interface, as mentioned above.

It may look a bit strange to use a map to implement a priority queue. But reconsider the frequent events given in section 2.6 when the execution time of a periodic task must be modified. For example, each time a node lookup is performed, the `RefresherEntry` standing for the k-bucket in whose range the looked-up id lies must be updated. Therefore, once a `RefresherEntry` has been placed in the priority queue, the probability that it needs to be modified is high. A map provides easy access to a specific entry.

For the modification of the scheduled time of execution, the above mentioned methods `getTimestamp` and `updateTimestamp` are used. A more thorough explanation the `RefresherThread`, which includes the re-scheduling of events, is given given in section 3.5.3.

3.2 Messages

3.2.1 Hierarchical Structure of the Different Messages

All message objects have some fields and methods in common. For example, every message must contain the id of the sender and it must be possible to extract it. That is the reason why a hierarchical structure was chosen for the messages. On the highest level of the hierarchy is the class `Message`, that definines the common fields and methods of all messages. `Message` is extended by the two classes `RequestMessage` and `ReplyMessage`. These two classes serve as the base types for request messages and reply messages, respectively.

3.2.2 Ping Request and Ping Reply Message

A ping request is sent to ensure that a node is still online. For this task it is sufficient that a `PingRequestMessage` consists of the sender id and a random id only. Conversely, in a `PingReplyMessage` are only the id of the replying host and the echoed random id contained. Notice that the `PingReplyMessage` is the only message that does not contain a random id, as this

message always concludes a message exchange. Figure 3.3 shows the structure of a ping request and a ping reply.

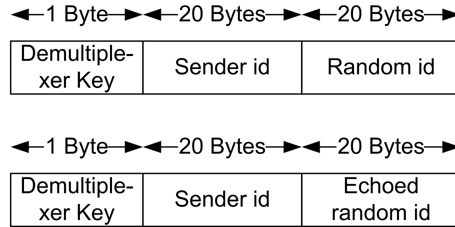

Figure 3.3: The structure of a `PingRequestMessage` (top) and a `PingReplyMessage` (bottom).

In section 3.1.1, it is explained that every kind of request message must contain a random id. Having the structure of a `PingRequestMessage` in mind, one can equivalently say that every request message contains implicitly a ping request. Translating this idea to reply messages, this means that every reply message contains a ping reply message as well as another ping request. This is consistent with the general pinging concepts introduced in section 3.1.2.

Those implicit ping requests and ping replies provide a convenient means to check whether a message is valid or not. The validity can be checked through saving the expected ping reply the reply message to be returned must carry, which consists of the id of the receiving node and the random id inserted in the just sent request message. Upon reception of the reply message, the saved ping reply and the one extracted from the reply message are compared. If they match, the reply message is valid. Otherwise, the reply message is a fake and can thus be discarded. Figure 3.4 illustrates this concept.

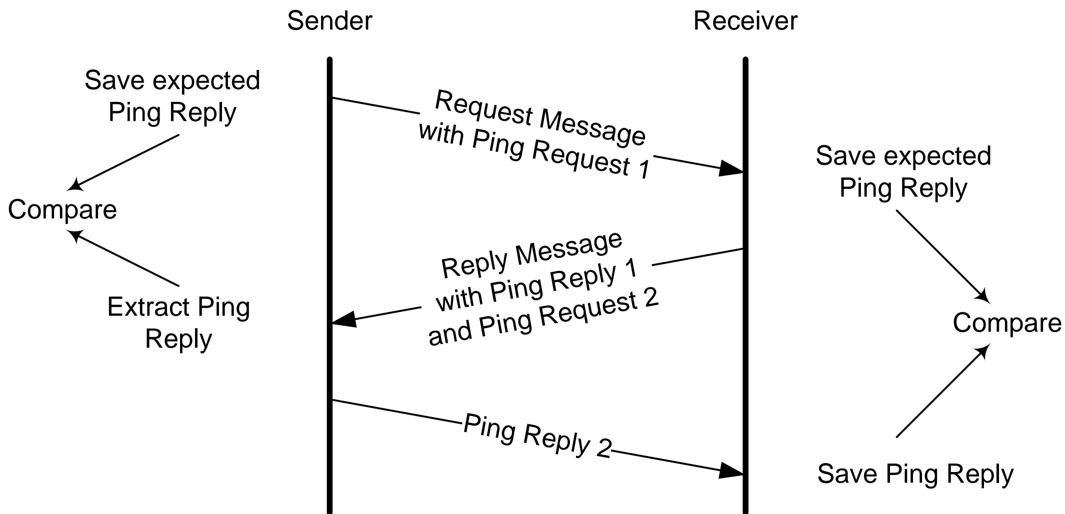

Figure 3.4: Extraction of ping replies to ensure validity of received messages.

Notice that the check of the validity of request messages happens indirectly through the additional ping reply that the sender must send back upon reception of the reply message.

3.2.3 Reasons For Introducing the PingRepReq-Message

Explicit ping requests are sent when the k-buckets want to check whether a host still participates in the network. To say consistent with the general pinging concept from section 3.1.2, the receiving node cannot just send a ping reply. When doing so, it could not ensure the pinging node's validity.

