Implementation and Evaluation of a new Peer-to-Peer Module for the Network Simulators OMNeT++ and OverSim

Bachelor Thesis

by

Stefan Schmid

born in
Leverkusen

submitted to

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

September 2014

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

Abstract

Gnutella is an open, decentralized peer-to-peer seach protocol, mainly used for file sharing application. This protocol is open for every peer that wants to participate. The name Gnutella represents an entire network of peers which run Gnutella applications and thus form one virtual network. Since setting up a real Gnutella network and running tests with different parameters , like varying numbers of participants, search terms and connectivity is very inconvenient and costly, the virtual network simulator OMNeT++ in addition to OverSim is used.

Assignment

The implementation and evaluation of a new module for the OMNeT++ network simulator using OverSim called Gnutella since it is has not been done before. This involves a lot of planning since a functional network of servents, the servents themselves, their neighbors and the all messages need to be written and later implemented in C++ and integrated into the simulator. It focuses on problems like scalability and increase of data volume inside a Gnutella network. The goal is to evaluate this Gnutella implementation taking all pros and cons of Gnutella networks into account and to point out the boundaries of this implementation.

Results

The evaluation and the simulation results shows that this implementation does not scale very well with an increase of servents participating in the network and thus does suffer from the same problems like the first Gnutella protocol does. These are, for example, the exponential growth of network traffic due to Pong and Query Message flooding. There are also a few security problems with Gnutella, which were not adressed in the implementation itself. Different simulations were run in order to verify that this implementation does work based on the basic principles of the original Gnutella 0.4 stable protocol.

Acknowledgments

A lot of people supported me during my work on this thesis to whom I wish to express my gratitude.

I want to thank my family and close friends, in paticular my girlfriend who was very supportive and patient over the last few months. She also helped me to find minor mistakes and helped me correcting them.

Thanks to Tobias Amft who mentored and proofread this thesis.

Thanks to OverSims Gia module teaching me most of needed understanding in order to implement the Gnutella module.

Thanks to the OMNeT++ Google Group which helped me debug my program.

And my last thanks go to Andr'as Varga, who developed OMNeT++. Without him this thesis would never have existed.

Contents

List of Figures **ix**

1. Outline **1**

2. Gnutella Protocol v0.4 **5**

3. About OMNeT++ and OverSim **15**

4. Gnutella Implementation using OMNeT++ and OverSim **19**

5. Evaluation of the Gnutella Implementation **23**

6. Conclusion **31**

A. Installing OMNeT++ and Oversim **33**

B. Simulation Result Tables **41**

Bibliography **43**

List of Figures

2.1. An Example of Query and QueryHits forwarding. 14

4.1. Random Network with 15 Terminals sending a Query message 21

5.1. Delivered QueryHits with max 3 neighbors 25
5.2. Delivered QueryHits with max 4 neighbors 26
5.3. Delivered QueryHits with max 10 neighbors 26
5.4. processed Query Messages with max 3 neighbors 27
5.5. processed Query Messages with max 10 neighbors 28
5.6. processed QueryHit Messages with max 3 neighbors 28
5.7. processed QueryHit Messages with max 10 neighbors 29

A.1. Terminal after ./configure command 36
A.2. Terminal after make command 37
A.3. Example of a freshly opened OMNeT++ Session 38

Chapter 1.

Outline

The remainder of this thesis is structured as follows. The second Chapter of this these deals with the historical background of the first Gnutella protocol in order to convey the basic idea behind it. The third Chapter explains the official Gnutella 0.4 RFC in detail and reveals some of its problems. The fourth Chapter explains the principle of OMNeT++ and illustrates what can be used in order to build your own modules and how it works in general. Chapter five gives a detailed description of the following Gnutella implementation and explains how different aspects of the RFC were translated into this module and why different changes had to be done in order to guarantee solid simulation results. Chapter six enumerates and comments in detail about the changes that had to be done in chapter five. The last Chapter seven presents different simulation results and interpretes them in order to verify or falsify the correct implementation of the Gnutella protocol.

Underlay and Overlay Networks

Since this bachelor thesis is based on the construction of an overlay network which is created by Gnutella. I will shortly explain some terms concerning the topic of my thesis.

Underlay Network

An underlay network represents the physical interconnections of a network and thus represents the physical layer. The physical layer can be divided into four seperate layers. These are:

Site Layer: The Site Layer is the physical representation of all houses and places, where participants can connect to the underlay via wire.

Optical Layer: The Optical Layer consists of the optical wires, where data can be sent through, connecting the houses to each other.

SONET/SDH Layer: Synchronous Optical Networking (SONET) and Synchronous Digital Hierarchy (SDH) are standardized protocols that transfer multiple digital bit streams over the optical layer. It is used to forward information through the cables.

IP Layer: The IP Layer can be refered as the first overlay, but you can also include it within in the underlay network. The IP Layer manages and uniquely assigns different IP-addresses to different clients, and thus enables direct communication between clients. Because IP-Addresses are not bound to the physical location of a system, and can be assigned randomly, the IP Layer may be defined as an overlay network. It inhabits characteristics of both the overlay and underlay.

Overlay Network

An overlay network is a layer of virtual network topologies on top of other networks. Overlay networks provide us with many advantages to handle the increasing growth of internet information and resources.

1. Overlay Networks allow developers and users to easily design their own enviroment and protocols on top of the IP layer, or other overlays, such as data routing and file sharing management, which is the most important aspect of this bachelor thesis.

2. Data routing in Overlay Networks are very flexible, because they can quickly detect and avoid network congestions, through message timouts and routing tables.

3. End-Nodes in Overlay Networks are highly connected, due to flexible routing (see (2)). As long as any physical connection exists, two end-nodes are able to comminicate with each other.

4. This high connectivity allows users to share a huge amount of information and resources available in the internet.

Examples of typical overlay networks are multicast overlays, peer-to-peer overlays (e.g. Gnutella (item of this bachelor thesis)), parallel file downloading overlays (e.g BitTorrent), routing overlays (e.g Skype)

There are a few disadvantages of overlay networks. Overlay networks have no information about the status of the underlay network. They have no control about the physical network and they lack important information, like the degree of capacity utilisation and other critical information. This leads to to an inefficient network resource usage, such as a mismatch between overlay and underlay topology or inaccurate probing results among end-to-end nodes. This and other factors generate a large amount of redundant information roaming around the network and thus unnecessaryly increases the network traffic. Overlay networks are open to all kinds of Internet users, this leads to serious issues regarding security and privacy. [1] The decentralized nature of overlay networks is likely to have a weak ability for resource coordination. Fairness of resource sharing among end-nodes in overlay networks is a not well adressed issue.

Chapter 2.

Gnutella Protocol v0.4

History of Gnutella

Gnutella was developed in early 2000, but released on March 14, 2000 by Frankel and his Nullsoft colleague Tim Pepper. The Gnutella client was distributed as a download on the Nullsoft webserver, despite the fact that Nullsoft was acquired by AOL and without its knowledge. This release day was announced on several websites specializing on technology and this lead to over one thousands downloads on the first and only night. The source code was later released, under the GNU General Public License (GPL). The following day AOL found out about the distribution of the Gnutella client and stopped the availability immediately in fear of legal actions. Additionaly they restrained Nullsoft from continuing their work on the project. [2]

But this event had just a minor impact on the spread of the Gnutella client. It took a few days before the reverse engineering of the source code was successful, and more compatible free and open source alternatives began to appear.

A Gnutella network is an alternative to semi-centralized systems, which were used in that time in order to spread information and files around the Internet, because it is fully distributed and thus decentralised. Gnutella allowed users to share any type of file with no restrictions. Gnutella had several other advantages over any other file-sharing systems at that time. For instance: Gnutella does not have a single point of failure. This means that it is nearly impossible to shut down an existing network of Gnutella peers, by powering off a specific server. Other file sharing systems do have a single point of failure, due to the fact that a centralized server exists, where all the file sharing data is stored. If one shuts down one of those servers, the whole network collapses. Gnutella does not rely on any centralized servers, so it cannot be taken down. The huge popularity induced a potential legal demise by another company, with the intention of protecting their own distribution system.

