Bachelor thesis

-

Reliable UDP and Circular DHT implementation for the MediaSense Open-Source Platform

Florian Rüter,
Timo Schröder

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

Abstract

MediaSense is an EU funded platform that is an implementation of an Internet-of-Things framework. This project adds two fundamental functions to it, namely, a new lookup service based on a peer-to-peer Distributed Hash Table (DHT) called Chord and a reliable communication protocol based on UDP (RUDP). The lookup service makes the use of a central server, that can be a single point of failure or get compromised, unnessecary. Reliable UDP transmits data from the very first packet onwards and avoids any connection management as it is packet based. The methodology for both functions was to develop a simulation environment, compatible to MediaSense, at its initiation, at which point its functionality can be tested and measurements can be taken. The resulting DHT simulation environment enables there to be deep insight into and a control of the state and action of the DHT. The resulting graphs show the performance properties of both the DHT and RUDP. In conclusion, the MediaSense platform has been extended by means of two usable functionalities and which also leaves space for further development such as for security enhancements and performance increases.

Keywords: Internet-of-Things, MediaSense, Peer-to-peer, Chord, Distribut Hash Table, Networking, Reliable UDP, Simulation

Table of Contents

Abbreviations

API	Application Programming Interface
ARQ	Automatic ReQuest
BSD	Berkely Software Distribution
DHT	Distributed Hash Table
FSM	Finite-state machine
IoT	Internet-of-Things
P2P	Peer-to-peer
QoE	Quality of Experience
QoS	Quality of Service
RTT	Round Trip Time
RUDP	Reliable UDP
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1 Introduction

This work is the final thesis for the programme European Computer Science Studies at Hochschule Osnabrück created during exchange studies at Mid Sweden University in Sundsvall within a timeframe of 400 hours per person.

1.1 Background

Applications that can change their behavior based on the context of the users are called context-aware applications. By intruducing smart mobile phones that carry a multitude of sensors and actors, these applications have caused there to be a very large market penetration. Mid Sweden University has so far produced the initial components for the accumulation of context information from sensors and wireless sensor networks from numerous sources, e.g. sensors attached to home networks or mobile phones. This project is associated with the development of a next generation Internet-of-Things (IoT) architecture and their supporting protocols. The IoT is defined as applications that use information from sensors and actors to provide personalized, automatized or intelligent behaviors to the users.

1.2 Overall aim and problem motivation

To initiate a connection between two MediaSense instances (contexts) it is necessary there to be a lookup-service. To handle the communication between two contexts a transport protocol is required. The present lookup-service, that has been implemented by Mid Sweden University so far, is based on a client-server architecture. The well-known network protocols Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) can be used for data transmission. Since its inception, there has been a requirement for the MediaSense platform not to have a single point of failure. The client-server architecture does not fulfil that need and for this reason a distributed lookup-service will be developed during this thesis work. The data transmission of the MediaSense platform has to be reliable and packet oriented. Neither TCP nor UDP

are able to simultaneously fulfil both requirements and thus it is necessary to produce a self-made communication protocol.

1.3 Detailed problem statement

To have verifiable goals, eight milestones are defined that are used throughout the development process and this report.

1. Develop a simulation environment using the same Application Programming Interface (API) as MediaSense, that enables multiple DHT clients to run in parallel

2. Implement a circular Distributed Hash Table (DHT) based on Chord running inside the simulation environment that enables register and resolve operations to be executed in $O(\log(n))$ time

3. Evaluate the performance of the DHT in terms of its self-healing capability and network usage

4. Transfer the DHT solution into the MediaSense platform

5. Develop a simulation environment, using the same API as MediaSense, which is able to host communication between two clients over an exchangeable protocol

6. Develop a reliable packet-based communication protocol (RUDP) based on Berkely sockets (BSD sockets)

7. Compare RUDP to TCP in terms of speed.

8. Transfer the RUDP solution into the MediaSense platform

1.4 Scope

The DHT implementation is focused on handling a client who joins, leaves and fails while retaining the health of the DHT. For the RUDP implementation the thesis focuses on transmitting data with the first sent packet and avoiding a 3-way handshake in order to synchronize the sender and receiver.

This thesis does not focus on any counteractive measurement regarding security issues such as network attacks and data encryption for both, DHT and RUDP.

A further requirement in relatin to this thesis work is that the code is free of external licensing. For that reason both parts will be created from scratch and without the use of third party libraries.

1.5 Ethics

An ethical discussion is not applicable to Reliable UDP as it is a communication protocol and no humans are directly involved with its use. Chord is also a low level technique but ni spite of this, it adds a key technique to applications that make use of it, namely the avoidance of a central server and the distributed storage of information, which has several implications.

As the information is not stored in a central point, it is much more difficult to monitor, manipulate or censor this information. For example, a suppressive nation could easily manipulate a central server instead of a distributed system.

The same is true for the opposite. As the data is spread among several systems of a distributed system, it might be easier for a malicious party to enter this system and spread incorrect information.

1.6 Outline

Chapter 2 describes the nessecary background in order to understand this work. Nevertheless a good knowledge of networking and computing is required. Chapter 3 describes the approach regarding how to solve the milestones defined in the detailed problem statement. Chapter 4 presents the details of the implementation and continues with the results and measurements in chapter 5. Chapter 6 is a conclusion covering the reflections of the thesis work including ideas for future work that have arisen during the project time. This document also includes 3 appendices, consisting of programm in- and output, that are related to descriptions in the text.

1.7 Contributions

In about 50% of the time pair programming was conducted, where one person was programming and the other was checking and contributing orally. The other 50% of the work was splitted into small parts during which the effort ranged from some hours to a day. Because the working units were so small a detailed listing is not useful. From time to time meetings were made, where the current state and a further line of action were discussed.

2 Theory

To understand the following chapters in this report, the reader must have some background knowledge. This required knowledge will be given during this chapter.

2.1 Internet-of-Things

The Internet-of-Things is specifically aimed at two main aspects. The first one is to interconnect all possible sensors and actors over the Internet. These sensors might be of all imaginable types, for example a GPS sensor integrated in a mobile phone, a temperature sensor in a house or a camera looking at the traffic at certain points on the streets. In addition, the actors can be from all imaginable types and following on from the earlier examples, an application that can tell you were your friends are at the moment, an automatic heating regulation or electronic traffic signs that are able to change their speed limit in relation to the traffic situation. All this produces a larger input that can be performed humans using their keyboards or by use of a mouse. This is excatly the reason why the IoT not only aims at the interconnection of everything but also to enable its components to be intelligent. This means that the sensors and actors should show intelligent behavior with regards to their output from other sensors/actors without any human influence (machine to machine communication) [1].

Mid Sweden University has so far developed an approach to build a framework that is able to connect sensors and actors over the Internet and to share information between them. This approach defines nine main requirements in relation to providing an adequate Quality of Service (QoS) and Quality of Experience (QoE). They have also built the first distribution of their framework, which allows for new features to be implemented in the future without a complete redistribution. The interested reader who would like to have more information about MediaSense should read [2].

2.2 Peer-to-peer computing

The term peer-to-peer (P2P) refers to a computer network in which each participant (peer) acts as client and server for the other participants. These architecture enable data to be shared over a network without the reqirement for a central server [3].

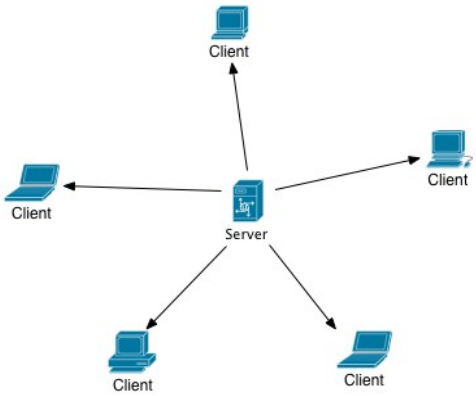

Figure 1: Client-Server model

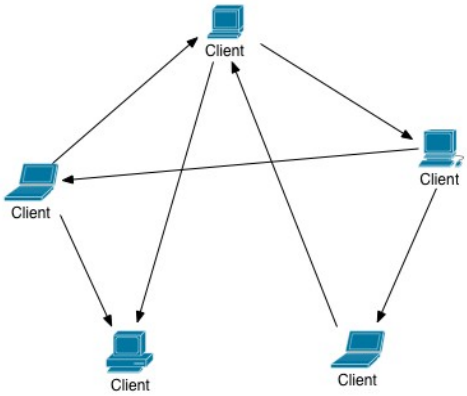

Figure 2: Peer-to-peer

P2P is a distributed application architecture that partitions duties and responsibilities among all its participating peers. Each peer, often also referred to as node, has to set aside a portion of its resources to make it directly available for the other P2P participants. Thus in a P2P architecture, each client has the same rights and responsibilities. In contrast to a P2P network, where each node acts as supplier and consumer, is the client-server-model, in which only servers supply and clients consume the information [3]. Figure 1 and Figure 2 present this in an illustrated way.

2.2.1 Architecture of peer-to-peer systems

Generally P2P systems are implemented as an abstract overlay network in the application layer, not influencing the physical network layer underneath. This overlay performs the indexing and peer discovery and makes the peer-to-peer system independent of the network topology [4].

P2P networks can be further divided into structured and unstructured systems. Structured systems organize their peers and resources using specific algorithms, typically using distributed-hash-tables (DHT) as

explained in chapter 2.3.2. On the other hand unstructured systems do not use any structure in their overlay networks [4].

2.2.2 Advantages and weaknesses

When compared to client-server networks, P2P networks possess a significant advantage, in that there is no single point of failure (i.e. the server in client-server networks). In fact P2P networks are becoming more stable as more clients are participating. Another advantage of P2P networks is, that the total amount of available resources is increasing as more clients join the network, while in client-server networks the available resources per client are reducing, because all clients have to share the resources from the server. However, P2P networks also have some disadvantages. Since each client in the system is responsible for sharing (publish) some of its resources, the whole system is more vulnerable in relation to untrusted or unsigned content. This would not occur in client-server networks because there is a system administrator, who is responsible for publishing the resources [3].

2.3 Hash function

A hash function maps data of variable size to a fixed size hash-value using an algorithm that transforms the data. To be usable for a hash table, the hash function has to be deterministic, which means that it always produces the same output with given input data. The function should also produce values that are uniformly spread over the possible range of hash values so as to reduce hash collisions. A collision occurs when two different input keys produce the same output [5].

An important property of a hash function is the output length, specified in bits. A function with n bits output length produces hash values in the range of $0-2^n-1$ and the amount of possible values is 2^n [5].

2.3.1 Hash table

A hash table is a data structure that stores key-value pairs. The value can be any user-specified value that should be associated with the key. A hash function is used to transform the key into a hash-value that is then used as an index, which indicates where the associated user-value is stored. The use of a hash function introduces the possibility of hash

collisions. There are several ways of dealing with a collision situation, and the easiest way is by denying the colliding key [5].

The hash-table used in this project does not use advanced collision handling techniques, so they are not described any further in this document.

2.3.2 Distributed Hash Table

A distributed hash table (DHT) is the same as a hash table, except that it is distributed among several parties, and in which each party is responsible for a certain amount of the hash space. In the sense of P2P, a party means one connected client in a distributed system. [5]

To partition the DHT, each node must have a unique identification key from the hash space that is either random or derived from unique data. [5]

Every node of a DHT can store and retrieve values in the DHT and must be able to handle requests when the requested key is in its own space of responsibility. The DHT nodes have to be connected to each other in some way, to be able to interchange requests [5]. There are many different possibilities for splitting up the hash space into smaller parts, as CAN, Tapestry, Pastry, Chord and others [6]. This project deals with Chord, which is described in more detail in chapter 2.3.3 .

2.3.3 Chord

The Chord lookup protocol arranges its participating nodes in a virtual circle with a maximum number of nodes limited by the amount of possible values of the used hash-function [7].

Each node is responsible for all hash values between its own key and the key of its successor minus one: [id, id_successor). As nodes enter, leave or fail, the range of responsibility changes continuously and the key-value pairs have to be shifted in order for the DHT to remain valid [7].