Thus, a message had to be defined which has to be sent as an answer to a ping reply. It is named `PingRepReqMessage` because it replies a ping and sends a new one in the same message. The structure of the `PingRepReqMessage` is given in figure 3.5.

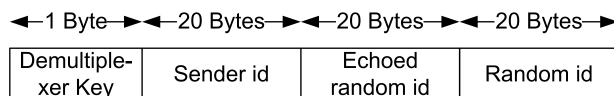

Figure 3.5: The structure of the `PingRepReqMessage`.

The sending of an explicit ping request results in the message exchanges depicted in figure 3.6.

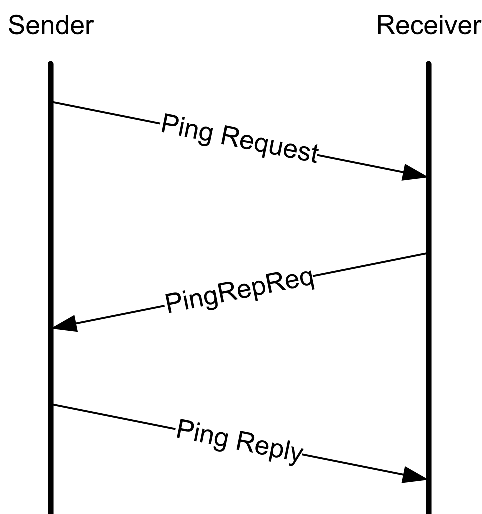

Figure 3.6: Course of the message exchange triggered by an explicit ping request

3.2.4 The FindNode- and FindValue-Message

Both `FindNodeMessage`s and `FindValueMessage`s are used in the node lookup. The former causes the receiver to locate the k closest nodes to the looked-up id it has stored in his k-buckets. The latter instructs the receiver to first check if it has stored a value corresponding to the looked-up id. If it has, the value shall be returned. Otherwise, it shall treat the `FindValueMessage` like a `FindNodeMessage`. To perform these actions, the receiver only needs to know the looked-up id. Thus, the payload of both messages consists of only that id, as shown in figure 3.7. The only field where the two messages differ is the demultiplexer key.

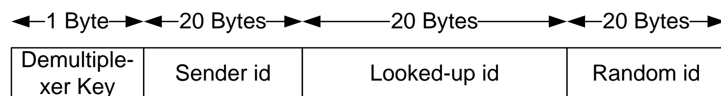

Figure 3.7: The structrue of the `FindNodeMessage` and `FindValueMessage`.

3.2.5 The Store-Message

`StoreMessage`s are sent to instruct a node to store the <key,value>-pair contained in the message. Therefore, the payload of a `StoreMessage` is a <key,value>-pair. A <key,value>-pair consists of an id (the key) and the IP-address and the port (the value) of the publishing node. The currently

used IPv4-address have a size of 4 bytes. The port can be in the range $0 \leq \text{port} < 655535$. As $65535 = 2^8 - 1$, 2 bytes are needed to represent the port. Together with the key which takes 20 bytes, the payload has an overall size of 26 bytes.

The message structure of the `StoreMessage` is depicted in figure 3.8.

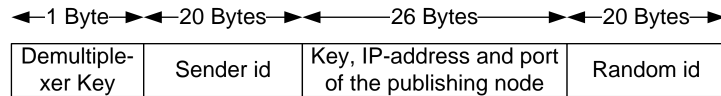

Figure 3.8: The structure of a `StoreMessage`.

3.2.6 The NodeReply- and ValueReply-Messages

A `NodeReplyMessage` is sent to answer either a `FindNodeMessage` or a `FindValueMessage`, when the node has not stored a value matching the contained key. With the `NodeReplyMessage`, the at most k closest nodes to the id in the received request message are returned. The contact information of a node consists of its id, IP-address and port. As explained in the discussion of the `StoreMessage`, such a triple has a size of 26 bytes. Thus, the payload of a `NodeReplyMessage` has a maximal size of $k \cdot 26$ bytes. In figure 3.9, the structure of a `NodeReplyMessage` is depicted graphically.

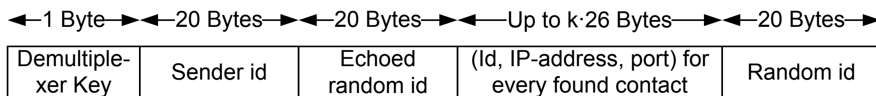

Figure 3.9: The structure of a `NodeReplyMessage`.

A `ValueReplyMessage` is the answer to a `FindValueMessage` when the receiving node has at least one value stored for the key in the request message. The payload of a `ValueReplyMessage` is therefore the key and all the corresponding values. A value is the IP-address and the port of the node storing the data object. Thus, a value has a length of 6 bytes. Figure 3.10 illustrates the structure of the `ValueReplyMessage`.

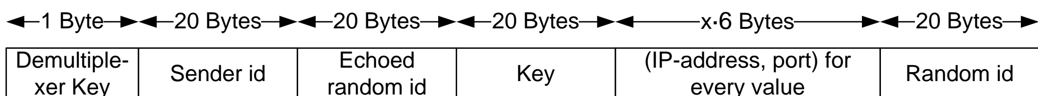

Figure 3.10: The structure of `ValueReplyMessage`. ”x” designates the number of values the key maps to.

