**Mid Sweden University**
The Department of Information Technology and Media (ITM)

Authors: Timo Schröder, Florian Rüter
E-mail addresses: t.schroeder84@web.de, florian.rueter@googlemail.com
Study programme: Computer Science , 180 Credits
Examiner: Ulf Jennehag, Ulf.Jennehag@miun.se
Supervisor: Stefan Forsström, Stefan.Forrstrom@miun.se
Scope: 11041 Words inclusive appendices
Date: 2012-06-07

Bachelor Thesis

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

**Timo Schröder**
**Florian Rüter**

# Abstract

…

**Keywords:** …

# Table of Contents

# Terminology

## Acronyms

| | |
|---|---|
| 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 |
| 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 the Mid Sweden University in Sundsvall.

## 1.1 Background

Applications that can change their behaviour based on the context of the users are called context-aware applications. By the introduction of smart mobile phones that carry a multitude of sensors and actors, these applications have had a very large market penetration. The Mid Sweden University has so far produced 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 comes along 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) a lookup-service is needed. To handle the communication between two contexts a transport protocol is necessary. The present lookup-service, that has been implemented by the Mid Sweden University so far, is based on a client-server architecture. The well-known network protocols Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) can be used for data transmission. Since its beginning it is a requirement to the MediaSense platform to not have a single point of failure. The client-server architecture does not fulfil that need. 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 fulfil both requirements at the same time. So the need for a self-made communication protocol arises.

## 1.3 Detailed problem statement

To have verifiable goals 8 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 allows the parallel running of multiple DHT clients

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

3. Evaluate the performance of the DHT in terms of 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, able to host two clients communicating 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 and reliability

8. Transfer the RUDP solution into MediaSense platform

## 1.4 Scope

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

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

A further requirement to this thesis work is that the code is free of external licensing. For that reason both parts will be created from scratch.

## 1.5 Ethics etc

State the problems issues

## 1.6 Outline

Briefly describe the report's outline. “Chapter 2 describes...”

## 1.7 Contributions

Who did what

# 2 Theory

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

## 2.1 Internet-of-Things

The “Internet of Things” aims specially for two main aspects. The first one is, to interconnect all possible sensor and actors over the Internet. These sensors might be of all imaginable kindnesses, for example a GPS sensor integrated in a mobile phone, a temperature sensor in houses or a camera looking at the traffic at certain points on the streets. Also the actors can be from all imaginable kindnesses, to follow the earlier examples, an application that can tell you were you friends are, and automatic heating regulation or electronic traffic signs that change their speed limit related to the traffic situations. That all produces a larger input than humans are doing by their keyboards or mouses. And exactly this is, why the IoT not only aims the interconnection of everything but also to get its components intelligent. What means that the sensors and actors should show intelligent behaviors on the output from other sensors/actors without any human help (machine to machine communication) [1].

The Mid Sweden University 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 to provide an adequate Quality of Service (QoS) and Quality of Experience (QoE). They also built a first distribution of their framework, which allows to implement new features in the future without a complete redistribution. The interest 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 allows, to share data over a network without the need of a central server [3].

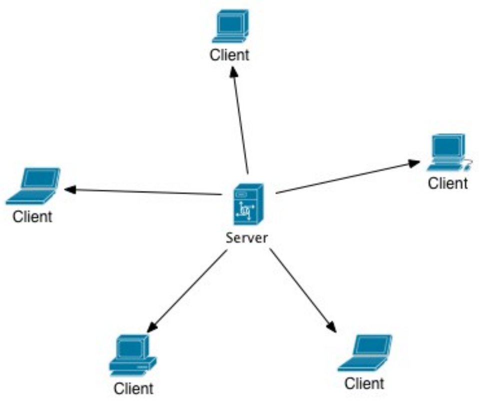


**Figure 1: Client-Server model**

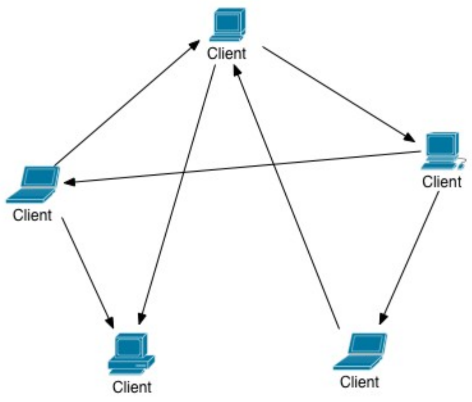


**Figure 2: Peer to peer**

P2P is a distributed application architecture that partitions duties and responsibilities among all 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. That means, in a P2P architecture each client has the same rights and responsibilities. In contrary to a P2P network, where each node acts as supplier and consumer, stands the client-server-model, where 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 from the network topology [4].

P2P networks can further be divided in structured and unstructured systems. Structured systems organize their peers and resources with specific algorithms typically using distributed-hash-tables (DHT) what is explained in chapter 2.3.2. Whereas unstructured systems do not use any structure in their overlay networks [4].

### 2.2.2 Advantages and weaknesses

In comparison to client-server networks, P2P networks come along with the big advantage, that there is no single point of failure (i.e. the server in client-server networks). In fact P2P networks are getting more stable, the more clients are participating. Another advantage of P2P networks is, that all available resources are getting more when more clients join the network, while in client-server networks the available resources per client are getting less, because all clients have to share the resources from the server. On the other side, P2P networks also bring some disadvantages. Since each client in the system is responsible to share (publish) some of its resources, the whole system is more vulnerable for untrusted or unsigned content. This would not happen 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 a given input data. The function should also produce values that are uniformly spread over the possible range of hash values 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 to deal with a collision situation, where the easiest way is to deny the colliding key. [5]

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

### 2.3.2 Distributed Hash Table

A distributed hash table (DHT) is the same as a hash table as described in XX except that the table is distributed among several parties, where each party is responsible for a certain amount of the hash space. In the sense of P2P, 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 the 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 to split up the hash space in smaller parts, as CAN, Tapestry, Pastry, Chord and others [6]. This project deals with Chord, which is is described in more detail in chapter 2.3.3 .

### 2.3.3 Chord

The Chord lookup protocol arranges its participating nodes in a 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 key-value pairs have to be shifted to keep the DHT valid [7].