To be able to forward requests, and to know the responsible range of hashes, each node must at least know its succeeding node. In this case a message between two random nodes would take $n/2$ steps, on average,

because every node can only forward it to its successor and the average traversal distance is half the circle. That means that the effort in finding a position increases linearly with the size of the DHT. This effort can be expressed as $O(n)$. To reduce the effort, a finger table is used. Figure 3 gives a graphical overview of such a finger table [7].

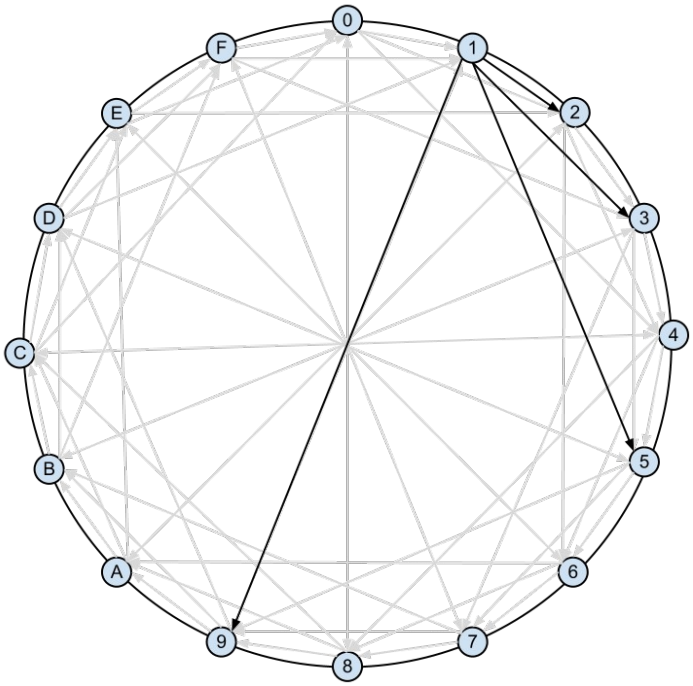

Figure 3: Chord example with 16nodes showing the finger table of node "1"

Each node in a chord DHT contains a list of links to other nodes of the DHT. This list is also referred to as a finger table. The finger table speeds up the query process. The definition of Chord requires the finger table to point to the succeeding nodes of the hash value calculated using the following Formula 1:

$$(n+2)^{i+1} \bmod m \qquad \textit{Formula 1: Finger entry [7]}$$

where n is the key of the node, i is the finger table entry and m is the size of the hash-value.

That means that a node can have up to n finger entries (where m is the maximum value of the hash function). Using this method the effort relating to finding a position is at maximum $O(log2(m))$ because, after

each traversal step, the distance remaining in which to find a key is decreased to half of its original value [7].

The successor, which is also the very first finger in the finger table, is the directly succeeding node in the DHT. If the finger table contains any incorrect links then the efficiency decreases. Despite that, the direct successor has a special role as it is important to have the DHT circular and not containing side-loops, intersections or orphaned nodes, which could cause queries to fail. Thus, check and repair mechanisms are used to keep the correct successor, which will be explained at a later point in this section.

A joining node has to perform the following steps to safely enter the DHT [7]:

- query the position it is supposed to be
- ask the predecessor of that position to update its successor to that of the new node
- ask the predecessor for its last successor, which becomes the new successor of the new node
- the preceding node has to register the key-value pairs that it is no longer responsible for to the new node
- start a finger table update process at the new node

A node that wants to leave the DHT conventionally has to perform the following steps [7]:

- register all of its key-value pairs to the predecessor
- send the succeeding node to the predecessor so that this node can form a circle again

Every node should perform checking and repairing in order to remain functionality and stability of the DHT. This can be, but is not limited to, the following list [7]:

- ask for the predecessor of the succeeding node and check if it is the asking node

- regularly update the finger table entries to maintain the query time at $O(log2(n))$

Every node in the DHT can query the predecessor of a position in the DHT. This query is either forwarded to the closest preceding node in the finger table, or, if a node finds itself being the predecessor of that position, answering it directly [7].

Chord itself does not specify a method for broadcasting. But in [8] an efficient approach is demonstrated. If a node wants to initiate a broadcast it is seen as the root of a spanning tree that covers all nodes in the DHT. The broadcast process uses unicast messages that originate at the root node and are propagated from other nodes until every node has received the broadcast. Broadcast in this case must not be confused with network layer broadcast as used in TCP / IP for example [8].

To avoid redundancy the broadcast message contains a limit value that defines the upper key in the DHT to which the broadcast will be forwarded. The lower limit is the propagating node itself [8].

Assuming that Node 0 in Figure 3 wants to initiate a broadcast, the range is the whole circle from its key to its key minus one. The first step is that Node 0 sends messages with the following limits to its fingers:

- Node 1: [1,2) - covering one 16th of the DHT

- Node 2: [2,4) - covering one 8th of the DHT

- Node 4: [4,8) - covering one 4th of the DHT

- Node 8: [8,0) - covering one half of the DHT

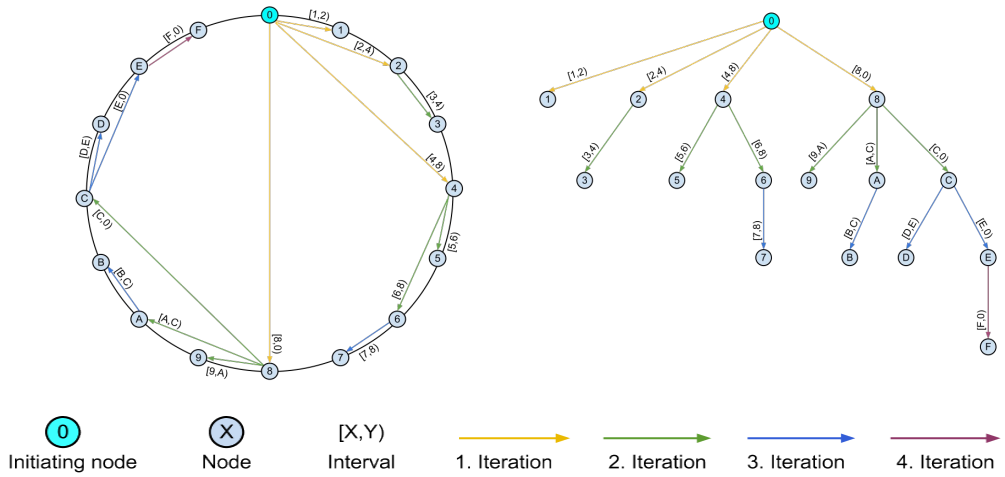

Figure 4: Broadcast algorithm

The receiving nodes forward the message to all fingers that are within the range of their own key and the received limit, without changing the limit. See Figure 4 for a visualization of the message propagation in several steps. The figure shows the propagation of the message in the circle (left) and also shows a visualization of the propagation in a tree structure (right) that is easier to understand.

All ranges combined, cover the whole DHT excluding the originating Node 0. If multiple fingers point to the same node, because the DHT is not complete, the message is merely send once in order to avoid duplicates. If a finger is missing then the broadcast will still be propagated correctly (assumed that the DHT is a correct continuous circle), but the efficiency will decrease [8].

2.4 Network data transmission

Typically network connections protocols are placed into two categories which are packet oriented and connection oriented [5].

A packet oriented protocols sends packets from a source to a destination. The path taken by the packets is not predetermined and can change from packet to packet. If more packets are sent, then the order can change on the way to the destination. Packets can also become lost without any notice. It is up to the application programmer to add meta information to the packet data, depending on whether a correct order and reliable transmission are required [5].

The User Datagram Protocol (UDP) is one famous packet oriented data transmission protocol [5].

A connection oriented protocol does not handle data as separate packets but as a continuous stream of bytes. To enable this, some arrangements have to be taken to overcome data reordering, corruption, and loss of data [5].

Usually a connection oriented connection can be split into 3 stages [5]:

1. Connection establishment - The sender contacts the receiver and attempts to set up connection parameters and synchronization. If the receiver agrees a virtual link is established between both sides.

2. Data transmission - Both sides can send and receive data during this phase.

3. Connection shutdown - Onec a side decides to shut down the connection link, it informs the other side converning that decision and both start a shutdown process in order to empty the buffers and stop sending data. After that the virtual link is considered to be closed. No data can be transmitted beyond that point.

Reliable data transmission uses several techniques to detect and correct errors. Those considered to be important in relation to this project are explained in the following sections.

2.4.1 Cyclic Redundancy Check

A CRC algorithm calculates a value of fixed length from data of variable length (similar to a hash function). The CRC value is unique for given data and small changes to the data should lead to totally different CRC values [5].

Before transmission ,the CRC value is calculated and attached to the data. After reception at the destination, the CRC value is calculated again and compared to the sent data. If both values differ, the data or the sent CRC value have been altered during transmission and the data can be considered as erroneous [5].

The precision to detect one or more errors depends on the length of the calculated CRC value. Assuming the CRC value consists of n bits, the probability of detecting an error is (2^n-1) and the probability of missing an error is $1/(2^n)$ [5].

2.4.2 Window flow control

The task of flow control involves not overload the receiving side of a communication link with data. This means that the sender stops sending information after a certain amount of sent data or packets and waits for information from the receiver to provide information regarding a continuation [5].

Window in this sense means that the receiver has a window in the form of a buffer that the sender can send data to. For this to work, every data unit requires a sequence number that identifies its position in the data stream. The receiver has to make sure that at least it is able to receive as many data as would fit into the window buffer [5].

A typical implementation is a moving window that has a start point and a length. The sender starts to fill the window with increasing sequence numbers but only up the point that the window length is reached. The receiver consumes received data in ascending order and then shifts the window to the point of the last unconsumed data unit. Hence the window becomes free for additional data to be received. The sender has to be informed about the window shift so that it knows when it is allowed to continue data transmission [5].

2.4.3 Reliable UDP

The term Reliable UPD (RUDP) originally refers to a simple packet based transport protocol that was defined to transport telephony signaling across IP networks. The IETF defined in their draft the following criteria [9]:

- *"transport should provide reliable delivery up to a maximum number of retransmissions (i.e. avoid stale signaling messages)."*
- *"transport should provide in-order delivery."*

- *"transport should be a message based."*
- *"transport should provide flow control mechanism."*
- *"transport should have low overhead, high performance."*
- *"characteristics of each virtual connection should be configurable (i.e. timers)."*
- *"transport should provide a keep-alive mechanism."*
- *"transport should provide error detection."*
- *"transport should provide for secure transmission."*

It was further defined that RUDP should be defined in such a way that it is easily possible to allow different characteristics for each connection so that the header length could be maintained as short as possible [9].

3 Methodology

The project consists of two separate parts. Part one is the development of a Distributed Hash Table (DHT) and part two involves the development of a Reliable User Datagram Protocol (RUDP). The approach for solving each milestone as defined in Chapter 1.3 will be described in this chapter.

All the development is conducted using the Eclipse [10] Integrated Development Environment (IDE) and all project related content will be shared over a Git repository [11].

3.1 Develop a simulation environment for DHT clients

The simulation environment should have an identical API to that of the MediaSense platform so as to ease the later transfer of the implemented DHT. Furthermore, the simulation environment should enable the following operations:

Host a theoretically unlimited amount of DHT clients (limited by resources only)

- Allow clients to join the network
- Allow clients to leave the network
- Simulate network connection delays and failures
- Regularly record client and DHT properties for statistical evaluation
- Monitor messages exchanged between clients
- Allow for the listing of detailed client information such as finger-table, successor, predecessor, internal state etc.
- Visualize the DHT in a Graphical User Interface (GUI)

3.2 Implement a circular DHT

As described in the background chapter, a DHT is a virtual space of addresses containing at least one client. Each client is responsible for the address range between its own address and its successor's address. In addition, each client has exactly the same responsibilities and rights in the DHT. The following list shows the functionality that each DHT client should have:

- Insert a joining node as the new successor if the client's address is between the current client's and successor's address
- Forward queries to the client in the finger-table whose address is closest to the destination address
- Maintain a list of fingers (finger-table) to other clients in the DHT
- Detect and handle unreachable clients
- Execute self-checks regularly and correct wrong links if necessary
- Initiate and forward broadcast messages

Generally, each query to a client that is participant of a DHT should return an answer.