3.3 Repositories

3.3.1 k-Buckets

As already stated in section 2.2, the k-buckets represent the central repository for storing contact information about other nodes in the Kademlia network. That is the reason why it is meaningful to begin the description of the implementations of the data structures with the k-buckets.

Selection of an Appropriate Data Structure for the k-Buckets

Having the description of the concepts behind the k-buckets in mind (see 2.2), one can formulate the requirements for the data structure to be used for the k-buckets as follows:

- It must be possible to access a specific k-bucket through providing the index of it.
- The adding and getting of contacts to and from the k-buckets must be as efficient as possible.
- Because of the least-recently-seen eviction policy the k-buckets shall implement, direct and easy access to the first and last element in a specific k-bucket must be ensured.

The requirements stated above lead immediately to the insight that the k-buckets can be looked at essentially as a two-dimensional array. In the one dimension, the appropriate k-bucket can be selected, while in the other one, a contact within this k-bucket can be retrieved. To implement such an array, basically two approaches seemed to be suitable, either a normal array or a linked list. Arrays have the advantage of positional access and that not for every element to be added new memory has to be allocated. A disadvantage can be the possible waste of memory when not all of the provided places the array offers are actually used. In contrast, a linked list never wastes memory as memory is only allocated when a new element shall be added and is freed again when an element is removed. Thus, a linked list is more memory efficient but also slower than an array. Another scenario where a linked list can outperform an array is when frequent adding and removing of elements at the beginning or the end of the list are needed and often iterations through the whole list must be performed.

For the k-buckets, it was first favored to use a two-dimensional array, because there is a fixed number of 160 k-buckets and every k-bucket stores at most k elements. But a closer look to the number of elements that will actually be stored in a single k-bucket and the operations that will be performed on it, revealed that an array is not the best solution. In fact, a linked list makes there more sense. Concretely, this is on the one hand because the k-buckets store elements close to the own id (i.e. k-buckets with low indexes) are usually empty. The reason for that is the missing of contacts with appropriate ids in the network, as such a contact would have to share a very long common prefix with the own id, which is very unlikely. Therefore, an array would have wasted memory in many cases. On the other hand, due to the least-recently-seen eviction policy of the k-buckets, contacts are only added at the head of the list and removed from the tail. Such operations clearly favor a linked list over an array. What even enforces the use of a linked list is that frequently a search for elements that are closest to some given id must be performed inside a k-bucket, what means iterating over the elements in the list. To sum up, a single k-bucket has nearly all characteristics that favor the use of a linked list over an array.

The situation is different for the superior list that holds the single k-buckets. This list has always a length of 160 and positional access is frequently necessary. Thus, an array makes sense there. All the arguments stated lead to the decision to implement the k-buckets through an array of size 160 that can take linked lists which actually store the contacts.

But the array and the linked list were not programmed but just imported from `java.util`. The array is represented by an `ArrayList` while the linked list is represented by a `LinkedList` from that package. It has not been found any reason not to use those ready-made data structures. Programming them from scratch would have been time consuming and error-prone.

Locating the Closest Nodes to a Given Id

In the node lookup procedure, nodes have to find a specific number of nodes closest to specified id in their k-buckets (see section 2.3). This is an easy task when the appropriate k-bucket contains at least the number of requested nodes. Otherwise, the nodes must come from different k-buckets. Therefore, several k-buckets must be visited in order to collect the nodes. The order of the visits is such that the distance to the given id increases from k-bucket to k-bucket. To get the idea how this is accomplished, one looks best at a concrete example.

Assume there are, for convenience of illustration, only 16 bit (instead of 160 bit) long ids used in the system. Assume further that the own id is 1001110110000101 and the id to which the closest ids shall be searched is 1001111110100001. To get the distance between the two, they have to be xored, as shown in table 3.1.

position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
own id	1	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1
given id	1	0	0	1	1	1	1	1	1	0	1	0	0	0	0	1
distance	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0

Table 3.1: Two 16 bit ids and the distance between them

The order in which the k-buckets have to be visited can now be derived from the distance. In this particular example it is: 9→5→2→0→1→3→4→6→7→8→10→11→12→13→14→15. The following remarks may clarify this:

- k-Bucket nr. 9 contains the elements with distance $0 \leq d < 2^9$ to the given id, where $d = 0$ corresponds to the case where the k-bucket contains the given id itself, and $d = 2^9 - 1$ to the id 1001111001011110
- k-Bucket nr. 5 contains the elements with distance $2^9 \leq d < 2^9 + 2^5$ to the given id, where $d = 2^9$ corresponds to the id 1001110110100001, and $d = 2^9 + 2^5 - 1$ to the id 1001110110111110
- k-Bucket nr. 2 contains the elements with distance $2^9 + 2^5 \leq d < 2^9 + 2^5 + 2^2$
- k-Bucket nr. 0 contains the elements with distance $2^9 + 2^5 + 2^2 \leq d < 2^9 + 2^5 + 2^2 + 2^0$
- For the k-buckets nr. 1 up to nr. 8, the pattern from k-bucket nr. 0 continues
- k-Bucket nr. 10 contains the elements with distance $2^{10} \leq d < 2^{11}$
- The k-buckets nr. 11 up to nr. 15 are similar to k-bucket nr. 10

To sum up, first the k-buckets are visited in descending order whose index corresponds to a position in the distance bit string which is one (phase 1). After that, it has to be continued at the k-bucket with the lowest index whose corresponding position in the distance bit string is equal to zero. It is then continued is ascending order (phase 2).