As this network grew even larger, the limits of the initial release of the gnutella protocol were revealed. It is scaling very badly with increasing number of peers. This problem was worked on and an improved variation on the protocol was released in early 2001. These improvements included a better scalabillity, by the creation of so called "ultrapeers" which are routing search requests and responses for users connected to them, while the old version treated every user as a client and a server at the same time.

Today the word Gnutella does not refer to this one project alone, but to the open protocol, which is used by various clients and was implemented and evaluated for OMNeT++/Oversim as part of this thesis.

The name "Gnutella" is a combination of the two words "GNU" and "Nutella", the name of a very popular hazelnut flavored spread. Frankel and Pepper ate a lot of this spread, and thus decided to name their project after this. They intended to release the finished product under the GNU General Public License, which they succeded, and thus creating this combination of words. Gnutella is pronounced with a silent 'G', making it sound even more similar to "Nutella".

The Gnutella Protocol Specification v0.4

Design Goals

Gnutella was designed to achieve the following aims within a very basic set of rules. Most importantly, Gnutella networks are supposed to be dynamic. That is, servents should be allowed to join and leave at any given time. Altogether, this goal was very well met. Furthermore, Gnutella networks should be highly scalable. This aspect can be considered the most problematic since the network works well with just a few servents but drops fast in efficiency if the network grows. Moreover, all servents shall remain anonymous in the network while participating. This is a basic requirement for every P2P network and as such was well achieved in Gnutella. There are, however, a few P2P networks that do not preserve anonymity.

Summary

Basically, Gnutella is a protocol for distributed search. Most commonly, it is used in peer-to-peer networks but can also be applied to traditional client/centralised server networks. It should be noted however, that in the Gnutella protocol every client is also a server, and vice versa. The Gnutella protocol refers to these clients as "servents". Servents provide a client-side interface and can issue

queries as well as view search results. At the same time, they are able to accept queries from other servents in the network. Additionally, servents check for matches against their local data set and respond with corresponding results. Since the Gnutella protocol is often used in peer-to-peer networks, it is highly fault tolerant and continues to work well even if servents go offline.

1. Connection Procedure and Protocol Negotiation:

When one servant has obtained the IPv4 address and the port number of another servent in the network, a TCP connection can be created, and the following Gnutella connection request string may be sent:

```
GNUTELLA CONNECT/<protocol version string>\n\n
```

where in this version <*protocol version string*> is set to be "0.4", because we are working with version 0.4 of the Gnutella Protocol here.

A servent willing to accept this TCP connection has to respond with:

```
GNUTELLA OK\n\n
```

Any other response indicates that the servent is not willing to accept the connection.

2. Protocol Definition:

The Gnutella protocol defines several descriptors for their servents, which are used for communication between servents. Currently, the following descriptors are defined:

Ping: Ping is used to discover other hosts on the network. After a servent has received a Ping descriptor it is expected to respond with the Pong descriptor.
Pong: Pong is the response to a Ping descriptor. A Pong descriptor always consists of the address of a Gnutella servent and information regarding the capacity it is willing to share on the network.
Query: This is the primary search mechanism on the network. After a servent has received a Query descriptor it is expected to respond with the QueryHits descriptor, if a match is found against its local data set.
QueryHits: QueryHits is the response to a Query descriptor. A QueryHits descriptor always provides enough information to acquire the data matching the corresponding Query.

Push: Push is used to allow a servent behind a firewall to contribute file-based data to the network. This is not used in Omnet++, because there are no servents behind firewalls.

It is only required for a servent to connect to one other servent in a network, when it joins a network for the first time. The acquisition of additional servents is not part if the protocol definition.

3. Peer-to-Peer Gnutella Packets: Descriptors:

After a TCP connection with an other servent has been established, they can send and receive Gnutella protocol descriptors. Each descriptor is preceded by a Descriptor header with the structure below.

3.1 Descriptor Header

Fields	Descriptor ID	Payload Descriptor	TTL	Hops	Payload Length
Byte offset	0...15	16	17	18	19...22

Descriptor ID: A 16-byte string uniquely identifying the descriptor on the network. The Descriptor ID must be final, and must not be changed when forwarding messages between servents. This is used to identify cycles, when any servent receives a message that it sent before, and thus reduces the traffic on the network.
Payload Descriptor:
0x00 = Ping
0x01 = Pong
0x80 = Query
0x81 = QueryHits
0x40 = Push (not used here)
TTL: TTL stands for Time-To-Live, and stands for the number of times the descriptor will be forwarded by servents in the network before it is removed. Each servent must decrement the TTL before forwarding to another servent. When the TTL hits zero, the descriptor must not longer be forwarded.
Hops: Hops stand for the number of times the Descriptor has already been forwarded and must be incremented by each servent before forwarding. The TTL and Hops fields in the header must meet the following conditions:

$$TTL_i + Hops_i = TTL_0$$
$$TTL_{i+1} < TTL_i$$
$$Hops_{i+1} > Hops_i$$

where TTL_i and $Hops_i$ stands for the value of TTL and Hops after the ith hop, for $i \geq 0$.

Payload Length: The Payload Length determines the length of the descriptor following this header. This means that the next descriptor header is exactly *Payload Length* bytes away from the end of this header. There are no gaps or bad bytes between two different descriptors in the Gnutella data stream.

Be aware that the TTL is the only mechanism for dropping descriptors and that the Payload Length is the only mechanism for finding the next descriptor header, because the Gnutella protocol does not provide any other means of "highlighting" decriptor headers, for example certain "Eye catcher strings" or synchronisation methods.

3.2 Descriptor Payloads

The Descriptor consists of a header and and optional payload, whose content depends on the Descriptor Payload field described above. This section details them further.

3.2.1 Ping (0x00) Descriptor Payload

Fields	Optional Ping Data
Byte offset	0 ... L-1

The standard Ping descriptor has no defined payload and is of zero length. A Ping Descriptor is represented by only the descriptor Header whose Payload Descriptor field is 0x00 and the Payload Length field is 0x00000000. A servent uses a Ping Descriptor to actively discover other servents in a network. Any servent who recieved a Ping may or may not respond with a Pong Descriptor which contains his own adress and the amount of data it is willing to share. There is no definition on how often a Ping Descriptor may be sent but there are several implementations which try to minimize the network traffic caused by Ping Descriptors.

3.2.2 Pong (0x01) Descriptor Payload

Fields	Port	IP Adress	Files Shared	Kilobytes Shared	Optional Pong Data
Byte offset	0...1	2...5	6...9	10...13	14...L-1

Port: The Port field contains the TCP Port number on which the responding host can accept incoming Gnutella connections.
IP Address: The IPv4 address of the responding host.
Files Shared: The number of Files that the servent with this IP address and port is sharing on the network.
Kilobytes Shared: The number of Kilobytes of data that the servent with this IP address and port is sharing on the network.
Optional Pong Data: Optional field and may contain further extensions for future versions of the Gnutella Protocol.

3.2.2.2 Port Numbers in Standard Pong Descriptors:

Standard and default Port numbers:

There is no defined *standard* port number for servents, and they may use whatever Port they feel fits. Every Port number between 1 and 65535 is valid, even though Gnutella servents use TCP Port number `6346` by default.

Gnutella Port Number and Download Port Number:

The Port number advertised in Pong descriptors may vary from the port number advertised in Query-Hits to enable download requests. They may assign the same TCP port for incomming HTTP connections used by download requests.

Non null Port numbers:

If the Port number is not null, then this indicates support by the servent for incoming Gnutella TCP connections.

Null Port Numbers:

Servents that receive a null port number in an incoming Pong should discard this Descriptor and

should not forward it to another servent, as this indicates that a direct Gnutella connection via TCP to the sending Host is not possible.