3.3 Evaluate the performance of the DHT

To produce comparable characteristics the DHT solution should be evaluated as specified in the following list. The comparison to other implementations is not part of this project. Note that N represents a different amount of nodes.

- Measure the correctness of the DHT in the following situations
    - N nodes joining
    - N nodes leaving
    - N nodes failing

- Measure average transmitted data per node in the following situations
    - N nodes joining
    - N nodes leaving
    - N nodes failing
    - Idle DHT with N nodes

3.4 Transfer the DHT solution to the MediaSense platform

After successful simulation and evaluation, the DHT should be transferred to the MediaSense platform. Although the API is the same it must be verified that the DHT still fulfills the desired functionality. The DHT will be tested by installing the MediaSense platform on several computer systems inside the University network.

3.5 Develop a simulation environment for communication protocols

The simulation environment should have the identical API as the MediaSense platform in order to ease the later transfer of the implemented RUDP. Furthermore the simulation environment should enable the following operations:

- Host two MediaSense communication instances
- Differenciate between the sending and receiving side's.
- Check the transmitted data for errors
- Record statistical data during data transmission
- Simulate network delay, packet corruption, packet loss, packet reordering and packet duplication

3.6 Develop a reliable packet-based communication protocol

The RUDP protocol should be a reliable and packet-oriented communication protocol using UDP as the underlying transmission protocol. The protocol should have the following properties:

- Connectionless (no connection establishment and shutdown needed)
- Detect and request the re-sending of corrupted and lost packets
- Detect and drop duplicate packets
- Reorder disordered packets if possible - request re-sending otherwise

3.7 Compare RUDP with TCP

As TCP is a sophisticated protocol that has been fine tuned during its years of use, comparision to this protocol appears reasonable in order to dertermine the strength and weaknesses of RUDP. Within the simulation environment both protocols will be compared in relation to the following disciplines:

- transmission speed in error free environment
- transmission speed in erroneous environments
- transmission overhead

3.8 Transfer RUDP to MediaSense

After a successful simulation, the RUDP protocol should be transferred to the MediaSense platform. Although the API is the same, it must be verified that the protocol still fulfills the desired functionality. RUDP will be tested by installing the MediaSense platform on several computer systems inside the University network.

4 Implementation

This chapter describes the implementation process of the goals defined in Chapter 1.3. Figure 5 illustrates this process as a reflection of the eight milestones. A sub-chapter to each defined goal describes the important points in detail.

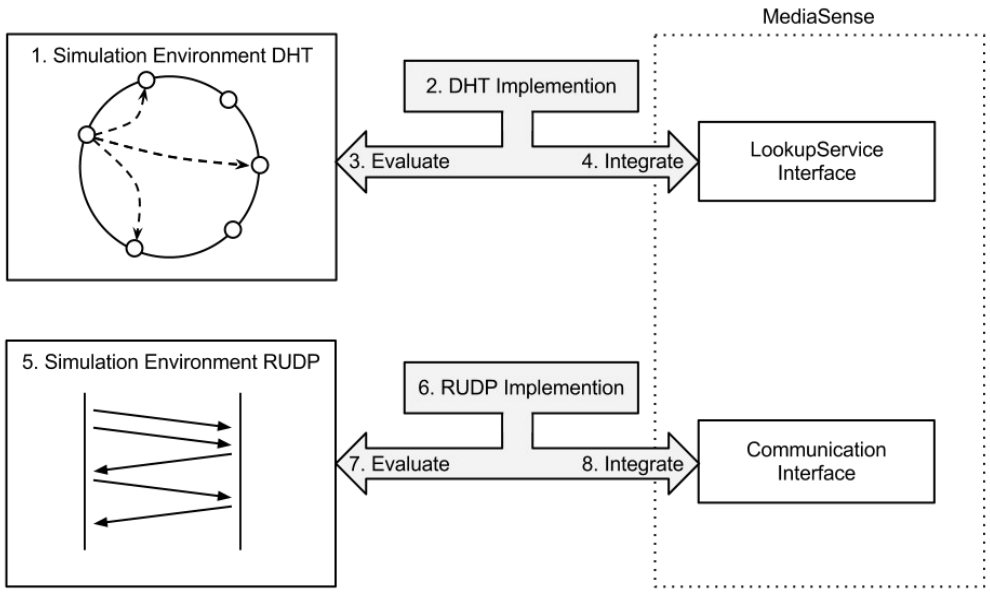

Figure 5: Workflow

4.1 Develop a simulation environment for DHT clients

Since one of the defined goals is to integrate the developed DHT solution into the MediaSense framework, the simulation environment is built upon the same interfaces as used by the framework. Figure 6 shows a rough class diagram with regards to how the simulation environment is structured and how the functionality is split up among several classes.

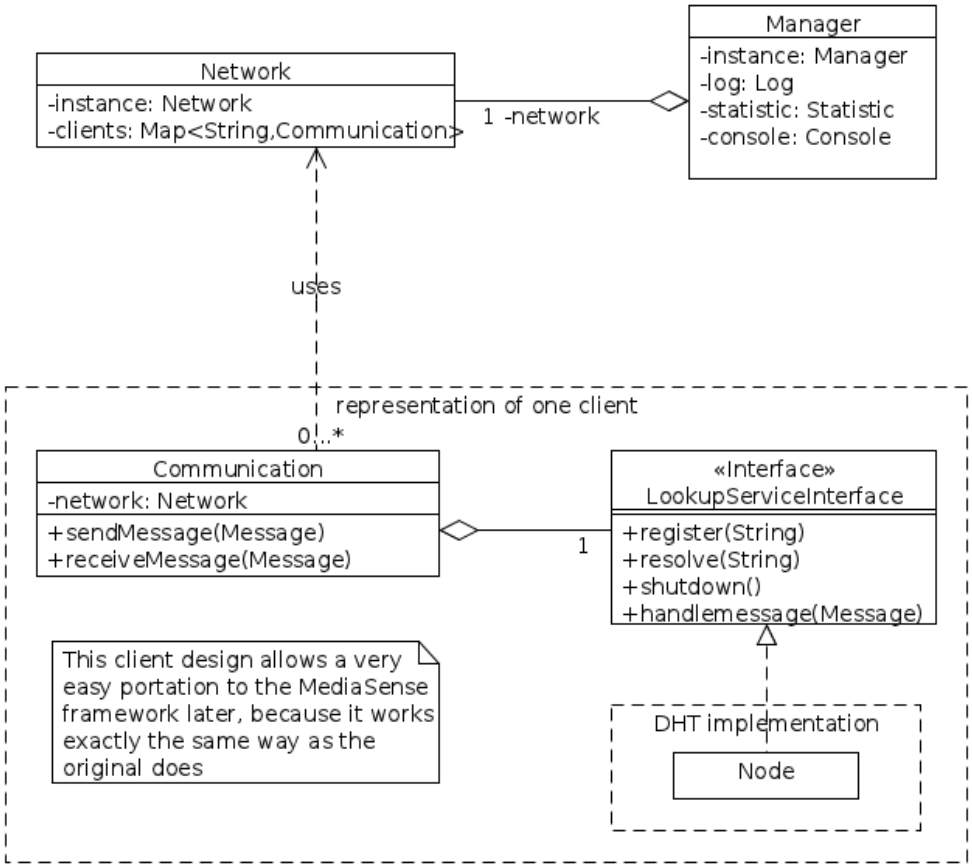

Figure 6: Class Diagram

4.1.1 Manager

The Manager class is the main entry point to the simulation environment. Here, everything is tied up, which in fact means, the debug log is started, the statistic for performance measuring is controlled and the commands from the console are forwarded.

4.1.2 Client Simulation

To simulate exactly the same pre-requirements that follow from the MediaSense framework, each client consists of an Communication object and a LookupService object running separately in independent threads. The Communication object is responsible for sending (send means, receive from the LookupService and forward to the network layer) and receiving (receive means, receive from the network layer and forward to the LookupService) messages, and can be viewed as the glue between the network and the DHT implementation.

4.1.3 Network

Because the development of the lookup-service is independent of the physical network layer, the simulated network is a fully reliable message forwarding service between the nodes. It allows for different communication delays to be defined for each node or for the whole network. To be able to forward messages to all clients, the network retains a map with Communication objects being related to their addresses. Because it is only possible to have one network, this class is implemented as a singleton.

4.1.4 Console

The Console class is the main interface for the user to control the simulation environment. The most important functionality is that clients can be added or removed from the network, sensors can be registered and resolve attempts can be started. A complete overview of all commands and a short description is placed in Appendix A.

4.1.5 Graphical User Interface

From the console it is possible by using the command "g" to start a graphical user interface. The GUI is built from basic Java Swing components, and is maintained in as simple a manner as possible. In addition, the GUI draws a visualization of the hash space and how the clients act in this space, therefore Java AWT Graphics classes are used.

4.1.6 Log

The Log class writes a text file with all possible debug outputs from the simulation environment. The debug outputs include the following points:

- Joining, leaving and disappearing nodes
- All messages forwarded by the network.
- Keep-alive initiations
- Finger changes

An example log file is placed in Appendix B

4.2 Implement a circular DHT based on Chord

Figure 7 shows a finite-state machine (FSM) which illustrates the life-cycle of a node. It should be mentioned that this FSM is only a brief mapping of the real implementation, but nevertheless it shows the most important functionalities.

The state q0 is the entry point when a node starts and q5 is the final state, which will be reached when a node shuts down. A node starts the connection by initiating a three way handshake (q0, q1, q2, q3). If a node is in the connected state it can itself accept connection requests from other nodes and include them into the DHT (q3, q4, q5).

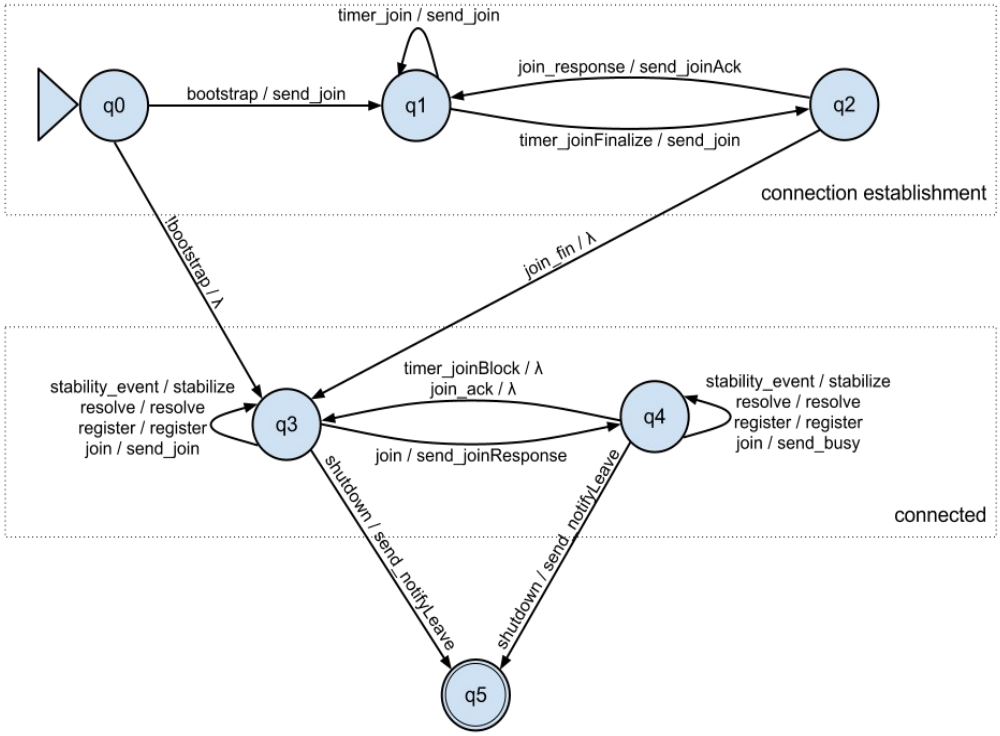

Figure 7: Finite State Machine for a node

4.2.1 Join process

The join process is illustrated in the box "connection establishment" in Figure 7. Starting in q0 there are two different possibilities. The first one is, that the node has no bootstrap address and therefore acts itself as a bootstrap node (create a new DHT instance). If this is the case, the node immediately changes to the connected status and the connection establishment box is departed from. The second possibility is, that the

node obtains a regular bootstrap address. In that case, the node will attempt to establish a connection to an already running DHT instance. If the connecting node does not receive an answer in a specified time, it will start a new attempt to join the DHT.