Calculation of the Number of Intermediate Nodes

In section 2.6, it is mentioned that every value has an expiry time. Furthermore, this expiry time is dependent on the number of nodes between myself and the key corresponding to the value. This number can be calculated with the same method as described above.

First, the distance of the key and my own id must be calculated. Thereafter, phase 1 of the procedure from above must be executed. In addition, the number of nodes in every visited k-bucket is summed-up. When finished, this number is the number of intermediate nodes. This is obviously true, because the the k-buckets are visited in increasing distance to the key. Why only phase 1 must be performed is due to the fact that after its end the own id is the next closest. Thus, no more nodes can be between the key an myself.

3.3.2 KeyValueContainer

In the `KeyValueContainer`, the <key,value>-pairs published by other nodes are stored. Remind from section 2.5 that a key is an id of a data object (for example a hash code) and a value is the 3-tuple {id, IP-address, port}, where "id" is the same id as the key, "IP-address" is the IP-address of the host storing the data object and "port" is the port on which the storing node accepts connections. To avoid confusing the terms "key" and "value" in the context of the Kademlia network with those in the context of describing a map data structure, the former are referred to henceforth as "object id" and "object pointer", respectively.

Besides the basic methods for adding, getting and removing of <key,value>-pairs, the KeyValueContainer contains also two methods for ensuring consistency: The replicating and republishing methods. After a first short paragraph on the data structure chosen for the <object id,object pointer>-pair storage, the emphasis is put on those two functions, because the other, very basic functions do not have any special properties.

Data Structure Used

The data structure used for the KeyValueContainer is a HashMap. As keys serve object ids, the values are Lists containing object pointers. The reason why a single object id maps to a list of possibly several, distinct object pointers is due to the fact that the hash code of two or more data objects can yield to the same result. A scenario when this can happen is if two or more data objects share a common portion of their names.

The choice of a Map data structure for the <object id,object pointer>-pair storage is logical. Hosts that request an object pointer, for contact information to a node that has the requested data object, will send FindValueMessages with the hash code of the wanted object (that is, a object id). Thus, a mapping from object ids to object pointers must be provided.

There are essentially two basic choices for Map data structures in java.util, namely HashMap and TreeMap. The former is better performing but stores the values in an order that cannot be specified, while the latter provides this functionality but is slower ($\log n$, n=number of elements in the Map, time complexity for the basic get, put and remove operations [7] compared to constant time for HashMap [8]). As in the case of the KeyValueContainer the ordering is not important, the choice was made for the HashMap.

Replication of <key,value>-Pairs to a Newly Discovered Node

As required by the specification of the Kademlia protocol, every time a new node is discovered, it must be checked if that node is closer to some of the stored <object id,object pointer>-pairs in the KeyValueContainer (see 2.5). Discovering a new node means that it is not already stored in the k-buckets. With that measure, *"consistency in the publishing-searching life-cycle"* [4] is sustained. Should that apply, StoreMessages must be sent to the node for the respective object pointers. This process is called "replication". The KeyValueContainer provides the method replicate to do that.

To find out whether the new node is closer to some of the stored object ids or not, the distance of every object id to the new node as well as to the myself must be calculated. After that, should the new node be closer, STORE-RPCs are being sent to the new node for all object pointers the object id maps to.

There is a specialty with sending those STORE-RPCs. As opposed to the normal course when RequestMessages are sent, in the current situation the node *must not* be added again to the k-buckets when it replies correctly with a PingReqRepMessage. The reason, as illustrated by figure 3.11, is the following:

Imagine the situation when the node that becomes aware of the new node has the corresponding k-bucket full. Therefore, the new node will not be added (it is assumed that the least recently seen node in the k-Bucket answers the ping). But the new node may be closer to some of the stored object ids. Thus, it replicates the respective object pointers to the new node. The new node in turn stores those object pointers, and answers the StoreMessages correctly with a PingRepReqMessage. If now the node tried again to add the new node to the k-Buckets, it would again not add it but replicate again the object pointers to which the new node is closer. Obviously, there would be a loop in the network.