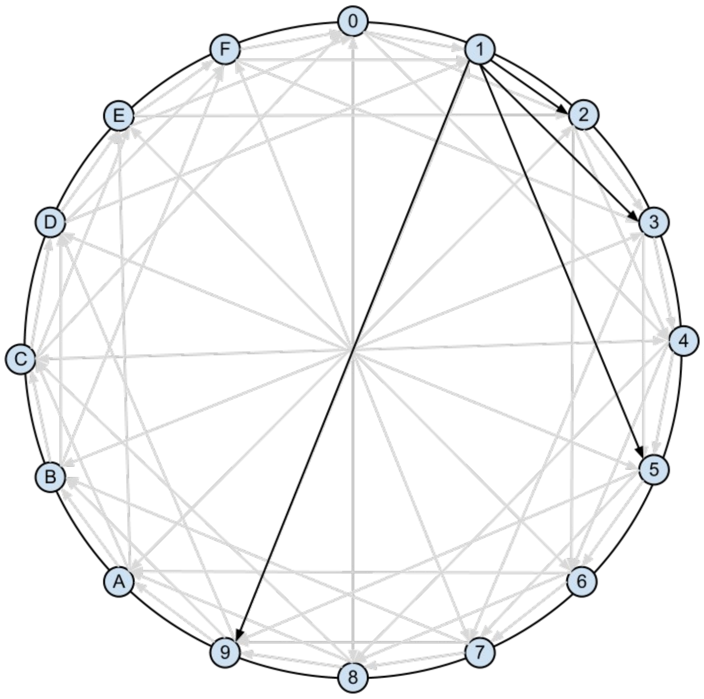


**Figure 3: Chord example with 16nodes showing the finger table of node "1"**

To be able to forward requests, and to know the responsible range of hashes each node must at least know its successor node. In this case a message between two random nodes will 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. This means that effort to find a position increases linearly with the size of the DHT. The effort can be expressed as $O(n)$. To reduce the effort a finger table is used. Figure 3 gives you a graphical overview over such a finger table [7].

Each node in a chord DHT contains a list of known other nodes of the DHT. This list is also referred to as 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 with the following formula 1:

$$(n+2^{(i+1)} \bmod m)$$

*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.

This 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 to find a position is at maximum $O(log2(m))$ because after every step the distance is decreased to the half [7].

The successor, that is also the very first finger in the finger table, is the direct succeeding node in the DHT. If the finger table contains wrong links 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, what could cause the queries to fail. Because of that check and repair mechanisms are used to keep the correct successor, which we will be explained at a later point in this chapter.

A joining node has to do 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 and
- 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, 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 regularly has to do 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 do checking and repairing to keep the DHT functional and stable. 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 keep 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 shall be forwarded to. The lower limit is the node itself [8].

Assumed that Node 0 in figure 3 wants to initiate a broadcast, the range is the whole circle from his key to his key minus one. The first step is, that Node 0 then 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

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

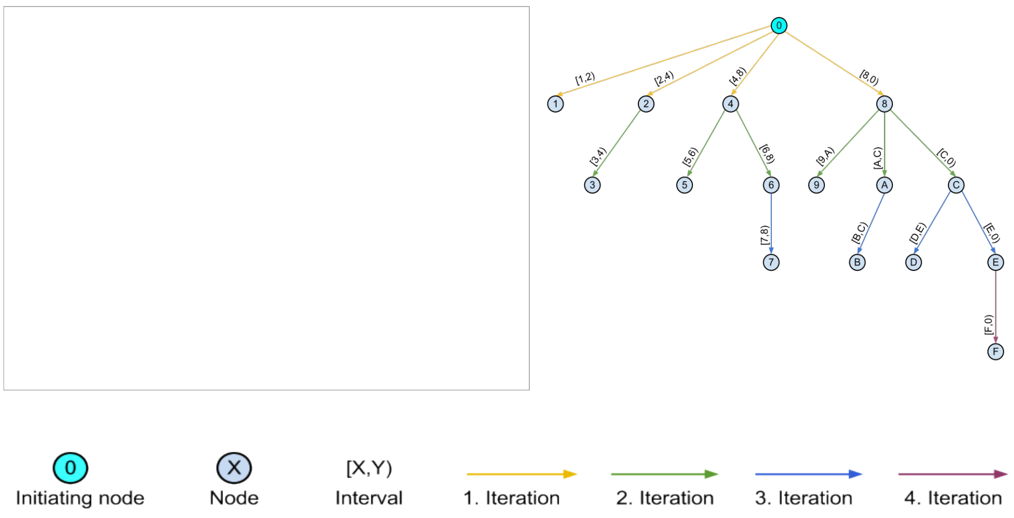


**Figure 4: Broadcast algorithm**

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 just send once to avoid duplicates. If a finger is missing the broadcast is propagated correctly anyhow (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 put in 2 categories which are **packet oriented** and **connection oriented** [5].

The **packet oriented** protocols sends packets from a source to a destination. The path the packets take is not predetermined and can change from packet to packet. If more packets are sent the order can change on the way to the destination. Packets can also get lost without any notice. It is up to the application programmer to include meta information into the packet data if correct order and reliable transmission is 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 tries to setup connection parameters and synchronization. If the receiver agrees a virtual link is established between both.

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

3. Connection shutdown - One side decides to shut down the connection link, informs the other side about that decision, and both start a shutdown process, like emptying buffers and stopping 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 who are important for this project are explained in the following sections.

### 2.4.1 Cyclic Redundancy Check

A CRC algorithm calculates a value of fixed length from a variable length of data (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 have been altered during transmission and the data can be considered as broken [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 to detect an error is $(2^n-1)$ and the probability to not detect an error is $1/(2^n)$ [5].

### 2.4.2 Window flow control

The task of flow control is to not overload the receiving side of a communication link with data. This means that the sender stops to send information after a certain amount of sent data or packets and waits for information from the sender when to continue [5].

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

A typical implementation is a >>moving window<< that has a starting point and a length. The sender starts to fill the window from on side with increasing sequence numbers but only until 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. This way the opposite site of the window gets free for additional data to be received. The sender has to be informed about the window shift so 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.

Further it was defined that RUDP should be defined in a way that it is easy possible to allow different characteristics for each connection so that the header length could be kept 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 the development of a Reliable User Datagram Protocol (RUDP). The approach to solve each milestone as defined in chapter 1.3 will be described in this chapter.

All the development is done 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 the identical API as the MediaSense platform to ease the later transfer of the implemented DHT. Furthermore the simulation environment should enable the following operations:

Host an theoretical 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 listing of detailed client information like finger-table, successor, predecessor, internal state etc.
- Visualize the DHT in a Graphical User Interface (GUI)

## 3.2 Implementation of 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 his own address and its successor, further 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 clients address is between the current clients and successors 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 health 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 to ease the later transfer of the implemented RUDP. Furthermore the simulation environment should enable the following operations:

- Host two MediaSense communication instances

- The first instance should be used to send data and the second to receive data

- 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 disestablishment 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

With the developed simulation environment the following comparisons between TCP and RUDP will be done:

- transmission speed in error free environment
- transmission speed in erroneous environments
- transmission overhead
- required additional traffic in error situations

## 3.8 Transfer RUDP to MediaSense

After 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. A sub-chapter to each defined goal describes detailed the important points.