If a node is responsible for another node to join the circle, it is "blocked" for this process and is unable to insert other nodes until the insertion process has finished. This is shown in Figure 7 in the box "connected".

4.2.2 Leave process

There are two possible ways in a DHT as to how the nodes might finish their availability. The regular way is, that the leaving node sends a NOTIFY_LEAVE message to the DHT in order to inform all other nodes about its disappearance. However, naturally, there are also problems such as failing links when the node is not able to inform the DHT about its disappearance. In that case, the first node that recognizes a failing node, has to send a NODE_SUSPICIOUS message to all participants in the DHT.

4.2.3 Message types

The DHT implementation uses 3 different categories in relation to propagating messages. Each category contains several message types, and for which each type can contain additional data that is associated with that type.

Some messages are directly sent to another node without being further forwarded. The following messages are direct node to node messages:

- REGISTER_RESPONSE - Response from a node, which registers a key-value pair that it is responsible for, to the initiating node
- RESOLVE_RESPONSE – Response from a node, which knows the location of a registered sensor, to the querieng node
- JOIN_RESPONSE - Response from the predecessor of a new joining node that the new node can enter the DHT directly after the predecessor

- JOIN_ACK - Acknowledge that a new joining node has accepted the response and accepted the predecessor's former successor as its successor

- JOIN_FINALIZE - Last message in the 3-way handshake to join the DHT that informs the joining node to consider itself to be regularly connected

- DUPLICATE_NODE_ID - Response to a JOIN-message if the joining node attempts to join with an already existing hash value

- FIND_PREDECESSOR_RESPONSE - Response from a responsible node to a FIND_PREDECESSOR-message to find the predecessor of a certain hash-value

- CHECK_PREDECESSOR - Request the predecessor of the destined node

- CHECK_PREDECESSOR_RESPONSE - Response to a CHECK_PREDECESSOR-message, containing the predecessor of the answering node

- CHECK_SUCCESSOR - Request the successor of the destined node

- CHECK_SUCCESSOR_RESPONSE - Response to a CHECK_SUCCESSOR-message, containing the successor of the answering node

Another possible message type involves the query messages. Query messages can be sent to any node in the DHT regardless of whether a node is responsible or not. The message is either forwarded to a node of the finger that is closest to the queried destination or answered if the receiving node is responsible for the destination. For this reason every message must contain at least a hash value (key) that represents the destination position. The following messages are query messages:

- REGISTER - Query to register a key-value (usually a sensor / actor) pair in the DHT

- RESOLVE - Query to resolve a value associated with a key stored in the DHT
- JOIN - Query of a new node to enter the DHT
- FIND_PREDECESSOR - Find the predecessor of a key specified inside the message

The last possible message type involves broadcast messages. A broadcast message is a container that can encapsulate any other message. As described in chapter 2.3.3 a broadcast message requires a limit which is the maximum hash value to which a certain node should forward a message. The initiating node sets these limits according to the position in the finger table. For this to work, a broadcast message must contain at least a start-key and an end-key to specify this range. For a normal broadcast the start-key is always the initiating / forwarding node. However, to be able to send multicast messages in the future, a start-key field is also present that contains the redundant information of the current node at the moment. The following messages are used for broadcast propagation:

- BROADCAST - Container itself; Only contains the start-key, end-key and the encapsulated message
- KEEPALIVE - Message that is regularly sent to check the network connection and to advertise nodes for finger table upgrading.
- NOTIFY_JOIN - Sent by a node after the join process to inform all other nodes of its arrival
- NOTIFY_LEAVE - Sent by a node before the shutdown process to inform all other nodes of its planned leaving
- NODE_SUSPICIOUS - Sent by a node that has encountered a network problem with another node to inform all other nodes in the DHT of that

4.3 Evaluate the DHT solution

This section explains the functionality that has been built to evaluate the DHT solution. Additionally, testing the DHT implementation in the simulation environment also forms part of the evaluation.

4.3.1 Health

The health of the DHT can be measured using the `health` function. These functions return a value between 0 and 1, where 0 represents the worst case and 1 represents a perfect DHT. The manager class is able to calculate which node must have which fingers in its finger table and checks how many fingers are missing or not at the correct position. Based on that information, the health calculation is built.

4.3.2 Statistic

With the command `statistic file-path`, it is possible to write a statistic to the specified file. The statistic is triggered once per second and writes the following data points to a matrix in the specified file. For an example statistic see Appendix C.

- Timestamp: A timestamp when the data in this line has been collected
- Sec: Absolute seconds since the statistic has started to collect data
- Hth: The health of the DHT
- Con: Currently connected nodes in the network
- Fin: Fingerchanges that have happened
- Data: Data that has been submitted over the network
- Pkt: Number of messages that have been submitted over the network
- ConD, FinD, DataD, PktD: See upper definition, but as delta since the last second

4.4 Transfer the DHT solution to the MediaSense platform

As the API of the simulation environment was adopted from the MediaSense platform, moving the source files was sufficient despite the removal of the debug functions. These were designed in such a way that they could be easily removed without influencing other program parts. The existing MediaSense framework already has a layer for lookup-services since it already has a client-server solution. Regarding this, a new package has been constructed into which the source files have been inserted.

4.5 Develop a simulation environment for communication protocols

Since another goal defined in chapter 1.3 is to integrate the developed RUDP solution to the MediaSense framework, the simulation environment is built on the same interfaces as used by the framework.

The hardware setup consists of 2 PCs connect by a router. The network bandwidth was 100 MBit/s. One PC was the sender, and the other the receiver of random data packets of 1024 Bytes length. The first 4 bytes were replaced by a continuous check sequence starting from 0. The receiver checked the sequence for continuity to detect transmission failures. Each measured data point is made up of a new transmission of 200000 packets leading to a total amount of 200 MB. Figure 8 shows a visualisation of the setup.

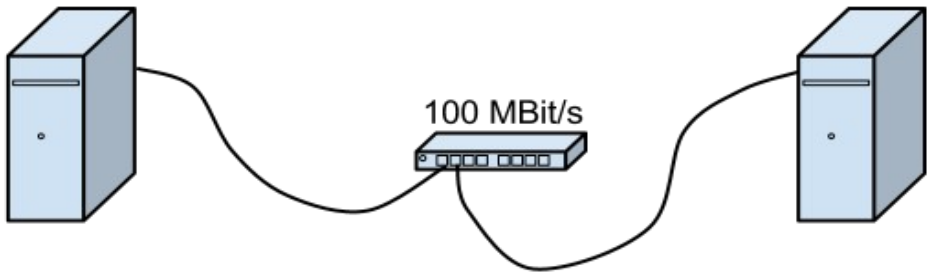

Figure 8: Hardware setup reliable udp simulation

4.6 Develop a reliable packet-based communication protocol

Figure 9 and Figure 10 show to two finite state machines describing the behaviour of the receiver and sender. Both are firstly required to

receive / send a FIRST flagged packet to have knowledge of their states and to proceed to a synced state. The FIRST packet can already contain user data. After this, synchonisation packets can be sent and received as described in the following sections. A RESET packet can be used to reset the SENDER if the synchronisation state was unequal. If the link is not used after a certain amount of time it is shut down.

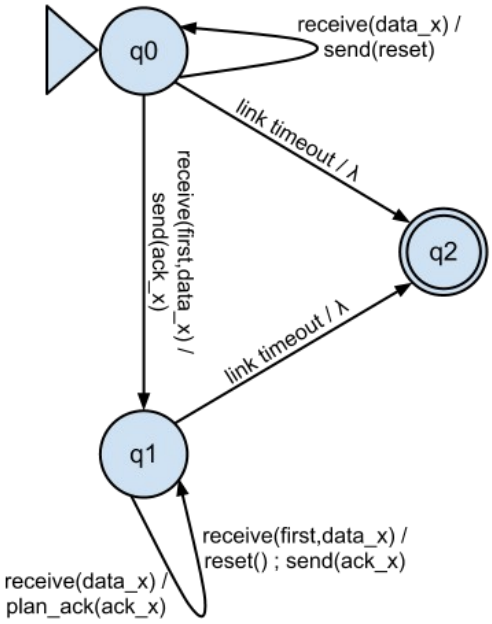

Figure 9: RUDP receiver

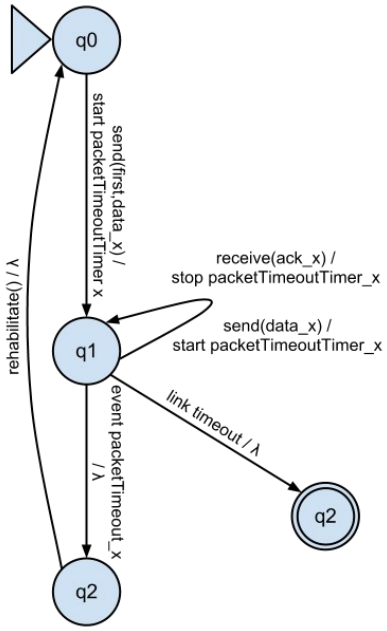

Figure 10: RUDP sender

4.6.1 Packet structure

Each RUDP packet consists of several fields as shown in Table 1. The function of each field is described below the table. Not every packet has the same structure because some parts (i.e. the acknowledgement and data fields) are optional.

Name	**Offset (Bytes)**	**Length (Bytes)**	**Type**
Flags	0	1	Bit field
Packet sequence	1	4	Integer
Window size	5	4	Integer
Fragment number	9	2	Short
Fragment count	11	2	Short
Acknowledge sequence	13	4	Integer
Acknowledge field size	17	2	Short
Acknowledge field	19	n * 2	Short-array 0 < n < 64
Data	13 - 83	0 - 65362	Byte array

Table 1: RUDP packet fields (*Italic* fields are optional)

- Flags (8 bits) - bit field containing flags
  - First (bit 0) - first packet containing application layer data
  - Reset (bit 1) - request to reset sender state
  - ACK - (bit 2) - packet contains acknowledge information
  - Data (bit 3) - packet contains application layer data
  - Resend (bit 4) - packet was resent
  - Fragment (bit 5) - packet is fragmented
  - Persist (bit 6) - request an acknowledge packet
- Packet sequence - continuous number that identifies each packet
- Window size - specifies the window size of the sender's receiving window
- Fragment number - specifies the fragment number of a fragmented packet from 0 to fragment count – 1

The following fields are only present if the ACK flag is set:

- Acknowledge sequence - specifies the start sequence of the acknowledged window
- Acknowledge field count - specifies the amount of acknowledge regions
- Acknowledge field - contains the acknowledged regions

The following field is only present if the Data flag is set:

- Data - contains application layer data

4.6.2 Data integrity

RUDP is based on UDP which uses a CRC value to ensure data integrity. The CRC value is 16 bits long and is placed in the UDP header. In the case of a corrupted packet, the underlying layer will automatically drop this packet, so no additional checksum field has been implemented in RUDP.

4.6.3 Link synchronization

Each packet used to transport user data, marked with the DATA flag, is associated with a sequence number that identifies this data. The packet number is valid between two communicating hosts, also called a link. Each sender has its own sequence number. The very first packet that is sent on a link is marked with the FIRST flag and contains the first sequence number chosen by the sender. Every packet, that follows the first packet, must have a sequence number increased by 1. Using this mechanism the receiver is able to recover the packet stream, if packet loss or reordering occurs. The range of sequence numbers is $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$. So care has to be taken on the overflow that occurs at the maximum value. RUDP does this by never using absolute comparisons.