Republishing

The KeyValueContainer provides the method republish to republish all object pointers it has stored in the HashMap. The method is called by the RefresherThread whenever a

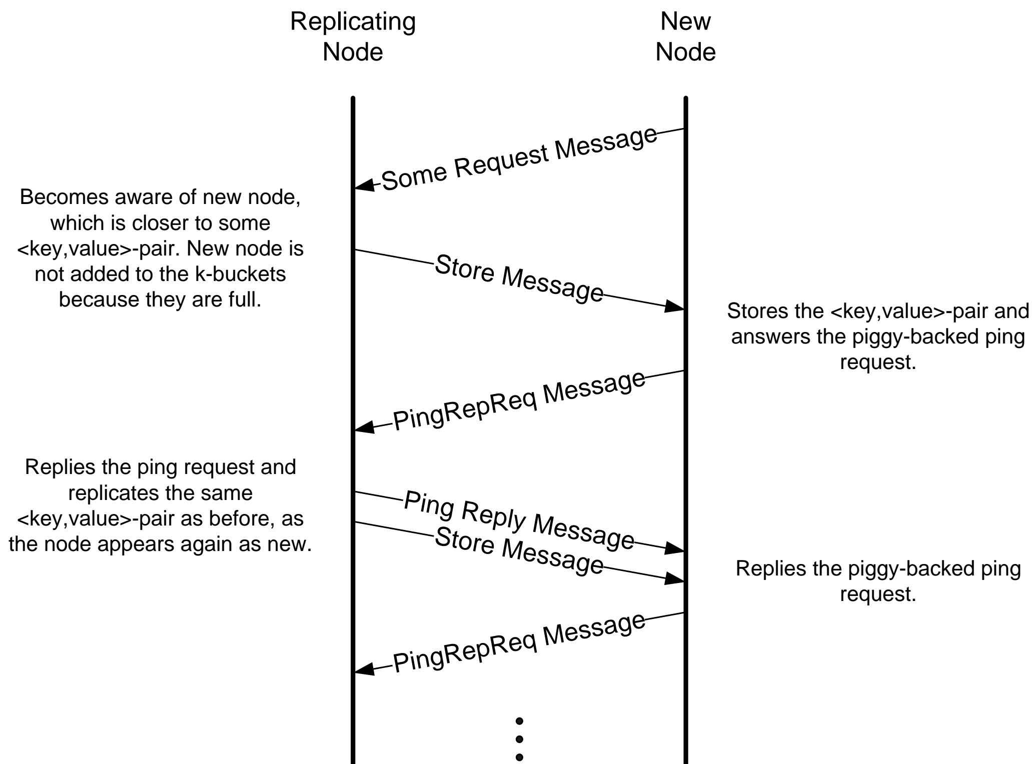

Figure 3.11: Adding the node to the k-buckets to which a STORE_RPC was sent results in a loop.

`KeyValueRefresherEntry` is due for execution (see 3.5.3). Invoking of the `republish`-method causes the performing of a node lookup for every key the `HashMap` has a mapping for, and the subsequent sending of `StoreMessage`s to the nodes the node lookup has revealed.

3.3.3 PublishedValuesContainer

The `PublishedValuesContainer` stores the <object id,object pointer>-pairs for which this node is the original publisher. The only reason why they must be stored is to provide the ability to republish them every 24 hours, as described in section 2.6. As data structure, the `PublishedValuesContainer` employs a `HashMap`, out of the same reasons as the `KeyValueContainer` (see 3.3.2). But there is a difference to the `KeyValueContainer`: every object id maps to at most one object pointer only. Even when a node publishes two or more data objects yielding the same hash code, the created object pointers are the same for all (remember that a object pointer stores only the id of a data object, i.e. the hash code, the IP-address and the port of the host who publishes it).

The `PublishedValuesContainer` has as well as the `KeyValueContainer` a `republish`-method. It republishes a single object pointer. Note that this is slightly different to the republishing the `KeyValueContainer` performs (see 3.3.2), where *all* object pointers are republished together. For the object pointers for that a node is the original publisher, this must be different, as every such object pointer has assigned its own `PublishedValuesRefresherEntry` (see 2.6) and is thus republished individually.

3.4 Buffers

3.4.1 PingReplyBuffer

The aim of the `PingReplyBuffer` is to temporary store the received `PingReplyMessage`s (according to the concept of all buffers employed, see 3.1). In the remainder of this paragraph, the design decisions behind the `PingReplyBuffer` are explained.

Protection Against Unrequested Entries

A straight forward implementation of the `PingReplyBuffer` would have one single data structure, a list for example, which saves the `PingReplyMessage`s received by the `ReceivingThread`. In addition, a method for checking if an expected `PingReplyMessage` has arrived would have to be included, such that the `ReplyingThread` can instruct the k-buckets to add a contact or the k-buckets can ensure that a contact is still alive. In a preliminary version of the program, the `PingReplyBuffer` was implemented in this fashion.

Imagine what happens if a malicious entity sends `PingReplyMessage`s that were not requested by the instance of the program whose `PingReplyBuffer` is the subject the current discussion. Those `PingReplyMessage`s would be added to the `PingReplyBuffer`, consuming more and more memory, as they will never be removed, because they were not requested. As a consequence, the system where the program is running on will eventually run out of memory. Hence, a countermeasure has to be included to avoid such a situation.

The Need for Two Data Structures

The concept developed for the protection of the `PingReplyBuffer` uses the following fact: Whenever a host is forced to send a `PingReplyMessage` back to the host who requested it, its contents are known beforehand (see 3.2.2). Thus, it can be decided whether a received `PingReplyMessage` is legitimate or not.