3.2.3 Query (0x80) Descriptor Payload

Fields	Minimum Speed	Search Criteria String	NUL Terminator	(Optional) Query Data
Byte offset	0...1	2...N	N+1	N+2...L-1

Port: The Port field represents the TCP Port number on which the responding Host can accept incoming Gnutella connections.
Minimum Speed: The minimum Speed (measured in *kbits/sec*) which is required for servents in order to reply to this message. A servent should only answer to a Query Descriptor with a QueryHits, if it is able to communicate at this speed or higher. Hints on how this field may be set are the following: [3]

0 = results are sent, regardless of available upload speed. Even if there is no free upload slot.
1 = any results will be accepted which can be transferred at a guaranteed minimum of 1.5 kb/s.
between 2 and 32767 = (currently available downlink bandwidth) * 0.7.
> 32767 should not be used in the Gnutella 0.4 Protocol

In order to determine which uplink speed can be guaranteed, the Minimum Speed field value n of the Query can be compared to:

$$n = \frac{\min[(\text{maximum uplink bandwidth}) \cdot 70\%, (\text{total unused uplink bandwidth})]}{[(\text{uploads in progress}) + 2]}$$

Search Criteria String: This search string is always terminated by a NUL (0x00). The maximum length is limited by the Payload Length field of the Descriptor header.
Query Data: The Query Data field is optional and not required for the Gnutella 0.4 version, and thus it is not further explained here.

3.2.4 QueryHits (0x81) Descriptor Payload:

Fields	Hits	Port	IP Address	Speed	QHD Data	Servent Identifier
Byte offset	0	1..2	3...6	7...10	10+N...L-17	L-16...L-1

Hits: The number of query hits in the Result Set field, which is explained below.
Port: The port number on which the responding host can accept incoming TCP connections.
IP Address: The IPv4 address of the responding host.

Speed: The maximum upload speed in kbits/second between 0 and 32767 of the responding host.
QHD Data: This field is optional and it is reserved for future extensions of the Gnutella 0.4 protocol. It is not used here.
Servent Identifier: This string is 16 byte long and identifies the responding servent on the network

QueryHits Descriptors are only sent in response to an incoming Query Descriptor. A servent should only reply to a Query with a QueryHits Descriptor if it contains data that strictly meets the Query Search Criteria. A QueryHits Descriptor must be initially generated with $Hops = 0$ and the TTL field equal to the number of Hops of the Query Descriptor it is responding to.

The Descriptor ID field in the Descriptor header of QueryHits must contain the same value as that of the associated Query Descriptor. This allows a servent to identify the QueryHits Descriptors associated with Query Descriptors it has generated.

4. Descriptor Routing

A Gnutella servent must route protocol Descriptors according to the following rules.

1. Pong Descriptors may only be sent along the same path that carried the incoming Ping Descriptor. This has the effect that only those servents that routed a Ping Descriptor receive the corresponding Pong Descriptor. A servent that receives a Pong Descriptor with $DescriptorID = n$, but has not seen a Ping Descriptor with $DescriptorID = n$ must remove the Pong Descriptor from the network and is not allowed to forward it to any other connected servent.

2. QueryHit Descriptors may only be sent along the same path that carried the incoming Query Descriptor. This has the effect that only those servents that routed a Query Descriptor receive the corresponding QueryHit Descriptor. A servent that receives a QueryHit Descriptor with $DescriptorID = n$, but has not seen a Query Descriptor with $DescriptorID = n$ must remove the QueryHit Descriptor from the network and is not allowed to forward it to any other servent.

3. A servent should forward incoming Ping and Query Descriptors to all of its directly connected servents, except the one that delivered the incoming Ping or Query.

4. A servent must decrement a Descriptor header's TTL field, and increment its $Hops$ field, before forwarding the Descriptor to any directly connected servent. if, after the decrementing the header's TTL field, the TTL field is equal to zero, the Descriptor is not forwarded along any connection and gets dropped.

5. A servent receiving a Descriptor with the same Payload Descriptor and Descriptor ID as one it has received before, should attempt to avoid forwarding the Descriptor to any connected servent. This reduces network traffic and bandwidth consumption.

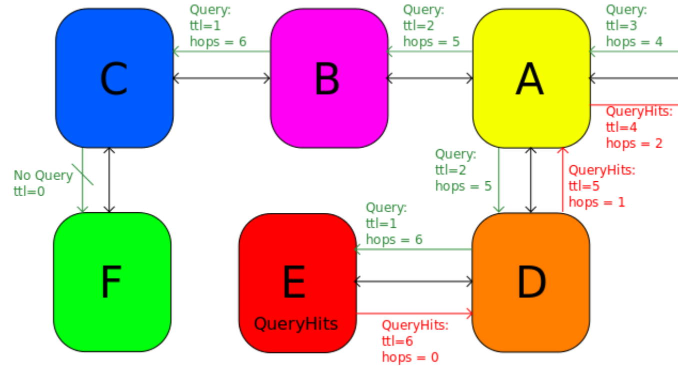

Figure 2.1.: An Example of Query and QueryHits forwarding.

Routing Example:
Figure 2.1 displays a routing example where Servent A receives an incomming Query Message and forwards it to Servent B and D since they are direct neighbors of A. The Query Message cannot reach Servent F , because its TTL value reaches zero upon arrival at Servent C and as a result gets dropped. Servent E holds the data requested by the incomming Query Message and responds with a QueryHits Message which gets routed back the way the corresponding Query took.

Drawbacks of Gnutella v0.4

Theoretically, Gnutella provides a convincing approach regarding the amount of independent servents processing messages and concerning how fast and easily messages are forwarded since there are no requirements for each servent except a Gnutella speaking application to join a Gnutella network, and to forward any message to every neighbor without having to decide what the best option is. In real-life networks however, it has to deal with some severe drawbacks. Not every servent in a network participates with an equal amount of files. To the contrary, an analysis of several Gnutella networks has shown that 50% of all files are served by a small amount of servents. [4] As a result, the network behaves more like a client server network than a peer-to-peer network.
Network bandwidh can be occupied by malicious data masquerading as ordinary Gnutella messages, causing severe security problems. Due to the open nature of a Gnutella network, which can be joined by any servent, problems with malicious servents may occur, harming the network with Pong or similar attacks. During a Pong attack a malicious peer floods the network with Pong messages, aquiring the IP address and port of any non-malicious peer responding. This can be exploited insofar as distributed denial of service attacks can be launched through spam messages and actively sabotage the Gnutella network. Additionaly, any servent can lose their anonymity through GUID tracing.

Chapter 3.

About OMNeT++ and OverSim

OMNeT++ is a discrete event simulation enviroment which has been developed by Andr'as Varga at the Technical University of Budapest, Department of Telecommunications. the moment where this thesis was written, the most recent Version of OMNeT++ is 4.4.1 from October 2014 and OMNeT++ has been around since the middle of the 90'. [5] The kernel and Utility classes are written in C++. The IDE is based on Eclipse, so that is why a Java Runtime Enviroment is required for installation. OMNeT++ is one of the most common simulators, despite the fact, that OMNeT++ does not itself count as a network simulator, but as an infrastructure of all sorts of simulation. The Framework uses its own public license (Academic Public license), which grants similar rights like GNU GPL. OMNeT++ has a large active community and a heavy used mailing list.

According to the official handbook [6], OMNeT++ supports:

- modellizing of wired and wireless communication networks.
- implementation of protocols
- modellizing of queuing systems
- modellizing of distributed hardware systems
- verification of hardware achitectures
- evaluation of complex software systems
- any application possible, which relies on discrete events and communication

NED files

OMNeT++ uses C++ to implement the behavior of nodes, but additionaly uses the domain specific language NED, which stands for "Network Description". The reason for this is, according to Andr'as Varga, that XML is too confusing, while scripting languages, like phyton and Tcl, do not support editing with texteditors and graphical modelling tools. [7] Since OMNeT++ 4.5 NED became a very useful language. It is now possible to create hierarchies, in order to describe more complex models. Using Interfaces, it is possible to write several implementations for one module with different parameters. Additionally NED is now able to utilize inheritance. This makes it possible to derivate from a TCP client to a FTP client. NED is able to use inner types, annotations with metadata (e.g graphical layout) and a package structure resembling the java package structure.

These features make it possible to reuse well written components and component libraries like INET since all features can directly be used without any further installation or configuration.

Messagecompiler

Messages in OMNeT++ are exchanged as C++ Objects. In order to save time you can use the messagecompiler "opp_msgc", to easily generate C++ classes. Here is an example:

```
FirstMessage.ned

packet MyFirstMessage{
    string message;
    int payload;
};
```

Using the opp_msgc compiler, two files are generated. One file is called `FirstMessage_m.cc` and the other file is the corresponding header file `FirstMessage_m.h` . These files can now be used in your overlay, like every other C++ source file.