Both, sender and receiver, have to be in a well-known state when the data transmission starts. In reality this means that a receiver must know the current sequence number of the sender, and the sender must know

the state of the receiver window start and its size. The first data packet on a link is marked with the FIRST flag. A new receiver must always obtain a FIRST flagged packet at the beginning and takes the used sequence number as a reference for all proceeding packets. A new sender assumes the receiver's window size is 1 and is thus only allowed to send one packet at the beginning, that is marked with a FIRST flag. The receiver then places its receiving window start to that sequence number and sends an ACK packet immediately, which contains its own window size. Figure 9 and Figure 10 offer an easier understanding.

Unsynchronized situations can appear for which the knowledge about the other state is not consistent. If the receiving side of the link receives a FIRST flagged packet when it does not assume one or receives an unflagged packet when it assumes a FIRST packet, it resets itself and sends a packet with the RESET flag set. After reception of a RESET the link must reset its state and behave as a newly created link.

Every sent packet has to be acknowledged implicitly or selectively, as described in the following paragraphs. Packets can be acknowledged in groups and acknowledgements are sent in the following situations:

- If the receiver buffer is less than 2/3rds full: 100 ms after a packet has been received. The 100 ms wait time is used to collect more packets and thus reduce the number of ACK packets
- If the receiver buffer is either full by more than or equal to 2/3rds: immediately after a packet has been received
- If the receiver buffer was completely full and gained new space
- If a packet with a PERSIST flag has been received

Figure 11 should be investigated after each of the following paragraphs. This will offer the opportunity to understand the different acknowledgement possibilities and the shifting of the window more easily.

If the receiving side of a link has received and consumed data packets, it shifts its own receiving window to the next unconsumed or unreceived

packet. The next outgoing ACK packet contains this new window start sequence. Every packet that lies between the old and the new window start is implicitly acknowledged as having been successfully received.

Received packets can also be acknowledged selectively by the receiver. This occurs when the receiver window contains gaps with unreceived packets or the packets have not yet been consumed and still block the receiver window. The *acknowledge field count* and *acknowledge field* are used to carry the selective acknowledge data. *Acknowledge field count* specifies the number of the following acknowledge fields (possible amount is 0 to 64). All *acknowledge field* data is relative to the *acknowledge window start* value. The format of the *acknowledge field* is as follows:

It contains *acknowledge field count + 1* values of the type short. Number n to number $n+1$ specifies a range of [n, n+1). Every even range specifies the acknowledged packets and every odd range specifies the unacknowledged packets.

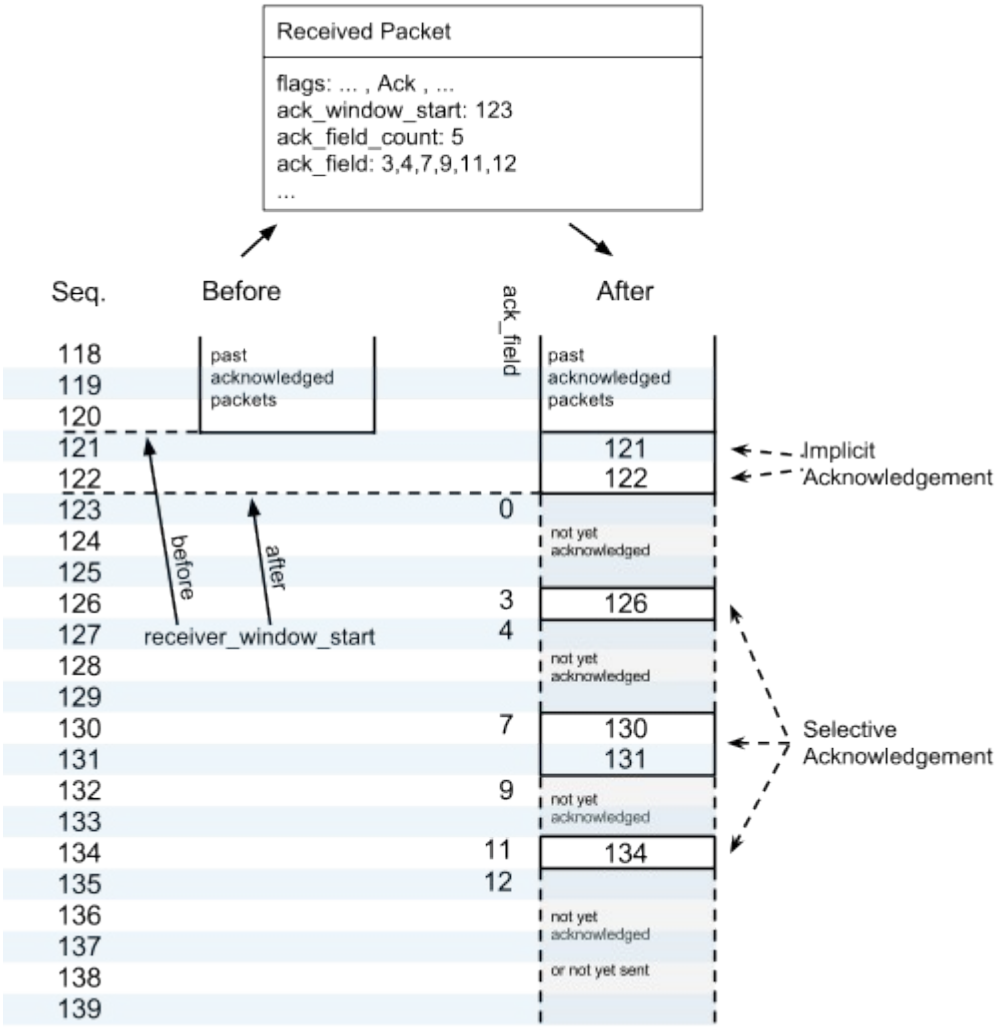

Figure 11: Example of receiving an acknowledgement packet

If a packet is not acknowledged implicitly or selectively after a fixed amount of time then the packet is resent and the wait timer for this packet is started again with the wait time doubled. A link is considered as having failed if a packet is not acknowledged after 5 send attempts. The sender is then put into a link-failed state. Each send attempt on that link will then lead to an exception of the type *DestinationNotReachableException*. The link can be reset using the rehabilitate() function. The link is then put into an initial state and all buffers are cleared. Failed packets have to be resent by the user application.

When the receiver window is full, the receiver informs the sender when the buffer contains free space again in order to receive more packets. If this ACK packet becomes lost, a deadlock situation can occur. To prevent this a persist timer is started at the sender when the last, buffer filling, packet was sent. After expiration of the timer a packet with the PERSIST flag is sent and the receiver answers with an ACK packet which provides information about the buffer state. The persist packet has the same resend behavior as a normal data packet, but does not contain data itself. If the persist packet is not acknowledged or the receiver buffer is not freed within the persist packet transmission attempt, the link is considered as having failed and put in a link-fail state.

4.6.4 Flow control

Every link consists of a sender and a receiver part. The receiver has a receiving buffer (receiver window), whose size can be decided by the client as deemed appropriate. The sender sends data "into" this window specifying the current position with the sequence number and is only allowed to stay within window bounds. The receiver can shift the window towards the positive direction if it received and consumed packets. The sender is informed about the new window position through the acknowledge sequence field in ACK packets. See Figure 12 for an example of receiver window action.

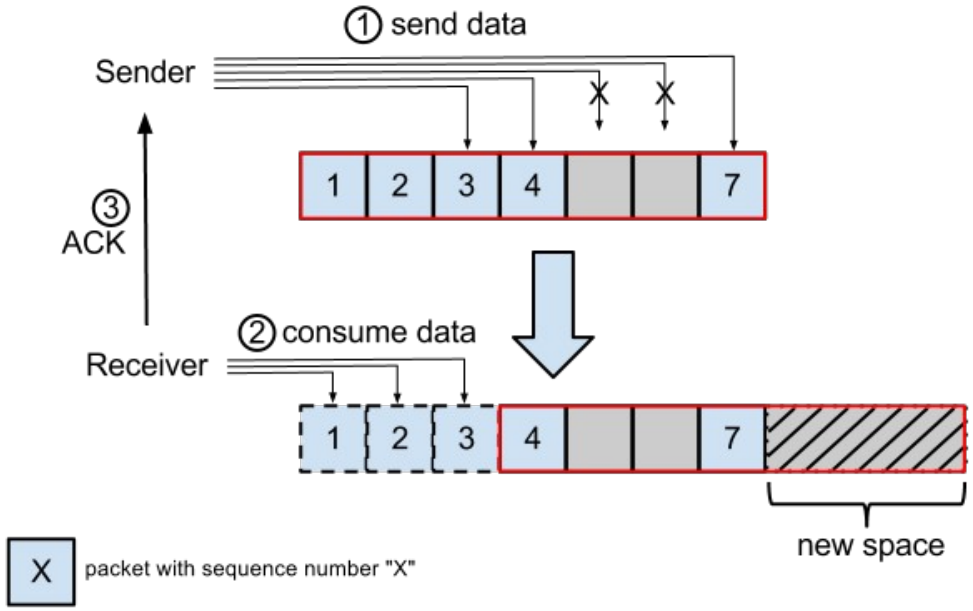

Figure 12: Visualization of the window shifting process

The client has to consideration the window size. A smaller window consumes less memory and a larger window allows more unacknowledged packets and less reaction time.

4.7 Compare RUDP to TCP

The simulation environment has been used for the evaluation. Two RUDP sockets are connected to each other over an isolated network. The reliability has been checked by transmitting random data in one direction between the 2 sockets and comparing the correctness and order of the data after its reception.

To test bad conditions in a reproducible way the Linux tool `tc` (traffic control) was used in order to simulate packet loss, packet reordering, and packet delay. This tool changes kernel network parameters for outgoing packets.

4.8 Transfer RUDP to MediaSense

As the API of the simulation environment was adopted from the MediaSense platform, only the class files had to be moved. For performance purposes the debug functions have been disabled before

integration. For verification, the MediaSense software has been tested with RUDP as the underlying protocol for the dissemination layer.

4.9 Additional changes

Some improvements have been made in addition to the mentioned milestones. The improvements have been implemented during the DHT implementation phase. The need arose because some design issues were discovered that should not be adapted by the implemented software.

4.9.1 Message serializer interface

The source and destination addresses are now passed as arguments to each message serializer as this information is available from the context of the underlying network layer. In addition, it is not necessary to include the source and destination addresses in the application layer protocol as they are always available from the network layer.

4.9.2 Binary message serializer

In addition to the "EnterSeparatedMessageSerializer" a binary message serializer was implemented in order to provide an efficient protocol without any redundancies. See Figure 13 and Figure 14 for details of the message fields for unicast and broadcast messages respectively.

Bit offset	0-7	8-31
0	MAGIC_WORD	
32	Type	Begin of user defined data

Figure 13: Unicast packet structure

Bit offset	0-7	8-31
0	MAGIC_WORD	
32	Type	start-key
256	...start-key	end-key
480	...end-key	Begin of user defined data

Figure 14: Broadcast packet structure

5 Results

This chapter describes the results of the project. These include the output of both the implementations that have been made. The main focus of this chapter relates to the results of the measurements and evaluations.

5.1 Chord simulation environment

The simulation environment contains two input / output methods. The first is a command line that allows every possible action on the DHT and is able to monitor events of the DHT. The second is a GUI whose main purpose is to visualize the structure of the DHT in a graphical way. The GUI contains only a basic set of DHT actions.