In the implementation, this distinction between legitimate and not legitimate `PingReplyMessage`s is achieved through announcing the `PingReplyMessage`s the `PingReplyBuffer` has to expect beforehand. Concretely, the `ReplyingThread` adds the expected `PingReplyMessage` to a data structure called `setExp` in the `PingReplyBuffer` whenever it sends any kind of reply message. Similarly, the k-buckets-object does the same indirectly when it sends a `PingRequestMessage` to some node. Indirectly because it is in fact the object which represents the node to be pinged that does it. The other opportunity when a node object must announce a `PingReplyMessage` to the `PingReplyBuffer` is when it is requested to send a STORE-RPC.

If a `PingReplyMessage` is received, the `ReceivingThread` passes this message through calling of the method `addReceivedMessage` to the `PingReplyBuffer`. There, it is decided whether the message is legitimate or not. Thanks to the announcement of the expected `PingReplyMessage`s, it can be checked if the received message is among the expected ones. For this purpose, the `setExp` data structure is searched for the received message. If it contains the message, it is removed from `setExp` and stored in another data structure named `setRec`. Otherwise, the received message is discarded. Obviously, `setRec` contains only the legitimate `PingReplyMessage`s received. This is necessary for the `checkMessage`-method (see below), which is called to check if a node actually sent a `PingReplyMessage`.

Some words about the choice of the data structures used for `setExp` and `setRec`. As their names already suggest, `Set` implementations out of `java.util` have been chosen, more precisely `HashSet`s. That is because frequent searches for some specific elements are necessary. In the case of `setExp` this applies when it is verified that a received message is also among the expected. For `setRec`, this applies when it is checked if the a node supposed to send a `PingReplyMessage` actually did that. Such operations are most efficiently performed by hashing. The restriction of a `Set` that it cannot store duplicate elements is no drawback since s node will never send two or more identical `PingReplyMessage`s.

Searching the PingReplyBuffer for PingReplyMessages

Providing the expected PingReplyMessage to the PingReplyBuffer is only the first step to check if a node answered an explicit or piggy-backed ping. The second step is to ensure that it actually replied correctly. For this purpose, the checkMessage-method has been implemented.

When this method is entered, it is first checked if the desired PingReplyMessage is already contained in the setRec data structure. If it is, the message is removed from setRec and true is returned. Otherwise, the thread waits until a specified amount of time has expired or if a new message has been added. In the former case, false is returned, while in the latter it is checked again if the newly added message is the desired one.

The getFirstMessage-Method

The PingReplyBuffer must provide a special method, named getFirstMessage, that is called exactly once during the execution of the application, namely at the very beginning (see 2.4 for details about the bootstrapping process). If the host is not a gateway, it gets passed the IP-address and the port of a gateway. The first action to be done is to send a PingRequestMessage to this host in order to ensure that it is alive. Thus, it has to be checked afterwards if the gateway responded, i.e. if it sent a PingReplyMessage. But as the id of the gateway is not known, the checkMessage-method cannot be called as the PingReplyMessage could not be announced. That is the reason why the getFirstMessage-method had to be implemented. It simply returns the first (and only) message stored in setRec if the gateway responded timely.

3.4.2 NodeReplyBuffer

The NodeReplyBuffer is similar to the PingReplyBuffer regarding its main purpose. Like the PingReplyBuffer, it serves as a temporary storage for received messages too, with the difference that it is responsible for the NodeReplyMessages. But there are also differences. For example the concept to prevent the NodeReplyBuffer from being misused to sabotage the system is completely different as well as the data structure used to store received node replies.

Protection Concept

The measures taken against the filling of the NodeReplyBuffer with unrequested messages is quite simple. When the NodeReplyBuffer is not used, it can be disabled. If the buffer is disabled and should nevertheless a NodeReplyMessage be received, it is discarded.

For the NodeReplyBuffer such a simple protection concept is sufficient, as opposed to the PingReplyBuffer. The justification lies in the circumstances when the respective reply messages are expected. In the case of the NodeReplyMessages, this is only when the application is in the process of a node lookup, as never else FindNodeMessages or FindValueMessages are being sent. The point in time when the NodeReplyMessages are expected is thus set by the application itself. It is therefore clear when the NodeReplyBuffer has to enabled and when it must not accept any messages.

Of course, during the time of a node lookup, unrequested messages may be added to the NodeReplyBuffer. But as a node lookup takes only a very limited amount of time, it is highly unlikely that enough of those messages are received to let the system run out of memory. Furthermore, after every node lookup, the NodeReplyBuffer is cleared.

The situation is different with the PingReplyBuffer. PingReplyMessages are expected to be received when any kind of request message has been answered. But it is not foreseeable when requests arrive. When a node has stored a large number of object pointers, a lot of request messages will be received. Thus, the same is true for ping replies. In such a case, the pingReplyBuffer cannot be disabled, as it must wait for those ping replies to arrive. As a consequence, even if the PingReplyBuffer had a switching-off mechanism, it would be enabled most of the time and therefore allowing unrequested ping replies to be stored. Obviously, translating the simple protection concept of the NodeReplyBuffer to the pingReplyBuffer is not applicable.

Conversely, the concept used in the `PingReplyBuffer` is feasible for the `NodeReplyBuffer`, though not favourable. As every reply message, a `NodeReplyMessage` provides as well the ability to extract the expected `PingReplyMessage` (see section 3.2). Thus, it would be possible for the `NodeReplyBuffer` to have a second data structure with expected ping replies as well. But as this second data structure is not really needed, because of the simpler possible protection concept, it is avoidable overhead.