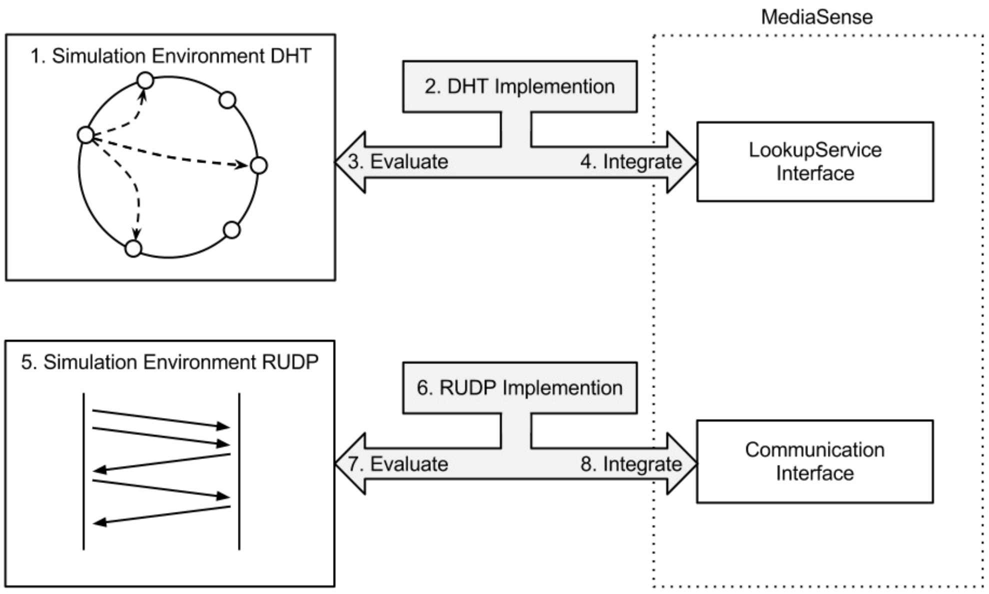


**Figure 5: Workflow**

## 4.1 Develop a simulation environment for DHT clients

Since one of the defined goals is to integrate the developed DHT solution to the MediaSense framework, the simulation environment is build upon the same interfaces as the framework uses. Figure 6 shows a rough class diagram about how the simulation environment is structured.

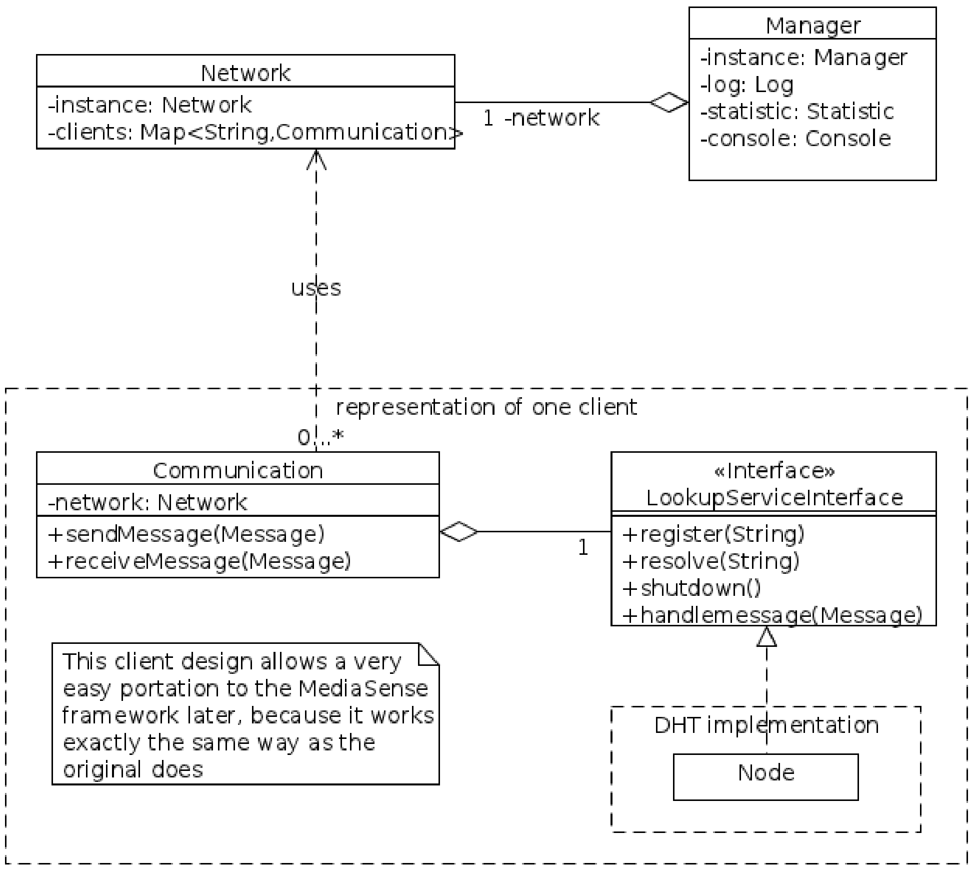


**Figure 6: Class Diagram**

### 4.1.1 Manager

The Manager class is the main entry point to the simulation environment. Here everything is tied up, what 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 per-requirements that follow from the MediaSense framework, each client consists of an Communication object and an LookupService object running separately in independent threads. The Communication object is responsible to send (send means, receive from the LookupService and forward to the network layer) and receive (receive means, receive from the network layer and forward to the LookupService) messages, and can be seen as the glue between the network and the DHT implementation.

### 4.1.3 Network

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

### 4.1.4 Console

The Console class is the main interface to the user, to control the simulation environment. The most important functionality is that clients can be added or removed from the network and that sensor 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 with the command “g” to start a graphical user interface. The GUI is built with basic Java Swing components, and kept as simple as possible. Further 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 do in include the following points:

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

- Finger changes

## 4.2 Implementation of a circular DHT based on Chord

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

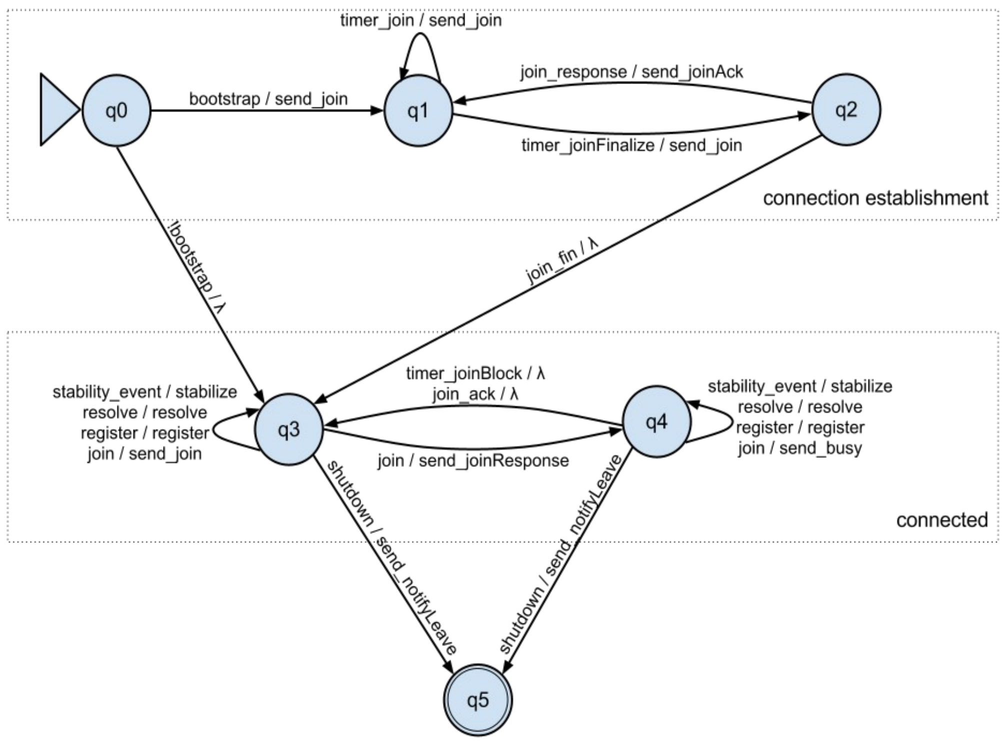


**Figure 7: Finite State Machine for a node**

### 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). Is that the case, the node immediately changes to the connected status and the connection establishment box is left. The second possibility is, that the node gets a regular bootstrap address. In that case, the node will try to establish a connection to a already running DHT instance. If the connecting node does not get 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 can not insert other nodes until the insertion process has finished.

### 4.2.2 Leave process

There are two possible ways in a DHT how the nodes might finish their availability. The regular way should be that the node sends a NOTIFY_LEAVE message to the DHT to inform all other nodes about its disappearance, but there are of course also problems like failing links when the node is not able to inform the DHT about its disappearance. In that case, the first node that recognizes the 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 of propagating messages. Each category contains several message types, where each type can contain additional data that is associated with that type

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

- REGISTER_RESPONSE - Response from a node that register a key-value pair that it is responsible for to the initiating node
- RESOLVE_RESPONSE – Response from a node that knows the location of a registered sensor to the queried key
- 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 predecessors former successor is its new 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 tries 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; See chapter XX for detailed repairing mechanisms

- 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; See chapter XX for detailed repairing mechanisms

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

Another possible message type are query messages. Query messages can be send to any node in the DHT regardless 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 one responsible for the destination. For this reason every message must at least contain 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 to 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 are broadcast messages. A broadcast message is a container that can encapsulate any other message. As described in chapter XX a broadcast message needs a limit which is the maximum hash value to which a certain node should forward a message. The initiating node sets these limits according the position in the finger table. For this to work a broadcast message must at least contain a start-key and an end-key to to specify this range. For a normal broadcast the start-key is always the initiating / forwarding node. But 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 send to check the network connection and to advertise nodes for finger table upgrading. See chapter XX for a detailed description if keep-alive messages
- NOTIFY_JOIN - Send by a node after the join process to inform all other nodes of its arrival
- NOTIFY_LEAVE - Send by a node before the shutdown process to inform all other nodes of its planned leaving
- NODE_SUSPICIOUS - Send by a node that has encountered a network problem with another node to inform all other nodes in the DHT of that

## 4.3 Evaluation of the DHT solution

This sub-chapter explains the functionality that has been built to evaluate the DHT solution. Additional to those parts that are defined in a sub chapter, testing  the DHT implementation in the simulation environment also is a part of the evaluation.

### 4.3.1 Health

The health of the DHT can be measured using the health function. These function returns 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. Out of that calculation the health is built.

### 4.3.2 Statistic

With the command “statistic 'file-path'”, it is possible to write a statistic to a specified file. The statistic is triggered once per second and writes the following points to a matrix in the specified file:

- 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 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 enough despite the removal of the debug functions. These were designed in a way where they could be removed easily without influencing other program parts. The existing MediaSense framework has already a layer for Lookup-services since it has already a client server solution. Regarding this a new package has been constructed where the source files has been inserted.

## 4.5 Develop a simulation environment for communication protocols

Since an other goal defined in chapter 1.3 is to integrate the developed RUDP solution to the MediaSense framework, the simulation environment is build upon the same interfaces as the framework uses. Figure XX shows a rough class diagram about how the simulation environment is structured.

TODO insert picture here

## 4.6 Develop a reliable packet-based communication protocol

Figure XX shows a FSM describing the behavior of the RUDP implementation.

TODO insert FSM

### 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 fields and the 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 | Field of short values. 0 < n < 32 |
| *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 receiver state
  - ACK - (bit 2) - packet contains acknowledge information
  - Data (bit 3) - packet contains application layer data
  - Resend (bit 4) - packet was resend
  - 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 bit long and is placed in the UDP header. In case of a manipulated packet the underlying layer will automatically drop this packet, so no additional checksum field has been implemented.

### 4.6.3 Link synchronization

Each packet that is 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 direction, sending and receiving, have an 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 the sender can choose. 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 and reordering occurs. The range of the sequence number 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. Virtually that 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 size. The first data packet on a link is marked with the FIRST flag. A new receiver must always get 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 to be 1 and is thus only allowed to send one packet at the beginning, that is also 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 the own window size.

Unsynchronized situations can appear where the knowledge about each others state is not consistent. If the receiving side of the link gets a FIRST flagged packet when it does not assume one or it gets 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 like a newly created link.

Every sent packet has to be acknowledged implicitly or selectively, as described in the next 2 sections. Packets can be acknowledged in groups. An acknowledgement is sent in the following situations:

- If the receiver buffer is less full than 2/3rd: 100 ms after a packet has been received. The 100 ms wait time is used to collect more packets and reduce the number of ACK packets
- If the receiver buffer is filled equal or more than 2/3rd: 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

If a receiving side of a link 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 successfully received.