The command line takes commands, each of which has a different amount of parameters or no parameters. A command is always one continuous word. If parameters follow, they are separated by a comma from each other and with a space from the command. The output is printed to the same command line. For a complete reference of all commands and their parameters see Appendix A. The functions of the command line are as follows:

- Add, remove and kill nodes
- Watch events such as transmitted messages, node changes and finger changes
- Print node information such as fingers, id and network address
- Print the health of the DHT
- Control other simulation functions such as the GUI and the statistics function

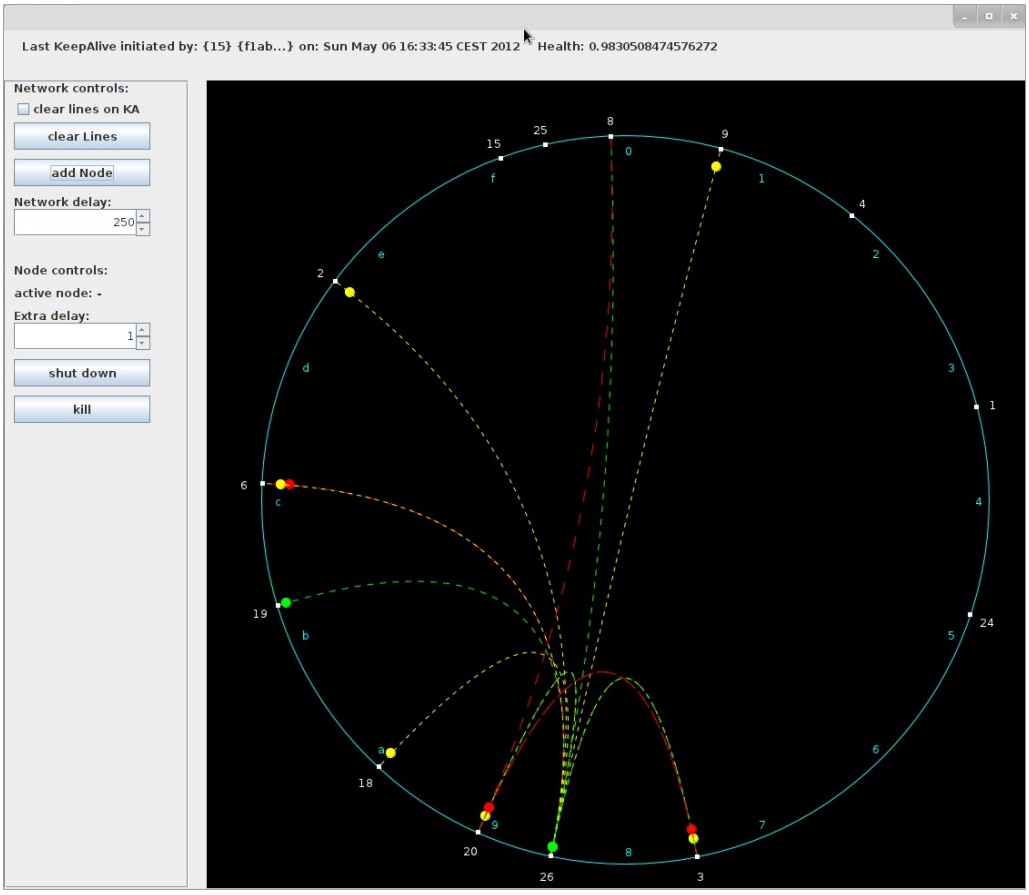

Figure 15: Screen-shot of the GUI

In the GUI, the DHT is visualized as a circle that represents the hash space of the DHT. Every node is marked as a small rectangle on the position of its hash value on the circle and the network address attached to it. By hovering over a node with the mouse the current finger table of that node is displayed as purple lines. Each finger change of a node is indicated by a line going from the node that contains the finger to the finger itself. Different colors indicate the finger action was being monitored as described in the following list:

- New finger added - yellow
- Finger removed - orange
- Better finger added (that replaces a worse finger) - green
- Worse finger removed (that was replaced by a better finger) - red

Every finger change is added to the graphic. The lines can be removed by either clicking “Deletes lines” manually or on every keep-alive broadcast message by checking the box “Clear on ka”. The later allows for a good visualization in relation to the effect that the last keep-alive event had on the health of the DHT. The GUI further allows some basic control of the DHT. This involves adding nodes, removing nodes and changing the network and node delay. For a screen shot of the GUI, Figure 15 should be investigated.

When the amount of events becomes too high, inspection of events on the command line could become too complicated. For that purpose a log-file is created that contains exactly the same information. The log file can be used for further evaluation, for example, a text editor. The log file is automatically created every time the simulation environment is started and is saved to a file called “media_sense.log” in the user's directory. Every event produces a new line in the file, always starting with a timestamp. The following events are logged:

- Transmitted messages
- Node add, remove and kill events
- Finger change events
- Keep-alive events

See Appendix B for an example log file.

5.2 Chord statistics

The following graphs were constructed by gnuplot [12] using the data files that have been written from the statistic class. Each graph will be described by a following text in order to provide a better understanding. The graphs will show the described scenarios from Chapter 3.3.

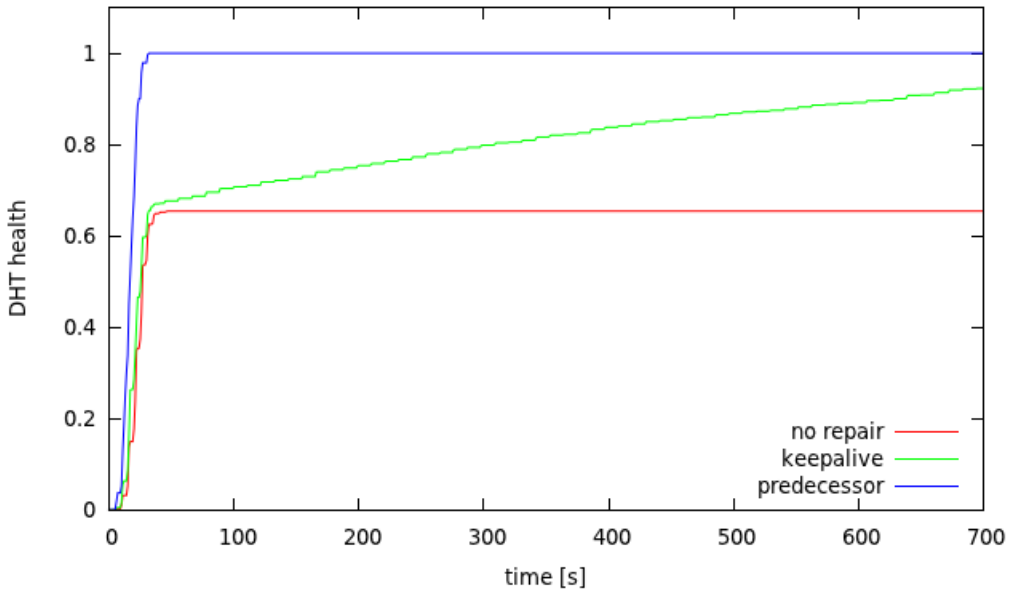

Figure 16: Health in different development statuses

Figure 16 shows the DHT health in relation to the time which has passed. The data lines have been recorded at different development phases, but all in the same scenario. This scenario was to insert 1000 nodes directly after starting the system. The red one (no repair) was at a very early level, when the DHT solution did not have any repair algorithms. It shows that the DHT health was never better than approximately 0.6 which means that 60% of all fingers in the DHT were correct. When keep-alive messages had been included (green line), the health did not remain at a certain level, but increased linearly over time towards 100%. At a certain point in the development process, the idea arose to additionally store the predecessor of every node in the finger table. With this extra information it is possible to have 100% health immediately after the insertion of all nodes.

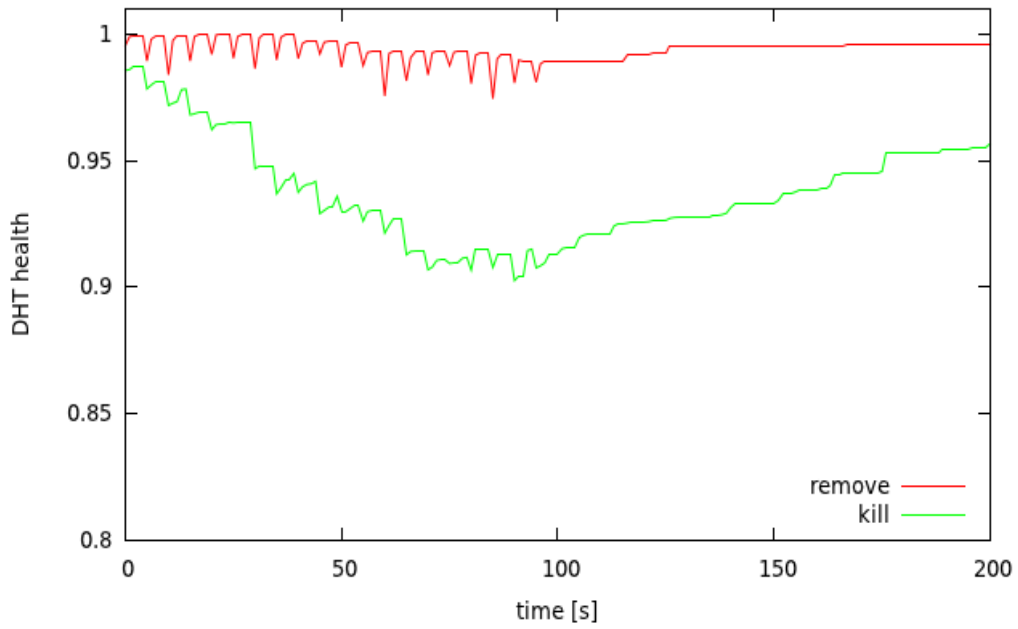

Figure 17: Comparison between leaving and failing nodes

Figure 17 shows two scenarios. In both cases 500 nodes were added to the DHT after the statistic was started. After the DHT was set up, 100 nodes were removed in groups of 5 and in periodic intervals of 5 seconds. The lines show the different behavior of the nodes leaving the DHT regularly (red line) and the nodes that fail (green line). Because the regularly leaving nodes send leave notifications with helpful information to the other nodes, the health is almost immediately again at a value of nearly 100% again. However, this is not the case if the nodes fail (for example imagine the nodes loose network connection), and in this case it can be seen that the health drops to approximately 90%, after which it slowly returns towards 100%. This is because failing nodes are unable to send recovery information to other nodes in the first moment. However, the required information is propagated, over time, with the assistance of the implemented repair mechanisms.

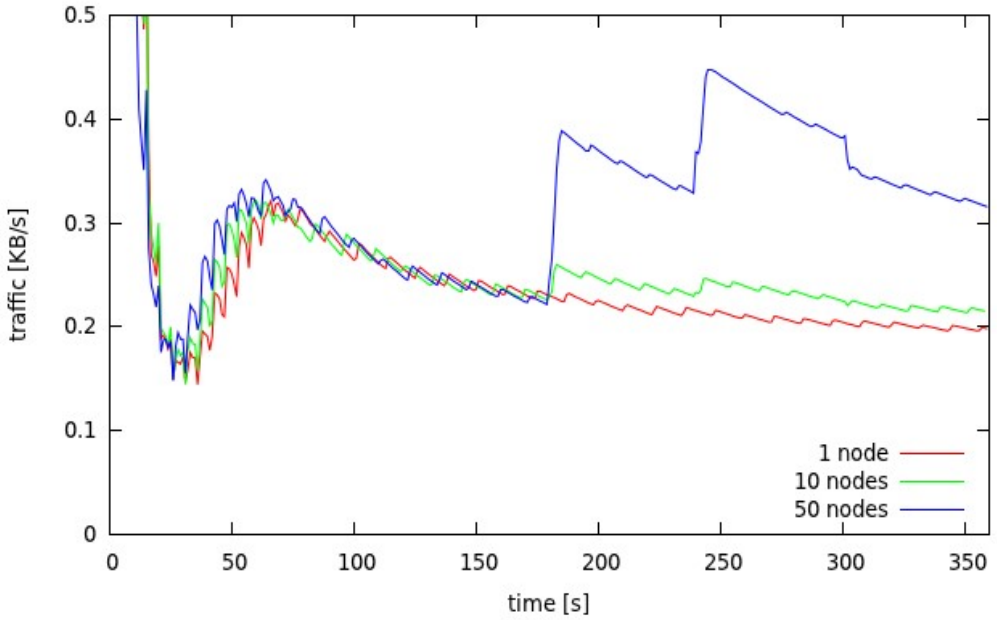

Figure 18: Average traffic per node per second

Figure 18 consists of three data lines, all of which show the traffic produced by the DHT in kilobytes per second for each node, on average. The test scenario was, that 500 nodes were inserted immediately after the program start. After that, a pause of 180 seconds has been made in order to determine the normalization of an idle DHT. When the pause was over, a certain amount of nodes were leaving the DHT (blue = 50 nodes, green = 10 nodes, red = 1 node). After another break of 60 seconds the same amount of nodes were failing (simulated) in the DHT. After the last break (60 seconds again) the same amount of nodes were added to the DHT. This measurement proves, that the average data amount produced for each node is at an average level of 200 Bytes per second. If there are very many nodes leaving the DHT at the same time, traffic peaks will be produced which is caused by the broadcast messages. Adding nodes to the DHT does not affect the traffic generation in a negative way.