Data Structure Used

One of the requirements for the data structure to hold the received `NodeReplyMessage`s is the ability to access them via the id of the node which sent it. This is because the node lookup-function keeps track of ids of the nodes which it has already queried. To find out if a node answered, the node lookup-function needs only to provide such an id to get the corresponding `NodeReplyMessage`. <key,value>-pair storage is provided by the `Map`-implementations in `java.util`. The choice of the actual implementation was straight forward. As there are not any ordering constraints, `HashMap` was chosen. It has the advantage of being the fastest implementation among all `Map`s.

Retrieving of NodeReplyMessages out of the NodeReplyBuffer

In a node lookup, always several `RequestMessage`s are sent concurrently. Thus, also several `NodeReplyMessage`s are expected. For the `NodeReplyBuffer`, this means that it has to look for several `NodeReplyMessage`s concurrently. The method `getMessage` has been implemented for this purpose, which works as follows: As soon as one desired messages has arrived, it is returned. If no message arrives within a defined timespan, `null` is returned. The node lookup function, which is the caller of this method, will then process the answer and make a further call when it is finished with processing.

A specialty is that the retrieval of `NodeReplyMessage`s can be interrupted by the `ValueReplyBuffer`. The reasons for this are stated in the next paragraph.

3.4.3 ValueReplyBuffer

The `ValueReplyBuffer` is very similar to the `NodeReplyBuffer`. Because the `ValueReplyBuffer` is used exclusively during node lookups as the `NodeReplyBuffer`, it relies on the same protection concept (see 3.4.2). The data structure is also a `HashMap`, because of the identical reasons as stated in 3.4.2. The difference is of course that the `Map` in the `ValueReplyBuffer` maps ids to `ValueReplyMessage`s.

Differences to the NodeReplyBuffer

Besides all similarities, there are some differences, too. Those differences lie in the higher relative importance of `ValueReplyMessage`s compared to `NodeReplyMessage`s. As opposed to node replies, the reception of `ValueReplyMessage`s during a node lookup means that probably one or more hosts returned the requested object pointers. While the reception of `NodeReplyMessage` is the normal case in a node lookup, the reception of a `ValueReplyMessage` is a special event indicating that the node lookup can be finished soon probably (if the `ValueReplyMessage` is valid).

Thus, value replies must get priority over node replies. In the implementation, this is accomplished through the ability of the `ValueReplyBuffer` to interrupt the `getMessage`-method of the `NodeReplyBuffer`. Whenever a new `ValueReplyMessage` is added, the `ValueReplyBuffer` calls the `setInterrupt`-method of the `NodeReplyBuffer`. The result in the node lookup function is that it immediately retrieves the received `ValueReplyMessage` out of the `ValueReplyBuffer`.

3.5 Threads

3.5.1 The Receiving Thread

The ReceivingThread is responsible to listen on the socket for newly received messages. If one has arrived, the ReceivingThread removes it from the buffer of the socket. After that, the thread checks the first byte of the message, which is the demultiplexer key. Based on that key, the corresponding message object can be constructed out of the byte[].

If the received message is a request message, it is passed to a ReplyingThread of the thread pool. Otherwise, if the message is a reply message, it is put in the corresponding buffer. Additionally, the piggy-backed ping request is extracted and passed to a ReplyingThread. The third possibility is when the received message is ping reply. In this case, the message must only be added to the ping reply buffer, as the receipt of such message signifies the end of a message exchange.

3.5.2 The Replying Thread

The ReplyingThread gets request messages passed from the ReceivingThread, which it evaluates subsequently. This means that it performs the action assigned to the passed message. If the passed message is a FindNodeMessage, for example, then the ReplyingThread causes the k-buckets to return the k closest nodes to the id contained in the FindNodeMessage. The next step is to generate the corresponding reply message. In the example from above, this would be a NodeReplyMessage. Referring to figure 3.4, it is clear that now the expected ping reply must be saved. More precisely, the expected ping reply is announced to the PingReplyBuffer. After that, the generated reply message can be sent. The last step is to wait until the receiving host returned the ping reply or a time out occurs. Should the ping reply be correct, the contact data of the node are tried to be added to the k-buckets.

3.5.3 The Refresher Thread

Basic Course of Operations

The RefresherThread performs the periodic tasks of Kademlia described in section 2.6. To that end, it uses the TreeMap based priority queue introduced in section 3.1.6. The basic course of operations is the following: First, the key of the entry at the lowest position in the map is analyzed. Remember that the keys represent the scheduled execution time of the RefresherEntrys assigend to them. After that, the thread suspends its operation until the time for the execution has come (or a rescheduling action takes place, see below). In the execution phase, the thread iterates over the list of RefresherEntrys assigned to the key. In every step, it executes the update method of the entry. This causes the specific periodic action to be performed. Subsequently, if there were republishing entries among the just executed, they have to be reinserted into the priority queue, but not before their execution time has been reset. That happens through a call of the updateTimestamp-method which all RefresherEntrys provide.