Received packets can explicitly acknowledged by the receiver. This is done when the receiver window contains gaps with unreceived packets or the packets have not been consumed yet and still block the receiver window. 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 32). All acknowledge field data is relative to the acknowledge window start value. The format of the acknowledge field is as follows:

The acknowledge field contains acknowledge field count + 1 values of type short. Number n to number n+1 specifies a range of n to (n+1) - 1. Every even range with an even number specifies acknowledged packets and every odd range specifies unacknowledged packets. The following example should make the mechanism more clear:

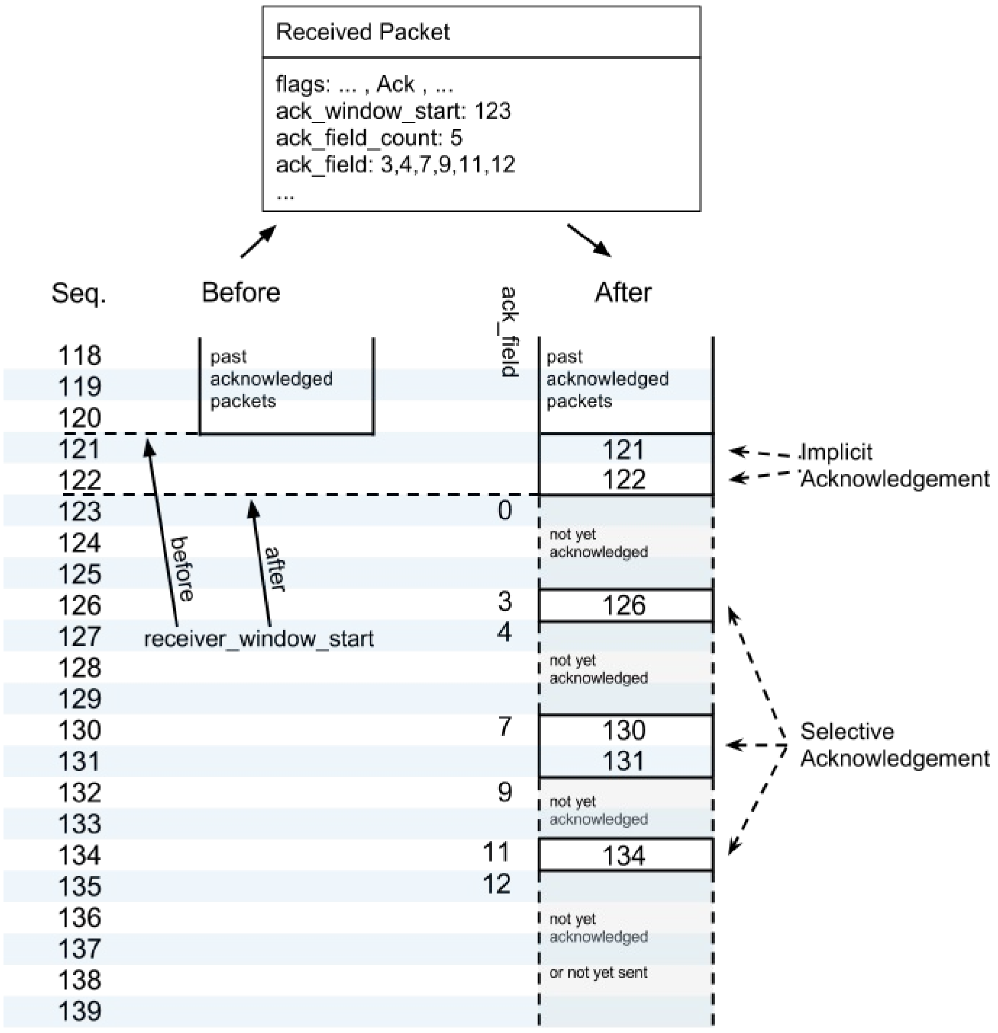


**Figure 8: Example of receiving an acknowledgement packet**

If a packet is not acknowledged implicitly or selectively after a fixed amount of time the packet is resent and the wait timer for this packet is started again with the wait time doubled. After 5 unacknowledged tries the link is assumed to be failed and put into shutdown state.

A link is considered as failed if a packet is not acknowledged after 5 send tries. The whole link, including sender and receiver, is then put into link-failed state. Every send and receive attempt on that link will then lead to an exception of 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 to receive more packets. If this ACK packet gets 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 informing 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 free within the persist packet transmission attempt, the link is considered failed and put in link-fail state.

### 4.6.4 Flow control

Every link consists of a sender and a receiver part. The receiver has a receiving buffer, or receiver window, whose size the client can decide as 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 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 9 for an example of receiver window action.

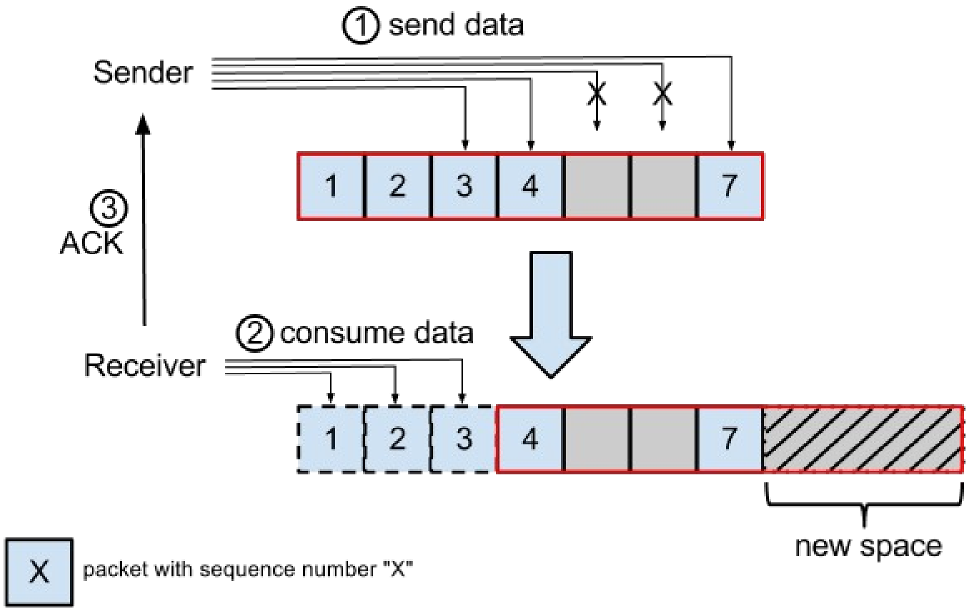


**Figure 9: Visualization the window shifting**

The client has to make the consideration about 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 in terms of speed and reliability

We used the simulation environment for the evaluation. Two RUDP sockets are connected to each other locally to exclude network interference from the evaluation. The reliability has been checked by transmitting random data between the 2 sockets and comparing the correctness and order of the data after receiving.

The evaluate the maximum possible speed random data is exchanged between the 2 sockets and the speed is measured. To test bad conditions in a reproducible way the Linux tool >>tc<< (traffic control) is used to simulate packet loss, packet reordering, and packet delay.

## 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.

TODO more text

## 4.9 Additional changes

Some improvements have been made apart from the mentioned milestones. The improvements have been implemented during the DHT implementation phase.