Summary of the existing Documentation

The OMNeT++ Manual has more than 300 pages, not including the addendum, in addition to many tutorials and guidelines which can be found online on the OMNeT++ homepage. The "TicToc Tutorial" is worth mentioning, because it helps alot to understand how OMNeT++ works, and teaches the user alot about the functional range of this simulator. There is also a very active Google Group, where many people look for help on a daily basis, extra to the up-to-date wiki about OMNeT++. There are many different sources of information about the usage of OMNeT++.

Modelling, Execution and Analysis

During the installation process of OMNeT++, the compiliation process takes up most of the time. A few packets are required, which are listed in the installation manual. Additionally, the source code must be downloaded and a configuration script must be created. A detailed description of the installation process is found in the addendum.

The first step in modelling is the creation of a network in OMNeT++. One has to write a NED file using any text editor or one can use the graphical interface to do so. It is possible to switch anytime between the graphical and the code representation of the NED file. Secondly you have to implement the described functional modules, using C++. They are derivations of *cSimpleModule* and at least one function *handleMessage()* is required for them to work. Finally one can gather everything in a configure file called *omnetpp.ini* , where you can select the network, the number of cycles and even determine a seed for the randomizer. You can customize and create your own configuration files for different simulations. *Omnetpp.ini* is the entry point into the simulation, where the parameters of ones simulation are defined.

One can start and select the simulation cycles from the IDE, and one can utilize the build-in visualisation. This happens using the graphical interface Tkenv . On default only the Top-Level-modules are visualized, and the communication inside the network is animated. This happens simultaneously in simulation time and visualisation. This means, that the moment it is visualized the very same moment it is simulated. This can come in handy for debugging purposes, but you can not rewind the simulation, although it is possible to slow down, go step-by-step or even speed up the simulation. The simulation visualisation window can be closed without the simulation stopping, it will increase in computation speed since no events need to be visualized anymore.

Statistical Outputs

There are three different kinds of results and output formats.

- scalars (single value)
- vector (many values)
- statistics (e.g diagram)

Those are saved as text in seperate files. There is a tool for the visual analysis already included in the IDE. Several programs and libaries are enumerated in the handbook, which can be used to evaluate the research data, like `NumPy, SciPy, MatPlotLib, MATLAB, Octave, ROOT, Grace` and several other spreadsheets. Of course there is always the possibility to write and use your own output-manager as a plugin.

OverSim

OverSim is an open-source overlay and peer-to-peer network simulation framework for the OMNeT++ simulation framework [8] This simulation framework was mainly used forrunning this simulation and gathering statistical data, by vizualizing the network topology, messages and servents in the network. It provides many useful features like differend types of Churn Generators, see Chapter 5 , and many simulations of existing routing protocols like `Gia, Cord, Kadmilla` and `Pastry`.

Chapter 4.

Gnutella Implementation using OMNeT++ and OverSim

In the following I am going to give a detailed description of my implementation of the Gnutella 0.4 protocol. First of all, I will explain the files required for the protocol's core functions, which thus are essential for it to work properly.

The Gnutella 0.4 Implementation consists of the following header files for the overlay:

```
GnutellaServent.h
GnutellaNeighborCandidateList.h
GnutellaNeighbors.h
GnutellaMessageBookkeeping.h
GnutellaKeyList.h
Gnutella.h
```

And the following header files belong to the application:

```
GNUTELLASearchApp.h
QueryMsgBookkeeping.h
```

The messages forwarded in the Gnutella network are defined in these header files:

```
ExtAPIMessagesGnutella_m.h
GnutellaMessages_m.h
```

Although it is not required in the original Gnutella protocol, an application is used, which uses two timers to start searches and assign key lists automatically, which will be targeted by queries later. Every time a *keyList_timer* runs out, the application takes a predefined maximum amount from a pool of potential keys, creates a message containing these keys, and sends this message to the overlay. If the *search_timer* runs out, the application checks whether the network is still in the initialization phase. If not, it selects a random key that is currently not searched for and sends a message containing this key to the overlay. Besides from gathering additional data for statistical aims, like the numbers of send messages, added neighbors, maximum and minimum amount of peers, average hops of messages, and sent bytes, the application serves no further purpose.

The overlay which consists of several Terminals which will be henceforth referred to as **servents**. They are defined in the `GnutellaServent.h`. Every servent consists of an *uniqueIP* and a *keylist*. It establishes connections to other servents in the network and saves them in a map called *neighbors*. Each of these servents receive a *timestamp* and will be dropped out of the map due to timeout in case it has not respondend for a certain amount of time and thus resetting its timeout. This is quite convenient since the amount of neighbors a servent can have at any given time is limited for evaluation purposes. Since unused connections get resolved, free space becomes available for new neighbors. In the original Gnutella protocol, new potential neighbors are actively discovered by forwarding Ping messages. The problem is now that many sets of cliques are created, if we take this approach, which makes it impossible to gather constistent simulation data. A clique referres in this context to a set of servents in which every servent is connected to each other in the same subgraph. To resolve this problem, the following approach was taken: Once a new servent joins the network, it gets added to a *sharedList*, a pool of all servents that did or do participate in the current network. The list is used in order to artificially create a connected graph with as many servents as possible. This is achieved by connecting every new servent to its predecessor in the *sharedList* in order to create a network which is called *path* in graph theory. Every implemented simulation is based on such a network.

Once a servent has a neighbor as well as a key list, a timer called *addNewNeighbor_timer* starts. Its purpose is to determine at what time exactly a servent randomly selects another servent in order to add a new neighbor. In the course of this process a huge network gets established in which any Query message is able to reach as many servents as possible and thus increasing the chance of receiving a QueryHit.

This approach still creates a randomized network, where cliques can still occur, but it also leads to more reliable simulation resuls. If a servent, called A, wants to add a new neighbor to its list of neighbors it sends a Ping message to an opposite servent, called servent B, and adds the servent B to a list called *possibleNeighbor*. Servent B receives the message and, if it has not already the maximum amount of neighbors, responds with a Pong Message and then adds servent A to its own *possibleNeighbor* list. This Pong message then reaches the servent A and a connection between the

two servents is established. Servent A confirms this connection with a Pong_ack message. Servent B then adds the servent A to its neighbors the moment it received the pong_ack. Now we have a double-sided connection between those two servents.

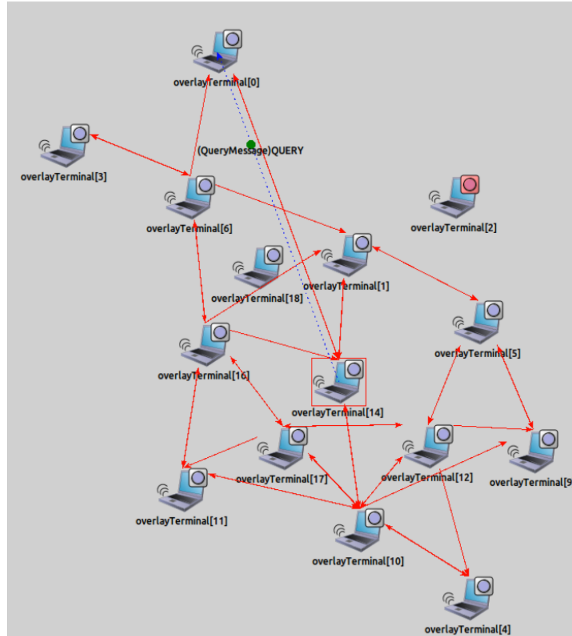

Figure 4.1.: Random Network with 15 Terminals sending a Query message

Now that we have an established, connective network we can start sending queries and other messages through the network. Everytime a query starts, the servent processes the query message and checks if its *keyList* contains the key defined in the QueryMessage. The message gets saved in a *msgBookkeepingList*, which keeps track of all query messages and their origin which this servent has already processed. This is later used in order to decide wether to forward a message or delete a message if we already processed this message before. If a servent does not have a key they forward this message to every other servent in their neighbor list except the one from whom they received the message. This leads to huge network traffic, an effect which is discussed later. If the servent has the requested key it does not forward this query anymore but replies to it by creating a QueryHit message. This QueryHit message contains the *searchKey* and the unique identification of the servent who holds the requested key. The QueryHit message travels back the very same path the corresponding query message took. This is accomplished with the help of the *msgBookkeepingList*. The moment a QueryHit message with the very same ID as the corresponding query message is received, the servent looks into his list and gets the origin from which he first received the query message. This enables the servent to send the QueryHit back on the same path the query message traveled initially. This guarantees that only the servents who forwarded this query message processes the corresponding QueryHit message. If any servents receives a QueryHit message ,without having processed the corresponding Query message in the past, the message gets deleted.