5.3 RUDP simulation environment

The simulation environment does not have any user input methods. After startup it immediately start to transmit data at maximum speed between the two sockets using the specified hostname. The transmission speed is displayed in MB/s and is updated every 500 ms. The measurement can be started with two parameters specified, the

transmission side (sender / receiver) and the used protocol (RUDP / TCP).

The number of transmitted packets and bytes is hardcoded. If the connection is so bad that the RUDP implementation considers the link as having failed, then the application stops with an error message.

5.4 Comparison of RUDP and TCP performance

The following 2 graphs show the performance evaluation of RUDP in contrast to TCP which can be seen as a reference implementation because it can be considered as beeing sufficiently sophisticated to deal with every kind of network failures.

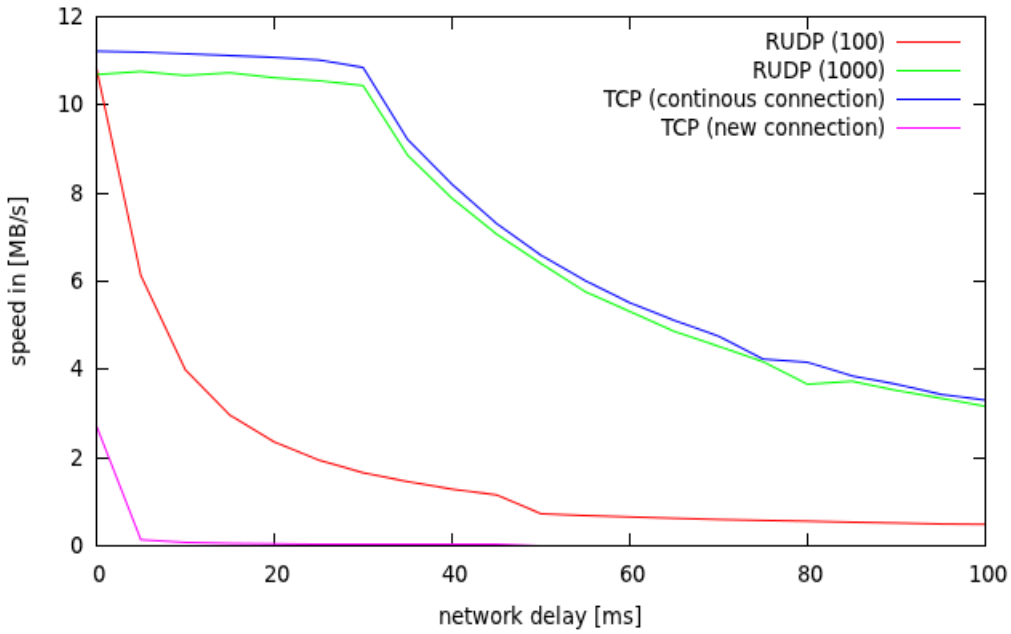

Figure 19: Comparison of RUDP and TCP in packet delay

Figure 19 shows the performance of TCP and RUDP with increasing network delay from 0 ms to 100 ms. The graph shows that the small RUDP window (100 packets) has a bad performance because the sender has to wait for the acknowledge answers from the receiver, which takes longer with increasing delay. After tuning the window size manually to a size of 1000 packets, the RUDP performs quite as well as the TCP because more packets can be sent in parallel, which drastically reduces the time waiting for acknowledgements. The continuous TCP connection achieves a high performance because the flow control and error

correction of TCP is highly capable of recovering from error situations and, in this graph, it can be seen as the upper limit of the transmition line. When TCP is used with reconnection the performance is very poor because 7 packets have to be used to transmit one data packet instead of immediate transmission in the RUDP. The breakdown at 30 ms is due to a full network output buffer in the kernel of the operating system. This buffer can only hold a certain amount of untransmitted data packets in its memory.

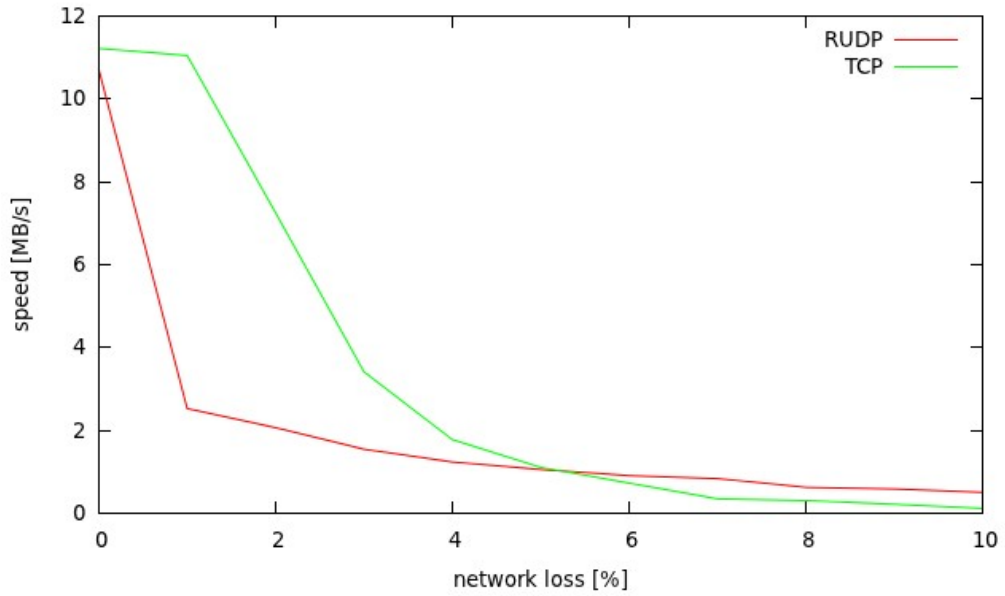

Figure 20: Comparison of RUDP and TCP in packet loss

Figure 20 compares both protocols under packet loss situations ranging from 0% to 100%. TCP performs significantly better than RUDP at low loss rates because its acknowledgement and packet resend technique are better. At a higher loss of above 5% RUDP is slightly better than TCP. A deeper analysis of both protocols could show the reason for this difference. The mesurement of reconnecting TCP has been missed out because of its immediate failure at 1% packet loss.

6 Conclusion

Each of the eight milestones that are listed in the detailed problem statement has been achieved successfully. Both function addons have been implemented successfully according to our specified methodology. The simulation environment for Chord was a rather powerful tool and it provided excellent assistance during development.

Basically, existing ideas and techniques relating to Chord and Reliable UDP, which already existed, were reassembled. However, it was also the case that new ideas and improvements were added. For example, the inclusion of the preceding node in the finger table, which greatly enhances the repair speed of the DHT. Further the possibility for RUDP to acknowledge packets implicitly and selectively.

There were no preconveived expectations in relation to the measured results but they are all reasonable and provided good feedback, which also lead to ideas for future development. One example being the evaluation of the RUDP procotol. The evaluation made clear that fixed parameters such as window size will not work in all situations. However, it provided surprising that the new scheme war better than TCP in the situation of high packet loss. A question that arises here is the amount of data overhead, which means how much data is actually required to successfully transfer one bit from the source to the destination. This was not measured during this project and it is expected that TCP is better in this discipline. Some improvements were implemented immediately because they exposed design issues.

6.1 Future Work

During the project some possible improvements were determined that did not fit into the requirements associated with the project. In addition, some implementations fell outside the scope of this project but it would be profitable to pursue them. These could form the subject for future work.

What both have in common is that they have only been tested in laboratory environments and they require more practical testing on "real" networks. The deployment of a beta product could fit that need.

6.1.1 Chord

At the moment the measured results for the Chord implementation are not relative because they have not been compared to other DHT solutions such as Pastry, Tapestry or CAN. This should be conducted in order to provide more reflections in relation this implementation.

The redundancy of data could be significantly improved. At the moment they are stored at 2 positions, the originating node and the node that is responsible for the hash value. Saving the data at more nodes would provide more safety and faster lookup possibilites.

The DHT solution has been tested by simulating different situations in the test environment. An automatic test scenario, which can produce tests over longer terms (for example a night) should proove to be good option.

Actually, the hash value (the node id) for the nodes is generated from the nodes network adress and it is possible that a more useful algorithm to generate the node id is available. For example to build groups of nodes.

6.1.2 Reliable UDP

All parameters of the RUDP implementation such as acknowledge time, window size, and more are static. The performance might be increased by finding appropriate values, or better, by implementing algorithms that vary the parameters depending on the available context such as the average round trip time (RTT) and others. RUDP requires a different link fail detection. At the moment. if a packet fails after a certain number of retries, the link is considered as having failed even if others packets have been transmitted successfully. A better approach would be to set the link as having failed only if NO packet at all could be transmitted for a certain amount of time.

To speed up the acknowledge process the sender could not only resend packets after the resend timer expires but also when an acknowledge packet has arrived that contains gaps in the ACK field because these packets are most probably lost.

Security vulnerabilities have not been taken into consideration during development and are required to be determined and countermeasures have to be implemented.

It would be of interest to have a C/C++ implementation so RUDP is not only bound to Java. An attempt was made to keep the packet structure simple so it is easy to parse RUDP packets on embedded devices.

References

[1] Margery Conner, "Sensors empower the Internet of Things", Electronics Design, Strategy, News article, 2010-05-27

[2] Theo Kanter, "MediaSense – an Internet of Things Platform for Scalable and Decentralized Context Sharing and Control"

[3] Buzzle.com, "Peer-to-peer vs. Client Server Networks", http://www.buzzle.com/articles/peer-to-peer-vs-client-server-networks.html, 2011-06-18
Retrieved on: 2012-03-17

[4] "A Survey and Comparison of Peer-to-peer Overlay Network Schemes",
IEEE Communications survey and tutorial, March 2004

[5] James F. Kurose and Keith W. Ross, Computer Networking – Atop down approach, fifth edition

[6] "Alternatives to the Chord protocol", unknown author, http://nislab.bu.edu/sc546/sc441Spring2003/CallaMiraniCHORD/alternatives.html
Retrieved on: 2012-03-26

[7] "Chord: A scalable Peer-to-peer Lookup Service for Internet Applications" MIT Laboratory for Computer Science, August 2001

[8] Efficient Broadcast in Structured P2P Networks, www.sics.se/~seif/Publications/paper3.pdf
Retrieved on: 2012-04-04

[9] "Reliable UDP Protocol", T. Bova and T. Krivoruchka, IETF Network Working Group, 1999-02-25

[10] Eclipse,
http://www.eclipse.org/

[11] Git repository with source code,
https://github.com/miun/mediasense

[12] Gnuplot,
http://www.gnuplot.info

Appendix A: Console commands

The command is separated with a space from arguments. Arguments are separated by a comma from each other. The command usage syntax is as follows:

Bold - type exactly like that

Italic - replace with an appropriate argument

Underlined - argument can be repeated

[Brackets] - argument is optional

node_add_n *count*
Add count nodes to the DHT. The network addresses are the next free addresses in continuous order.

node_add *address*
Add multiple nodes with the specified network address(es). Address could be any string.

node_remove *address*
Remove multiple nodes specified by their network *address*(es). The node(s) is / are shutdown normally.

node_remove_n *count*
Remove count nodes. The node(s) is / are selected randomly. The node(s) is / are shutdown normally.

node_kill *address*
Kill multiple nodes specified by their network *address*(es). The node(s) is / are killed without any notice. This is the simulation of a network failure.

node_kill_n *count*
Kill *count* nodes. The node(s) is / are selected randomly. The node(s) is / are killed without any warning. This is the simulation of a network failure.

```
msg_watch ([! | all | broadcast | !broadcast] | [[!]join |
[!]join_response | [!]join_ack | [!]join_fainalize |
[!]join_busy | [!]duplicate | [!]register |
[!]register_response | [!]resolve | [!]resolve_response |
[!]keepalive | [!]find_predecessor |
[!]find_predecessor_response | [!]check_successor |
[!]check_successor_response | [!]check_predecessor |
[!]check_predecessor_response | [!]join_notify |
[!]leave_notify])
```