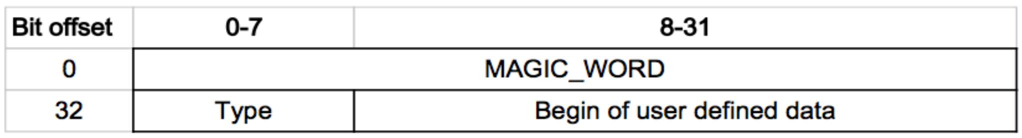
TODO more text

### 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. Also 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.

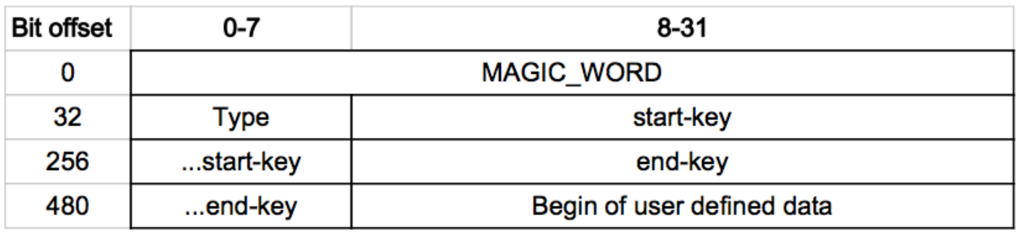
### 4.9.2 Binary message serializer

Additionally to the “EnterSeparatedMessageSerializer” a binary message serializer was implemented to have an efficient protocol without any redundancies. See figure 10 and figure 11 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 10: 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 11: Broadcast packet structure**

# 5 Results

This chapter describes the results of the project.

TODO more text

## 5.1 Chord

Divided by the sub-headings this chapter will describe the results of each defined goal.

TODO more text

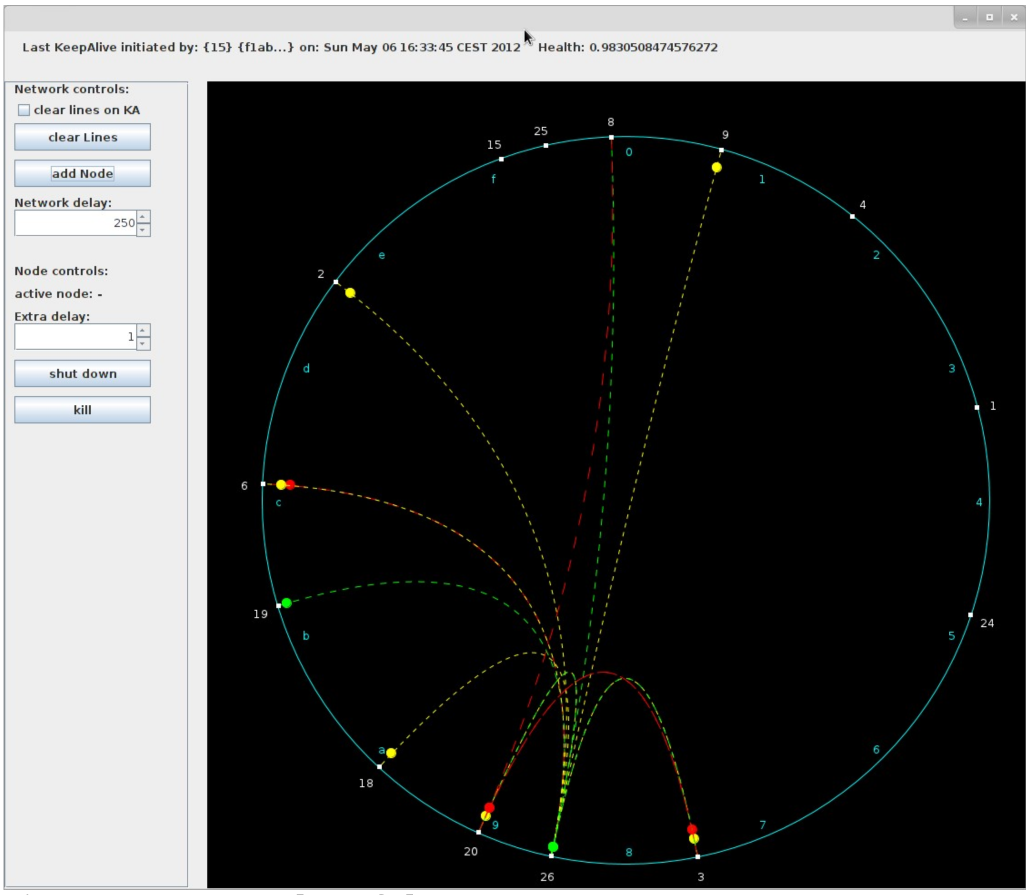
### 5.1.1 Simulation environment

The simulation environment contains 2 input / output methods. The first is a command line that allows every possible action on the DHT and 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, where each command has a different amount of parameters or no parameters. A command is always one contiguous word. If parameters follow, they are separated by a coma from each other and with a space from the command. 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 like transmitted messages, node changes and finger changes
- Print node information like fingers, id and network address
- Print the health of the DHT

- Control other simulation functions like the GUI and the statistics function



Last KeepAlive initiated by: {15} {f1ab...} on: Sun May 06 16:33:45 CEST 2012 Health: 0.9830508474576272

Network controls:
☐ clear lines on KA
Network delay: 250
Node controls:
active node: -
Extra delay: 1

**Figure 12: Screen-shot of the GUI**

In the GUI the DHT is visualized in a circle that represents the hash space of the DHT. [Insert picture] 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 a node with the mouse the current finger table of that node is displayed in a purple. Every 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 which finger action was monitored as described in the following list:

- New finger added
- Finger removed
- Better finger added (that replaces a worse finger)

- Worse finger removed (that was replaced by a better finger)

Every finger change is added up. 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 good visualization on which effects the last keep-alive event had on the health of the DHT. The GUI further allows some basic control of the DHT. That is adding nodes, removing nodes and changing the network and node delay.

When the amount of events is 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 like using in a text editor. The log file is automatically created every time the simulation environment is started and is saved to a file in the users directory called “media_sense.log”. 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.

### 5.1.2 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 for better understanding. The graphs will show the described scenarios from chapter 3.3.