Differences

The purpose of the following chapter is to focus on the differences between the official Gnutella Protocol Specification v0.4 and my OMNeT++ implementation of the protocol. While both do not differ in basic functionality, several adjustments had to be done. Most importantly, the implementation in the OMNeT++ Network Simulator uses the User Datagram Protocol (UDP) instead of the Transmission Control Protocol (TCP). TCP provides several functions which are not required for the simulation, such as the retransmission functionality. Using UDP in an enviroment which utilizes zero package loss, basically, simulates TCP behavior. Since all simulations run with zero package loss, there is no need to retransmit any messages if none are lost while transmitting. Due to the fact that Gnutella does not have a single point of failure, setting package loss to zero does not affect the protocol's functionality. Thus, we have no need of the TCP's function to resend packages. Additionaly, the simulator does not send ACK-Messages for every message received, therefore it is not possible to determine whether a message reached its destination. This also disables the use of congestion and overflow control because servents cannot tell whether a connection is in need of congestion control and thus resends a package at a later time in order to prevent package loss. UDP is better suited for the simulation than TCP since it is easy to implement and to maintain, as well as it reduces network traffic. Hence, evaluation becomes easier since the collected data focuses solely on the protocol. An altered version of TCP's "triple handshake" is used in order to establish direct connections between two servents and to ensure that the opposite servent is reachable.
The simulation utilizes an application whose purpose is to run a timer in order to start new queries and to send those to the Gnutella overlay. Contrary to the original protocol (where every servant starts its own queries), the application allows the user to control how often queries are sent to the network and how many query-hits are received. Furthermore, key lists are sent to every servant that can be searched for. The Gnutella 0.4 protocol specifies five different descriptors [3], including Ping, Pong, Query, QueryHit, and Push. Out of those, only the Push descriptor was left out for implementation. This was due to it being a mechanism that allows a firewalled servent to contribute file-based data to the network. Since the simulation does not include NAT Firewall, the Push descriptor was not rendered benefitial for our purposes. Messages sent in the simulator do not have any optional payload since most payloads include information about upload and download speed as well as information about the overall number of shared files in the network. Several other Desciptor headers, including the Query Routing Table, Bye, Open Vendor, and the Standard Vendor Descriptor, serve as an optional extention of the current Gnutella .04 protocol and thus were excluded from implementation. In the simulator, we do not forward any Ping messages in order to aquire new neighbors. Instead, a global list of all servents in the network is used for randomized neighbor aquisition, leading to a more widespread network. This feature allows to generate many consistent connected graphs and reduces the unintended ocurrance of many cliques with more than three servents. Cliques reduce the Gnutella protocol's efficiency by increasing network traffic and incrementing the arrival of the same query message at any given servent.

Chapter 5.

Evaluation of the Gnutella Implementation

Parameters

In order to validate the correct functionality of the Gnutella protocol in this network simulator we need to collect some data and run some simulations with different parameters. This section presents the collected data and describes what different kinds of parameters were involved. Like in a real world example the amount of participants in a network is not constant. From time to time any servent can drop out of a network. This may be caused by exiting the application or a loss of the internet connection or anyting similar. In order to simulate a drop out of the network OverSim provides a module called *churnGenerator* [9] which can be edited in the *omnetpp.ini* and set for every single configuration. If no churn generator is selected, OMNeT will use the default configuration taken from the *default.ini* file. There are currently four different kinds of Churn Generators defined. The first and most basic Churn Generator is the `NoChurn` Generator. With NoChurn selected, servents will be added until the overlay has reached its defined maximum amount of servents. After this, the network will remain static and no further servents will be added or removed. The second and most common Churn Generator is called `LifetimeChurn`. LifetimeChurn picks a random lifetime for each servent, created from a given probability function. After the lifetime reaches zero the servent will drop out of the network and another servent will be created after a certain *deadtime*, which is determined by the same probability function. Note that each servent which dropped out does not actively disconnect from the network. This means that a network using the lifetime churn is dynamic and it is even possible to have a few more terminals running in the overlay than the maximum amount. This does also mean that there can be less terminals running in the overlay than the maximum amount. The next type of Churn is called `ParetoChurn`. It functions similar to LifetimeChurn except the selection of the random lifespan is determined by a two-stepped process. [10] The last type is the `RandomChurn`. In a fixed interval a random number is drawn. This number decides whether a random servent gets added, removed or migrated.

Of the four options mentioned before, LifetimeChurn was selected for all simulations because it provides the best compliance with real-time events without sacrificing too much runtime like ParetoChurn does.

In the Gnutella configs the maximum amount of keys present in the network is set to 200. This translates to 200 different files in a real-life Gnutella network. This number directly affects the number of QueryHits. It is more likely to find a requested file in a network if the amount of available files is limited. The aim was to get reliable data about the number of sucessful QueryHits without huge deviations from this average value.

There is also a setting called *keyLength* which determins the length of every searchKey in the network in nibbles (half a byte). This affects the size of Query, QueryHit and keyList messages from the application. This option was set to 40 and thus reducing the maximum size of these messages. Since TCP connections were not used and do not have the ability to use flow or congestion control, this option does not affect the simulation very much. The bigger this setting the longer it takes for each servent to process messages and as a result slowing down the collection of data. A small value was choosen in order to speed up the simulation. While these effects are neglectable with a small amount of servents, it had a noticable effect on higher amounts of servents (more than 700). Due to Gnutellas bad scaling with growing number of servents [4] it was important to preserve as much runtime as possible and focus on the number of queries compared to the number of QueryHits.

The following simulation results were tested with a varying amount of servents. All simulations were run with 42, 140, 700 and 1400 servents since those numbers were all mutiples of Gnutellas default TTL = 7. Other numbers are also possible in order for the simulation to work. With varying number of servents there was another parameter choosen to be focused on. A few more simulations were run with varying maximum number of neighbors each servent has. One simulation with 3 neighbors, another with 4 and 5, and last 10 neighbors. With a growing number of neighbors the network gets more interconnected and more successful querys can be expected since query messages can spread around more easily and reach more different servents. But on the other hand this also hugely affects network traffic rendering it unscalable. These results focus on the number of fowarded query messages as well as on the number of sucessfully delivered QueryHits messages.

Simulation Results

These simulations were run in order to verify that this gnutella implementation does suffer from the bad scaling like the real-world application does, focusing on the Gnutella descriptors and events like QueryHit, and Query Messages. The simulation was repeated with 3, 4, 5 and 10 maximum neighbors.

Number of Terminals	42	140	700	1400
Message evaluaded per Terminal	delivered QueryHits	Deviation	QueryHits	Queries

The following graphs visualizes the mean value and standard deviation of sucessfully delivered QueryHits messages per servent and are the results of several simulations. The total simulation time is 2000 seconds, with 200 different searchKeys and with LifetimeChurn turned on. The x-axis depicts the number of different servents in the network, refered as terminals and the y-axis depicts the mean value and standard deviation of successfully delivered QueryHits messages per servent.