Offloading Non Time Critical Tasks to the Refresher Thread

RefresherEntriys inserted in the priority queue likely need to be rescheduled (this was one of the reasons for choosing a TreeMap, cf. section 3.1.6). The point in time when an entry must be rescheduled is dependent from external circumstances, that is from events outside the RefresherThread. For example, this can be in a node lookup or when a new contact is added to the k-buckets. It would be not a good idea to let those rescheduling actions take place when it is known the first time that they must occur. This is most of the time in time critical situations with timeouts involved. Thus, it is not desirable to prolong the respective task and to risk a timeout. As a consequence, these actions are be offloaded to the RefresherThread, which performs them later. Most of the times, this thread must wait for the next refreshing action to take place. In the meantime, it can perform the rescheduling of entries. Another task that has been offloaded to

the `RefresherThread` is the replicating of object pointers. The reasons are the same as for the rescheduling.

Chapter 4

Testing

To ensure the proper functioning of the implementation, different kind of tests were performed:

- During programming, several instances of the application were launched on the same computer, which communicated over the loop back address. The purpose of these tests was to check if just finished extensions of the program work in principle.
- The next step was installing the implementation on up to 10 hosts on the Tardis student computers. Therewith it could be shown that the implementation works on a limited number of computers in a real network.
- The largest test was performed on the PlanetLab platform. PlanetLab is a worldwide interconnection of computers to test networked applications [9]. On PlanetLab, the implementation could be tested on 160 hosts connected through the real Internet. Thus, this scenario comes already quite near the intended use of the program within Celeste. During the tests on PlanetLab, no serious bugs emerged.
- A quite different test was the check of the source code with the static class file analyzer Jlint [10]. This tool emitted warnings, but they were checked and considered as non-critical.

Chapter 5

Future Work

In the context of the implementation described in this report remain several issues to be performed in the future. Specifically, these are:

- Further tests need to be performed. All the periodic tasks have not been tested yet on the PlanetLab platform. This is because the respective version of the program was not finished when the already performed tests were executed on PlanetLab.
- Performance measurements: Up to now, no performance values are known. For example, the time how long a node lookup takes would be an interesting value.
- Depending on the results of the preceding point, performance improvements need to be included. Some improvements are described in [11].
- The main task will be the integration of the Kademlia DHT into Celeste. This could not be done yet because it is a complex and time-consuming task, mainly due to frequent changes in the interfaces Celeste provides.
- Once the integration in Celeste has been accomplished, a comparison of the Kademlia DHT to the existing one in Celeste would be highly interesting. Such a comparison could justify whether it was reasonable to develop a Kademlia DHT for Celeste or not.

Chapter 6

Conclusions

In this report, an implementation of the Kademlia DHT in Java was described. Specialties of Kademlia are its XOR-topology and the iteratively working node lookup. In the future, the implementation shall become part of the Celeste automatic storage system.

The main architectural characteristics are separate threads for receiving and answering of messages, as well as a thread for performing the periodic tasks. Furthermore, messages have been implemented as normal Java-objects, which can be converted to `byte[]` through a custom built serialization. For the temporary storage of messages, buffers have been included, which need protection concepts to avoid a memory leakage of the system. The implementation uses three long term storages: The k-buckets are used to store the contacts of known nodes in the network and thus represents the routing table. In addition, there are two repositories to store <key,value>-pairs.

Several tests were performed to ensure the proper functionality of the system. Up to now, no severe faults have occurred. Therefore, it can be assumed that the implementation basically works.

There are some tasks to be performed in the future. Most notably, the implementation of Kademlia must be included in Celeste, as this was the reason for developing it.

Bibliography

[1] Germano Caronni, Raphael Rom, Glenn Scott. *Celeste: An Automatic Storage System.* http://www.sun.com/products-n-solutions/edu/whitepapers/pdf/celeste_automaticstorage.pdf

[2] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. *Tapestry: A Resilient Global-scale Overlay for Service Deployment.* IEEE Journal on Selected Areas in Communications, January 2004, Vol. 22, No. 1.

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

[4] Petar Maymounkov, David Mazières. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.* In *Proceedings of IPTPS*, Cambridge, MA, USA, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf

[5] Stefan Saroiu, P. Krishna Gummadi and Steven D. Gribble. *A Measurement Study of Peer-to-Peer File Sharing Systems.* Technical Report UW-CSE-01-06-02, University of Washington, Department of Computer Science and Engineering, July 2001.

[6] Sun Developer Network. *Javadoc Tool* http://java.sun.com/j2se/javadoc/

[7] Sun Developer Network. *API Specifications* http://java.sun.com/j2se/1.5.0/docs/api/java/util/TreeMap.html

[8] Sun Developer Network. *API Specifications* http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html

[9] PlanetLab. http://www.planet-lab.org/

[10] Jlint Static Class File Analyzer. http://artho.com/jlint/. Artho Software.

[11] Daniel Stutzbach, Reza Rejaie. *Improving Lookup Performance over a Widely-Deployed DHT.* In *Proceedings of IEEE INFOCOM*, Barcelona, Spain, April 2006. http://www.cs.uoregon.edu/~reza/PUB/infocom06-kad.pdf