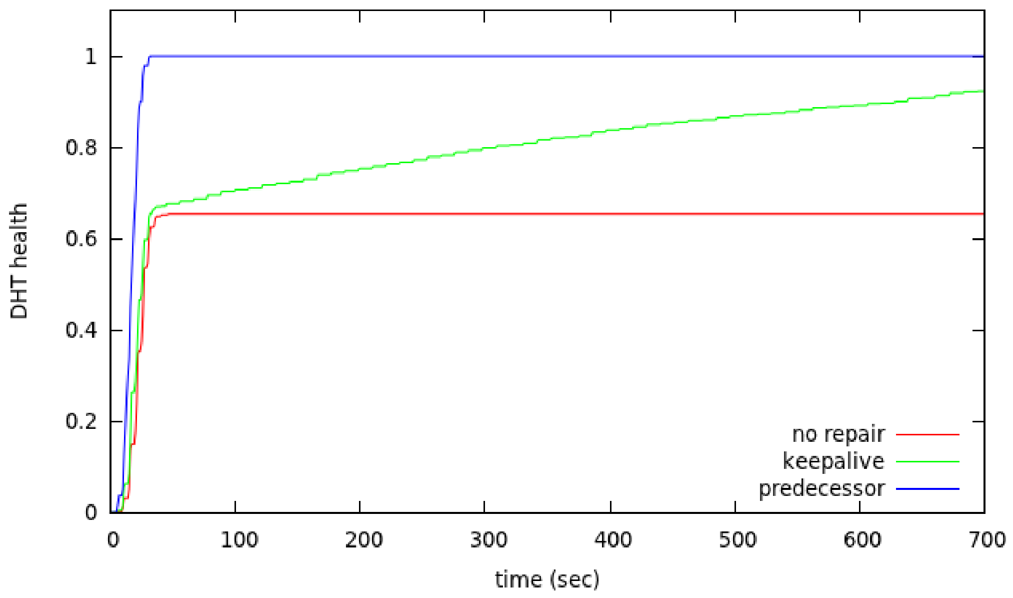


**Figure 13: Health in different development statuses**

Figure 13 shows the DHT health in relation to the passed time. The data lines have been recorded at different development statuses, 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 point, when the DHT solution did not have any repair algorithm. It shows that the DHT health were never better than approximately 0.6 what means that 60% of all fingers in the DHT are the right ones. When a regular sent keep-alive message (green line) has been introduced, the health did not stagnate at a certain point, but needed a very long time to get until 100%. Because of that, it has been introduced that each node knows additionally to the finger table its preceding node. With those extra information it is possible to have 100% health immediate after the insertion of all nodes.

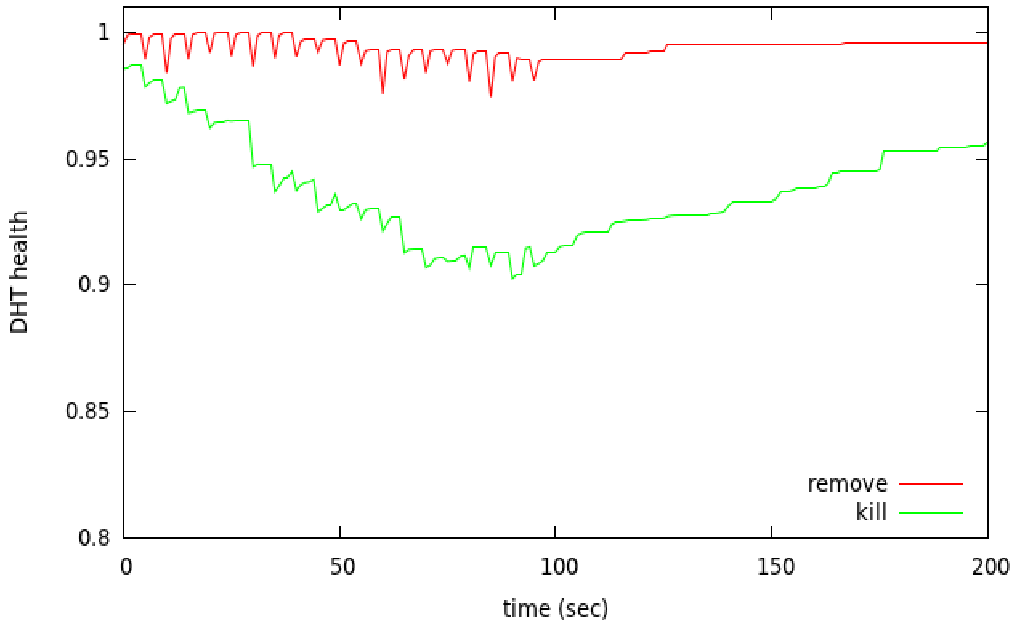


**Figure 14: Comparison between leaving and failing nodes**

Figure 14 shows two scenarios. In both cases first 500 nodes were added to the DHT and after that the statistic was started. After the start of the statistic 5 nodes left the DHT in a regular interval of 5 seconds until 100 have left (100 seconds on the vertical axis). The lines show the different behavior of nodes leaving the DHT regularly (red line) and nodes that fail (green line). Because the regularly leaving nodes send leave notifications with helpful information to other nodes, the health is immediately at a value of nearly 100% again. That is not the case if the nodes fail (for example imagine the nodes just loosing network connection), here you can see that the health drops to approximately 90% and after that slowly returns to the direction of 100%. This is because failing nodes can not send the useful information to other nodes in the first moment, but the needed information propagates with help of keep-alive messages slowly.

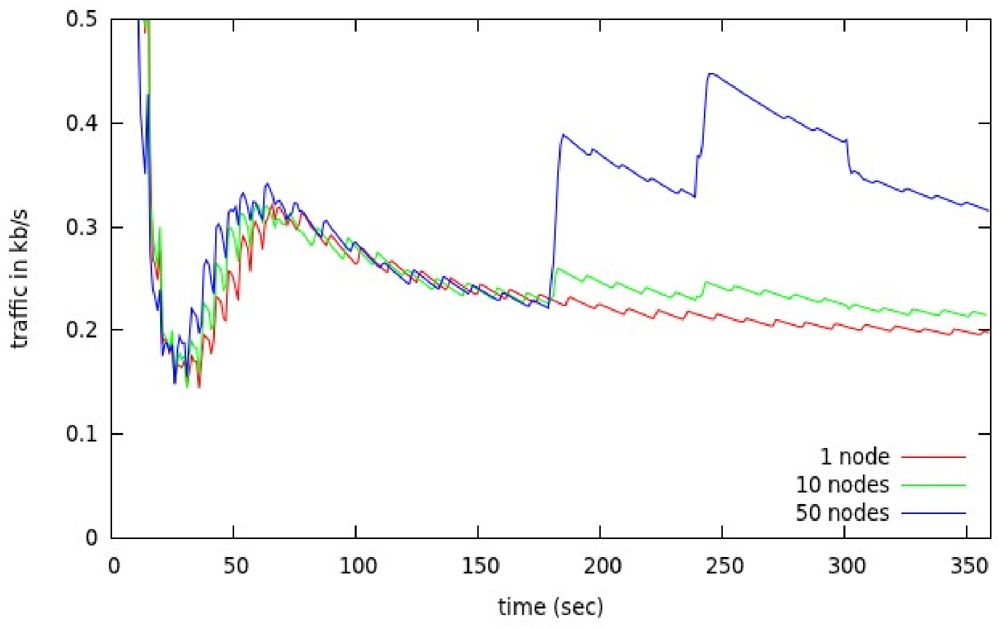


**Figure 15: Average traffic per node per second**

TODO insert text flo

## 5.2 Reliable UDP

TODO TEEEXT

### 5.2.1 Simulation Environment

The simulation environment does not have any user input methods. After startup it starts immediately with transmitting data at maximum speed between two sockets using the loopback device “localhost”. The transmission speed is displayed in MB/s and is updated every 500 ms. To simulate bad traffic conditions the Linux command line tool “tc” has been used. If the connection is so bad that the RUDP implementation considers the link as failed, the application stops with an error message.