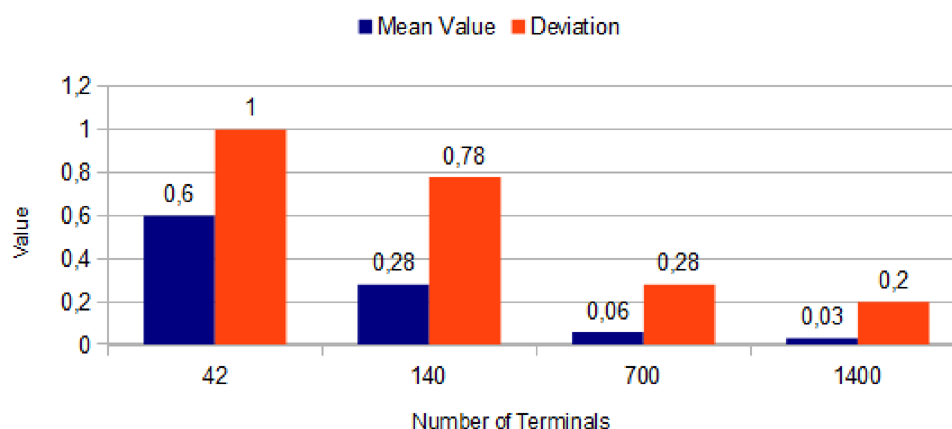

Figure 5.1.: Delivered QueryHits with max 3 neighbors

Note that for every simulation a mean value of 26,31 new queries were started per terminal and that every search key is equally distributed over all terminals. This means that there are no common search keys shared by many of the terminals. As can be seen from Figure 5.1, the mean value of sucessful delivered QueryHits decreases with an increase of the number of terminals with a constant amount of neighbors. The mean value of delivered QueryHits is 2,07 per terminal in a network consisting of 42 terminals with an standard deviation of 1. If we set the amount of terminals to 140 the mean value halves and amounts to 0,28. With an constant amount of new queries started on each servent, this means that queries get less successful with an increase of participating servents. With 1400 participating terminals the mean value drops down to 0,03 successful queries per terminal. This leads to the conclusion that the efficiency of a Gnutella network drops with increasing numbers of terminals

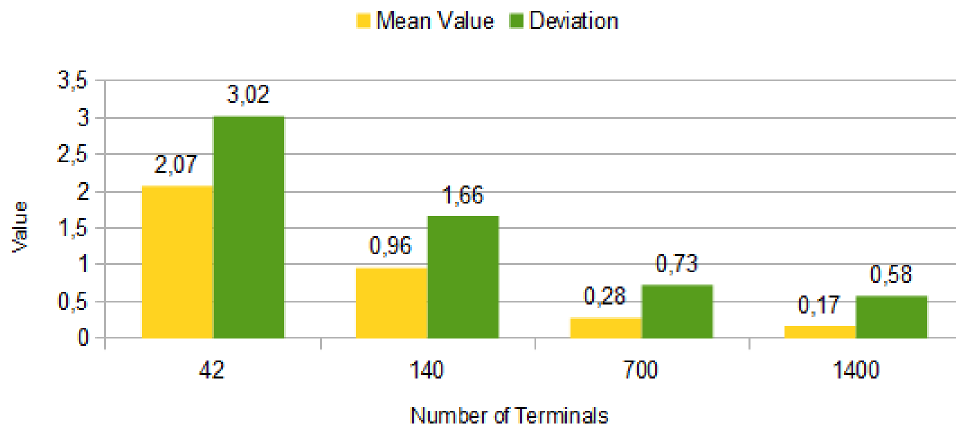

Figure 5.2.: Delivered QueryHits with max 4 neighbors

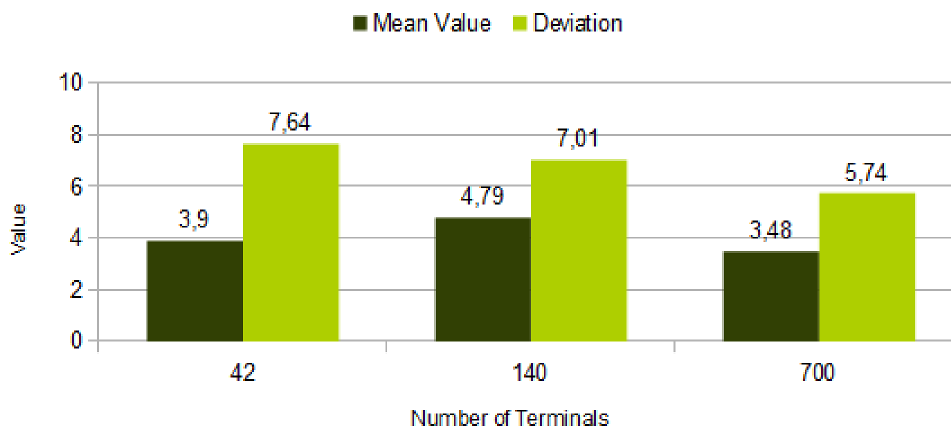

Figure 5.3.: Delivered QueryHits with max 10 neighbors

and a constant number of neighbors. Figure 5.2 depicts the same simulation while allowing each servent to have a maximum amount of four neighbors at any given time. The mean value of QueryHits in a network with 42 terminals is 2.07, and with 140 terminals it is 0.96. We can see that the amount hast more than tripled compared to the results with three neighbors. The value has changed for 700 and 1400 terminals as well, but it is still too small. A mean value of 0.28 with 700 terminals and an average of 26.31 new queries per terminal means that only 1.06% of all queries were successful. With 42 terminals the percentage of successful QueryHits is 7.86%. If we further increase the amount of maximum neighbors and set it to 10 we see Figure 5.2 that the mean value of delivered QueryHits for 42 terminals is 3.9. We have doubled the amount of maximum neighbors and the number of QueryHits doubled as well. For 140 terminals we get a value of 4,79 and with 700 terminals we get a value of 3.48. This is 12 times the value from before. This means that 13,38% of all queries were successful. Hence, an increase of neighbors leads to an increase in efficiency. In order to verify this statement we take a look at how many query messages each terminal forwarded in the same simulation on average.

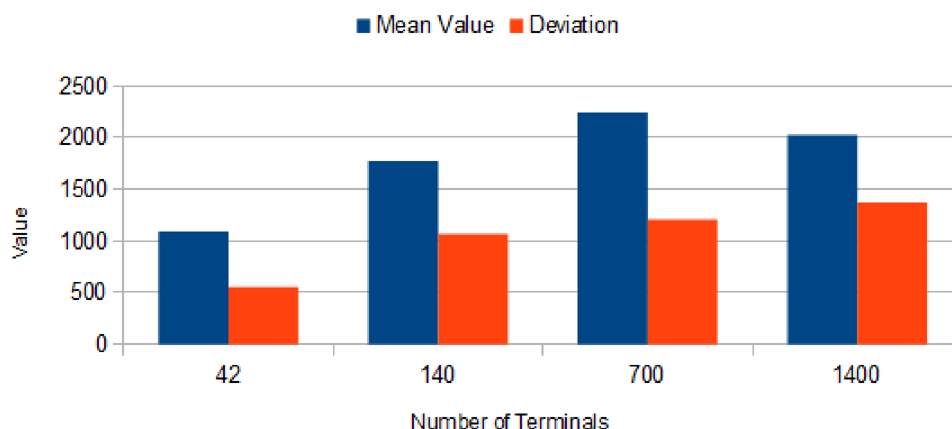

Figure 5.4.: processed Query Messages with max 3 neighbors

A network with 42 terminals and a maximum amount of three neighbors forwarded on average 1093,5 query messages and a network with 700 terminals and the same amount of neighbors forwarded 2246,19 query messages. This is double the amount of processes query messages. There were more than 20 times as many terminals involved compared to 42 terminals. This is a huge increase in network traffic while the amount of successful QueryHits drops [Figure 5.1]. This is an indicator of Gnutellas scaling issues. Figure 5.5 visualises the results of the same simulation but with a maximum amount of 10 neighbors per terminal. We saw in Figure 5.3 that an increase in neighbors also increases the number of successful QueryHits, but in order to achieve this each terminal forwarded on average 119036,44 query messages. Compared to 2246,19 query messages that is 53 times the value than before. While the amount of neighbors was increased by 7 the amount of queryMessages per

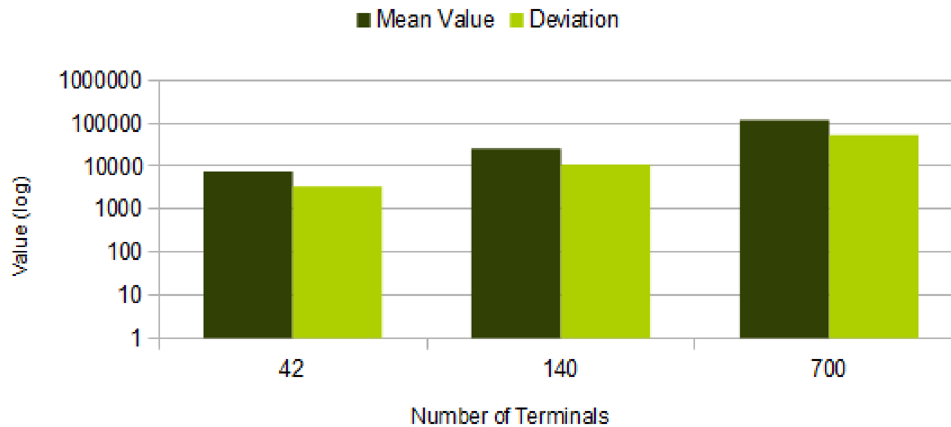

Figure 5.5.: processed Query Messages with max 10 neighbors

terminal grew exponentially. This is another indicator of Gnutellas scaling issues.