Activate and / or deactivate monitoring of the specified message categories or specified messages. The categories are **!** for none, **all** for all, **broadcast** for activating keepalive, notify_join, notify_leave and node_suspicious, and !**broadcast** for deactivating them. [!]**broadcast** does not interfere with other message types. Either a category or a list of specified messages can be entered.

register *address, sensor*
Register the sensor *sensor* at the node node specified by *address*.

resolve *address, sensor*
Try to resolve the *sensor* starting at the node specified by *address*.

g *radius*
Start the GUI. Optionally specify the *radius* of the DHT circle in pixel.

node_info [*address*]
Print information about all nodes or the node specified by *address*. The information contains state of connectivity, state of blocking and if blocked, for which node the block is held, and the hash value of the node(s).

sensor *address*
Show sensor information for the node specified by the network address address. The information contains a list of own sensors and where these are registered, and a list of foreign sensors this node is responsible for and the network address of the originating node.

node_watch *address*
Opens a small window for every node specified by its network address

address. The list contains the finger finger-table of that node and is automatically updated on changes.

msg_delay [[*delay*], *address*]
Shows the current global network delay when no parameter is specified. Sets the global network delay when *delay* is specified, and sets the node specific network *delay* if an *address* is specified.

circle *address*
Iterate through the DHT circle starting by the node specified by the network address *address*. Every traversed node is shown in order including network address and hash value. The traversal stops if the starting specified by *address* node is reached again, a side-loop or a hole has been detected. After the traversal orphaned nodes, if any, are listed.

finger *address*
List the finger table of the node specified by the network address *address*. The information includes the logarithmic position to the base of 2 of the finger, the hash value of the finger and the network address. The predecessor, that is not a part of the finger table, is also included in the listing. Successor and predecessor are marked with **SUC** and **PRE** respectively.

ka_watch
Switches the monitoring of keepalive events **on** or **off** depending on the former state.

health [**m**]
This command shows the current health of the DHT. The health is the percentage of correct fingers of all nodes. Fingers that are present but not perfect are treated as not present. The **m** option includes a list of missing fingers that were needed to reach 100% health.

wait *delay, [random_delay]*
Waits the specified *delay* in milliseconds. If *random_ delay* is also specified this command waits between delay and (delay + random_delay) milliseconds selected by a random number generator. This command is only useful inside script files. See the **exec** command for further details

exec *file*
Execute the script file specified by *file*. A script must contain valid command line commands or comments. A comment line starts with a # and is valid for the current line only. There is one command that is only valid inside script files, which is the **goto** command. See description of the goto command in the section for more details.

statistic *file*
Start a statistic and write it to the file *file.* Only one statistic can be run at a time. If a statistic was already running before, it is closed before the new statistic is started. See chapter 4.3.2 inside the project report for more details on statistics.

break *address*
This command is for debugging purposes only and works only when the simulation environment is started within a Java debugger. It stops the execution of the node specified the network address *address.* Outside the debugging mode this command does nothing.

goto *mark*
This command is only valid within script files. It jumps the current execution pointer to the line that is marked with *mark.* A mark can be specified by a **colon** followed by a *mark*-name. A mark cannot be combined with other commands on the same line.

This example waits 1000 ms, adds 2 nodes and then jumps to the beginning.

```
:make_nodes
wait 1000
node_add_n 2
goto make_nodes
```

Appendix B: Example log file

```
Logging started Sat May 05 18:02:56 CEST 2012

----------
18:03:04.783 | MSG: type: MSG-JOIN - from: (1) - to: (0) key: {356a...} - origAdr: (1)
18:03:05.036 | MSG: type: MSG-JOIN-RESPONSE - from: (0) - to: (1) | joinKey: {356a...} suc:
{b658...} pre : {b658...}
18:03:05.288 | MSG: type: MSG-JOIN_ACK - from: (1) - to: (0) key: {356a...}
ADD-NEW finger: (1)-{356a...} @NODE: (0)-{b658...}
ADD-NEW finger: (1)-{356a...} @NODE: (0)-{b658...}
ADD-NEW finger: (0)-{b658...} @NODE: (1)-{356a...}
18:03:05.541 | MSG: type: MSG-JOIN_FINALIZE - from: (0) - to: (1) key: {356a...}
ADD-NEW finger: (0)-{b658...} @NODE: (1)-{356a...}
18:03:08.640 | MSG: type: MSG-JOIN - from: (2) - to: (1) key: {da4b...} - origAdr: (2)
18:03:08.892 | MSG: type: MSG-JOIN - from: (1) - to: (0) key: {da4b...} - origAdr: (2)
18:03:09.144 | MSG: type: MSG-JOIN-RESPONSE - from: (0) - to: (2) | joinKey: {da4b...} suc:
{356a...} pre : {b658...}
18:03:09.396 | MSG: type: MSG-JOIN_ACK - from: (2) - to: (0) key: {da4b...}
ADD-NEW finger: (2)-{da4b...} @NODE: (0)-{b658...}
ADD-BETTER finger: (2)-{da4b...} @NODE: (1)-{356a...}
18:03:09.649 | BROADCAST:{356a...} -> {b658...} | MSG: type: MSG-JOIN-NOTIFY - from: (0) -
to: (1) hash: {da4b...} - Adr: (2)
ADD-NEW finger: (1)-{356a...} @NODE: (2)-{da4b...}
18:03:09.649 | MSG: type: MSG-JOIN_FINALIZE - from: (0) - to: (2) key: {da4b...}
ADD-NEW finger: (0)-{b658...} @NODE: (2)-{da4b...}
18:03:09.901 | MSG: type: KEEP-ALIVE - from: (1) - to: (2)
18:03:13.106 | MSG: type: MSG-CHECK_SUCCESSOR - from: (0) - to: (1) | hash: {b658...}
18:03:13.107 | MSG: type: MSG-CHECK_PREDECESSOR - from: (0) - to: (1) | hash: {b658...}
18:03:13.359 | MSG: type: MSG-CHECK_SUCCESSOR_RESPONSE - from: (1) - to: (0) | preHash:
{b658...} preAddr: (0)
18:03:13.359 | MSG: type: MSG-CHECK_PREDECESSOR_RESPONSE - from: (1) - to: (0) | preHash:
{da4b...} preAddr: (2)
18:03:15.793 | MSG: type: MSG-CHECK_SUCCESSOR - from: (1) - to: (2) | hash: {356a...}
18:03:15.794 | MSG: type: MSG-CHECK_PREDECESSOR - from: (1) - to: (2) | hash: {356a...}
18:03:16.046 | MSG: type: MSG-CHECK_SUCCESSOR_RESPONSE - from: (2) - to: (1) | preHash:
{356a...} preAddr: (1)
18:03:16.046 | MSG: type: MSG-CHECK_PREDECESSOR_RESPONSE - from: (2) - to: (1) | preHash:
{b658...} preAddr: (0)
18:03:18.298 | Node: {b658...} Addr: 0 initiated KEEP-ALIVE
ADD-NEW finger: (0)-{b658...} @NODE: (2)-{da4b...}
18:03:18.551 | BROADCAST:{da4b...} -> {356a...} | MSG: type: KEEP-ALIVE - from: (0) - to:
(2)
18:03:18.551 | BROADCAST:{356a...} -> {b658...} | MSG: type: KEEP-ALIVE - from: (0) - to:
(1)
18:03:19.903 | MSG: type: MSG-CHECK_SUCCESSOR - from: (2) - to: (0) | hash: {da4b...}
18:03:19.903 | MSG: type: MSG-CHECK_PREDECESSOR - from: (2) - to: (0) | hash: {da4b...}
18:03:20.155 | MSG: type: MSG-CHECK_SUCCESSOR_RESPONSE - from: (0) - to: (2) | preHash:
{da4b...} preAddr: (2)
18:03:20.155 | MSG: type: MSG-CHECK_PREDECESSOR_RESPONSE - from: (0) - to: (2) | preHash:
{356a...} preAddr: (1)
18:03:23.107 | MSG: type: MSG-CHECK_SUCCESSOR - from: (0) - to: (1) | hash: {b658...}
18:03:23.107 | MSG: type: MSG-CHECK_PREDECESSOR - from: (0) - to: (1) | hash: {b658...}
18:03:23.359 | MSG: type: MSG-CHECK_SUCCESSOR_RESPONSE - from: (1) - to: (0) | preHash:
{b658...} preAddr: (0)
18:03:23.359 | MSG: type: MSG-CHECK_PREDECESSOR_RESPONSE - from: (1) - to: (0) | preHash:
{da4b...} preAddr: (2)
18:03:25.794 | MSG: type: MSG-CHECK_SUCCESSOR - from: (1) - to: (2) | hash: {356a...}
18:03:25.794 | MSG: type: MSG-CHECK_PREDECESSOR - from: (1) - to: (2) | hash: {356a...}
18:03:26.046 | MSG: type: MSG-CHECK_SUCCESSOR_RESPONSE - from: (2) - to: (1) | preHash:
{356a...} preAddr: (1)
```

Appendix C: Example Statistic file

```
# TriggerType: TRIGGER_SECOND
#-----
#TimeStamp       Sec     Hth      Con      ConD     Fin      FinD     Data     DataD    Pkt
         PktD
15:00:19.505     0       0.0      1        1        2        2        22731    22731    1079
         1079
15:00:20.505     1       1.9708316909735908E-4     1        0        4        2        22760
         29      1080    1
15:00:21.505     2       1.9708316909735908E-4     1        0        4        0        22760
         0       1080    0
15:00:22.505     3       1.9708316909735908E-4     1        0        4        0        22760
         0       1080    0
15:00:23.505     4       1.9708316909735908E-4     1        0        4        0        22760
         0       1080    0
15:00:24.505     5       1.9708316909735908E-4     2        1        6        2        52071
         29311   2171    1091
15:00:25.505     6       1.9708316909735908E-4     3        1        25       19       54762
         2691    2434    263
15:00:26.505     7       1.9708316909735908E-4     3        0        25       0        54762
         0       2434    0
15:00:27.505     8       1.9708316909735908E-4     3        0        25       0        54762
         0       2434    0
15:00:28.505     9       1.9708316909735908E-4     3        0        25       0        54762
         0       2434    0
15:00:29.505     10      1.9708316909735908E-4     4        1        28       3        88737
         33975   3533    1099
15:00:30.505     11      1.9708316909735908E-4     7        3        90       62       94885
         6148    4005    472
15:00:31.506     12      1.9708316909735908E-4     7        0        90       0        95050
         165     4010    5
15:00:32.506     13      1.9708316909735908E-4     7        0        90       0        95050
         0       4010    0
15:00:33.505     14      1.9708316909735908E-4     7        0        90       0        95050
         0       4010    0
15:00:34.505     15      1.9708316909735908E-4     8        1        93       3        130200
         35150   5100    1090
15:00:35.505     16      7.883326763894363E-4      15       7        209      116      145538
         15338   5826    726
15:00:36.505     17      9.854158454867955E-4      15       0        236      27       148435
         2897    5875    49
15:00:37.505     18      9.854158454867955E-4      15       0        236      0        148435
         0       5875    0
15:00:38.506     19      9.854158454867955E-4      15       0        236      0        148435
         0       5875    0
15:00:39.505     20      9.854158454867955E-4      16       1        238      2        183251
         34816   6943    1068
15:00:40.505     21      0.002364998029168309      29       13       418      180      212895
         29644   7987    1044
15:00:41.505     22      0.00256208119826566 8     30       1        565      147      232091
         19196   8299    312
15:00:42.505     23      0.0027591643673630273     30       0        588      23       233676
         1585    8324    25
15:00:43.505     24      0.0027591643673630273     30       0        588      0        233676
         0       8324    0
15:00:44.505     25      0.0029562475364603865     31       1        591      3        267367
         33691   9355    1031
15:00:45.505     26      0.005912495072920773      56       25       799      208      309561
         42194   10677   1322
15:00:46.505     27      0.012810405991328341      59       3        1329     530      401714
         92153   12074   1397
```