### 5.2.2 Comparison of RUDP and TCP performance

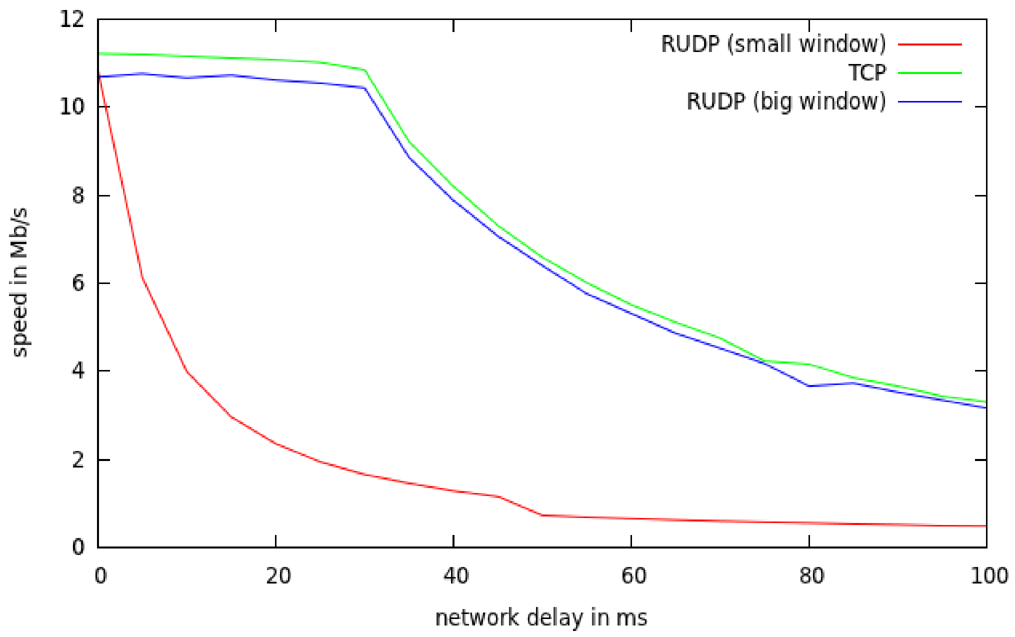


**Figure 16: Comparison between RUDP and TCP with network delay**

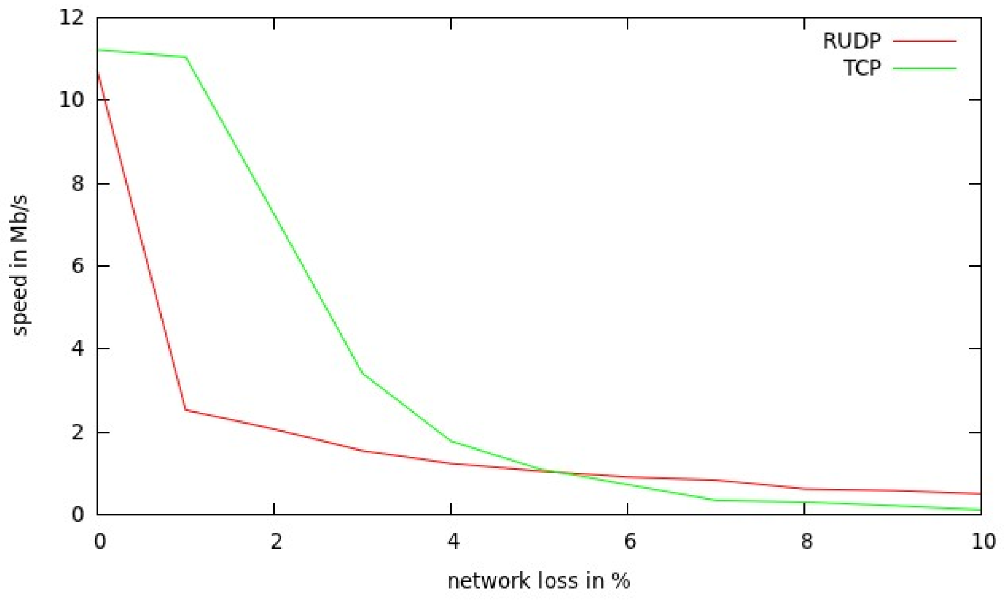


**Figure 17: Comparison between RUDP and TCP with network delay**

The measurements will include the following parameters:

- packet loss in percent

- packet reordering in percent
- packet delay in ms

and the measured result will consist of

- Fail: yes / no
- Speed in Mb/s

# 6 Conclusion

TODO here will be the conclusion

## 6.1 Future Work

During the project we figured out some possible improvements that would not fit into the given effort of the project. Also some implementations were out of scope of this project but would be nice to have. These could be subject to future work.

Both parts Chord and RUDP have been tested in laboratory environments and need more practical testing.

### 6.1.1 Chord

### 6.1.2 Reliable UDP

All parameters of the RUDP implementation like acknowledge time, window size, and more are fixed. The performance might be increased by finding appropriate values or better by implementing algorithms that vary the parameters depending on the available context like average round trip time and others.

Security vulnerabilities have not been taken into consideration during development. They have to be figured out and countermeasure have to be implemented.

It would be nice to have a C/C++ implementation so RUDP is not only bound to Java.

# 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, 2011-18-6

[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/CallaMirani CHORD/alternatives.html

[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

[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

<u>Underlined</u> - 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** *<u>address</u>*
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)
18:03:26.046 | MSG: type: MSG-CHECK_PREDECESSOR_RESPONSE - from: (2) - to: (1) | preHash: {b658...} preAddr: (0)
18:03:28.759 | Node: {b658...} Addr: 0 initiated KEEP-ALIVE
18:03:29.011 | BROADCAST:{da4b...} -> {356a...} | MSG: type: KEEP-ALIVE - from: (0) - to: (2)
18:03:29.011 | BROADCAST:{356a...} -> {b658...} | MSG: type: KEEP-ALIVE - from: (0) - to: (1)
```