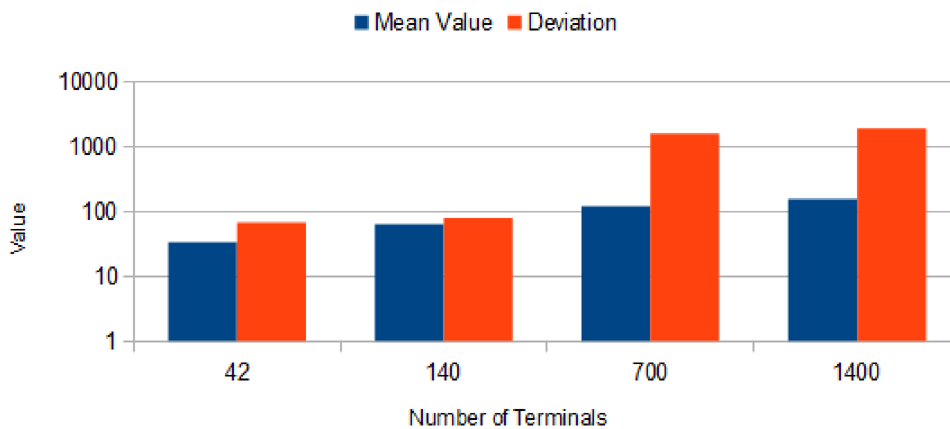

Figure 5.6.: processed QueryHit Messages with max 3 neighbors

These two graphs depicture the mean value of forwarded QueryHit messages per servent. A logarithmic scale was used since the deviation is far off from the mean value. As you can see the mean value rises from 33.52 with 42 terminals to 121,84 with 700 terminals. This value does not grow exponentially. If you would keep the number of neighbors constant but increase the number of terminals the average value of processed queryHit messages would slowly decrease. With a growing network it gets less likely to get a queryHit since the number of terminals you can reach via Ping and Ping messages is limited by the TTL in the query message. This is true for the Gnutella protocol since queryHit messages do net get duplicated and forwarded to every neighbor but they take the very same path back to the servent which started the corresponding query.

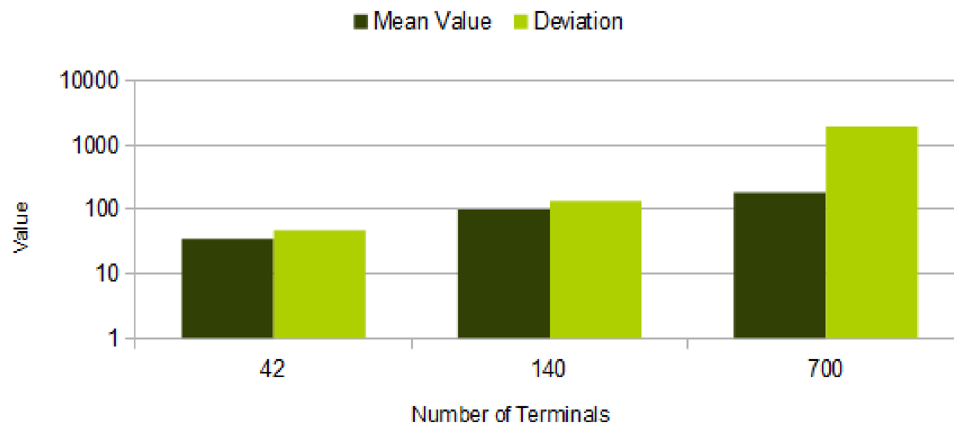

Figure 5.7.: processed QueryHit Messages with max 10 neighbors

Evaluation

Gnutella scales very badly [4]. The data presented in the simulation results leads to the same conclusion. We can see that an increase of servents or neighbors leads to a polynomial increase of network traffic, due to the many queryMessages flooding the network. In a real world example the protocol does scale even worse since many factors were not taken into account in this Gnutella simulation, like the limitation of each servent to process packages, or the need to use TCP's flow control in order to guarantee a reliable delivery of the message. This simulation assumed the best case scenario in terms of network infrastructure. This means that there are no limitations on single connection capacities and every servent is able to process the same amount of packages in the same time without dropping one or the need to retransmitt lost packages. These are all aspects which are part of the real-world application and slow down the Gnutella network even further. Additionaly the real-world example uses Ping and Pong messages in order to actively probe the network for additional neighbors. This also increases the network traffic by a small amount [4] All in all, this Gnutella implementation mimics the real-life application in many ways due to a bad scaling according to the numer of query messages flooding the network and the decreasing number of successful queryHit messages.

Chapter 6.

Conclusion

Summary

The purpose of this thesis is to implement a new Gnutella module for the network simulator OMNeT++ and Oversim. The Gnutella module mimics the official RFC in many ways, but focused on Gnutellas scalability. Few adjustments were done in order to create consistent , strong connective networks which then were evaluated. It is not possible to simulate the distribution of data of a real Gnutella network with this module, but recreate the same problems with the current implementation. Like Gnutella, this module does scale badly with increasing number of servents and this leads to a massive amount of network traffic and sent Query Messages. It is important that all servents in the simulation participate equal. This means that every servent has as many search keys like any other servent and issues the same amount of new Querys. This is the original intention behind Gnutella but unfortunaly does not work this way in a real-world example where 50% of all data is served by just 1% of servents [4]. The way each servent joins the Gnutella network was changed in order to work around the creation of independent sets of cliques in the simulation. This is a simulation specific problem because, unlike the real-world example, the network gets created and every servent joins simultaniously. Utilizing the bootstraplist leads to the creation of many independent sets of networks instead of one bigger network. In a real world example a network is already established and every servent gets the same bootstrap list, and thus connects to the same network. It has been shown in this thesis that even with 700 servents the network begins to malfunction and loses its efficiency exponentially fast, leading to massive amounts of Query messages occuping bandwidth. Using TCP instead of UDP makes this problem more severe since many of those Query Messages would get dropped and re-send in the process leading to even more network traffic.

Outlook

There are many ways to adjust this module in the future. A new idea can be added to this implementation adding a functional method to use Ping forwarding in order to actively search for new potentional neighbors and getting rid of the *sharedList* approach which was used in this implementation. Another way to adjust this module is to remove the equality of all servents and adding a method which forces a few servents to be more active and others and to be willing to share more data than others. This would make this module a better approach to the real-world example in the way, mentioned above, where 50% off all data is served by just 1% of the servents.

Appendix A.

Installing OMNeT++ and Oversim

General Information

This chapter describes how to install OMNeT++ on Linux, even though more platforms are supported.

Supported Platforms:

OMNeT++ is at least supported on the following operating systems:

- Windows 7,8 and XP
- Max OS X 10.7, 10.8 and 10.9
- Linux distributions:
  Ubuntu 12.04 LTS, 13.04
  Fedora Core 18
  Red Hat Enterprise Linux Desktop Workstation 6.4
  OpenSUSE 12.3

The Simulation IDE can be used on the following platforms:

- Linux x86 32/64-bit
- Windows 7,8 and XP
- Mac OS X 10.7, 10.8 and 10.9

Simulations can be run practically on any unix-like enviroment with a decent and up-to-date C++ compiler. IDE platforms are restricted because the IDE relies on a native shared library, which was compiled for the above platforms and distrubuted in binary form for convenience.

Installing Guide for Linux

If your Linux distrubution is not listed above, you still may be able to use some distrospecific instructions in this Guide.

Some Ubuntu derivatives (Ubuntu instructions may apply);

- Kubuntu, Xubuntu, Edubuntu, ...
- Linux Mint
- Some Debian-based distributions (Because Ubuntu itsef is based on Debian)
- Knoppix and derivatives
- Mepis

Some Fedora-based distrubutions (Fedora instructions max apply):

- Simplis
- Eeedora

Installing the Prerequisite Packages:

OMNeT++ requires several packages to be installed on the computer. These packages include the C++ compiler (gcc), the JRE and other libraries and programs. These packages can be installed from the software repositories of your Linux distribution. You may need superuser permission to install packages. Note that not all packages are available from software repositories; some (optional) ones need to be downloaded separately from their web sites, and installed manually.

Downloading and Unpacking

Download OMNeT++ from `http://omnetpp.org`. Make sure to download the generic achive, `omnetpp-4.4.1-src.tgz`

Copy the archive to the directory where you want to install it. This is usually your home directory, `/home/<yourname>`. Open a terminal and extract the archive using the following command:

```
$ tar xvfz omnetpp-4.4.1-src.tgz
```

This will create an omnetpp-4.4.1 subdirectory with the OMNeT++ files in it.

Enviroment Variables

OMNeT++ needs its `bin/` directory to be in the path. To add `bin/` to `PATH` temporarily (in the current shell only), change into the OMNeT++ directory and execute the `setenv` script:

```
$ cd omnetpp-4.4.1
$ . setenv
```

the script also adds the `lib/` subdirectory to `LD_LIBRARY_PATH`, which may be necessary on systems that do not support the rpath mechanism

In order to set the enviroment variables permanently, edit `.bashrc` in your home directory. Use your favourite text editor to edit `.bashrc`, for example gedit:

```
$ gedit ~/.bashrc
```

(you do net need to change to your home directory for this command to work)

Add the following line at the end of the file, then save it:

```
export PATH=$PATH:$HOME/omnetpp-4.4.1/bin
```

(if you have extracted the files in a different path, you have to change the line accordingly)

You need to close and re-open the terminal for the changes to take effect.

(Please note, if you use a shell other than bash, consult the man page of that shell to find out which startup file to edit, and how to set and export variables. Note that all Linux distributions covered in this Installation Guide use bash unless the user has explicitly selected another shell.)

```
crunchbang@crunchbang: ~/sim/omnetpp-4.2.2
checking for LibXML XML parser with CFLAGS=" -O2 -DNDEBUG=1   -D_LARGEFILE_SOURCE -D_FILE_O
FFSET_BITS=64 -fno-stack-protector -DHAVE_SWAPCONTEXT " LIBS="-lxml2"... no
checking for LibXML XML parser with CFLAGS=" -O2 -DNDEBUG=1   -D_LARGEFILE_SOURCE -D_FILE_O
FFSET_BITS=64 -fno-stack-protector -DHAVE_SWAPCONTEXT -I/usr/include/libxml2" LIBS="-lxml2"
... no
checking for Expat XML parser with CFLAGS=" -O2 -DNDEBUG=1   -D_LARGEFILE_SOURCE -D_FILE_OF
FSET_BITS=64 -fno-stack-protector -DHAVE_SWAPCONTEXT " LIBS="-lexpat"... no
configure: WARNING: No suitable XML parser found: no support for XML input
checking for Akaroa with CFLAGS=" -O2 -DNDEBUG=1   -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=
64 -fno-stack-protector -DHAVE_SWAPCONTEXT -DXMLPARSER=none -I/usr/local/akaroa/include" LI
BS="-L/usr/local/akaroa/lib -lakaroa -lfl"... no
configure: WARNING: Optional package Akaroa not found
configure: creating ./config.status
config.status: creating Makefile.inc
config.status: creating test/core/runtest

WARNING: The configuration script could not detect the following packages:

    MPI (optional)  PCAP (optional)  LibXML/Expat  Akaroa (optional)

Scroll up to see the warning messages (use shift+PgUp), and search config.log
for more details. While you can use OMNeT++ in the current configuration,
be aware that some functionality may be unavailable or incomplete.

Your PATH contains /home/crunchbang/sim/omnetpp-4.2.2/bin. Good!

TCL LIBRARY is set. Good!
crunchbang@crunchbang:~/sim/omnetpp-4.2.2$
```

Figure A.1.: Terminal after ./configure command

Configuring and Building OMNeT++

In the top-level OMNeT++ directory, type:

```
$ ./configure
```

The configure script detects installed software and configuration on your system. It writes the results into the Makefile.inc file, which will be read by the makefiles during the build process.

Normally, the configure script has to be running under the graphical enviroment (X11) in order to test for wish , the TcL/Tk shell. IF you are logged in via an ssh session, or there is some other reason why X is not running, the easies way to work around the problem is to tell OMNeT++ to build without Tcl/Tk. To do that, use the command

```
$ NO_TCL=1 ./configure
```

instead of plain `./configure`

If there is an error during `./configure`, the output may give hints about what went wrong. Scroll up to see the messages. (Use Shift+PgUp; you may need to increase the scrollback buffer size of the terminal and re-run ./configure .) The script also writes a very detailed log of its operation into config.log to help track down errors. Since config.log is very long, it is recommended that you open it in an editor, like gedit, and search for phrases like error or the name of the package associated with the problem

```
crunchbang@crunchbang: ~/sim/omnetpp-4.2.2
/gcc-debug//txcl2.o out/gcc-debug//txcl3.o out/gcc-debug//txcl4.o out/gcc-debug//txcl5.o ou
t/gcc-debug//txcl6.o out/gcc-debug//txc2.o out/gcc-debug//txc3.o out/gcc-debug//txc4.o out/
gcc-debug//txc5.o out/gcc-debug//txc6.o out/gcc-debug//txc7.o out/gcc-debug//txc8.o out/gcc
-debug//txc9.o out/gcc-debug//tictocl3_m.o out/gcc-debug//tictocl4_m.o out/gcc-debug//ticto
cl5_m.o out/gcc-debug//tictocl6_m.o  -Wl,--no-as-needed -Wl,--whole-archive  -Wl,--no-whole
-archive -L"/home/crunchbang/sim/omnetpp-4.2.2/lib/gcc" -L"/home/crunchbang/sim/omnetpp-4.2
.2/lib"-loppmaind -u _tkenv_lib -Wl,--no-as-needed -lopptkenvd -loppenvird -lopplayoutd -u
cmdenv_lib -Wl,--no-as-needed -loppcmdenvd -loppenvird  -loppsimd -ldl -lstdc++
ln -sf out/gcc-debug//tictoc .
make[2]: Leaving directory `/home/crunchbang/sim/omnetpp-4.2.2/samples/tictoc'
===== Compiling sockets ====
cd /home/crunchbang/sim/omnetpp-4.2.2/samples/sockets && make
make[2]: Entering directory `/home/crunchbang/sim/omnetpp-4.2.2/samples/sockets'
g++ -Wl,--export-dynamic  -Wl,-rpath,/home/crunchbang/sim/omnetpp-4.2.2/lib -Wl,-rpath,. -o
out/gcc-debug//sockets  out/gcc-debug//Cloud.o out/gcc-debug//ExtHttpClient.o out/gcc-debug
//ExtTelnetClient.o out/gcc-debug//HttpClient.o out/gcc-debug//HttpServer.o out/gcc-debug//
QueueBase.o out/gcc-debug//SocketRTScheduler.o out/gcc-debug//TelnetClient.o out/gcc-debug/
/TelnetServer.o out/gcc-debug//HttpMsg_m.o out/gcc-debug//NetPkt_m.o out/gcc-debug//TelnetP
kt m.o  -Wl,--no-as-needed -Wl,--whole-archive  -Wl,--no-whole-archive -L"/home/crunchbang/
sim/omnetpp-4.2.2/lib/gcc" -L"/home/crunchbang/sim/omnetpp-4.2.2/lib" -loppmaind -u  tkenv
lib -Wl,--no-as-needed -lopptkenvd -loppenvird -lopplayoutd -u  cmdenv lib -Wl,--no-as-need
ed -loppcmdenvd -loppenvird  -loppsimd -ldl -lstdc++
ln -sf out/gcc-debug//sockets
make[2]: Leaving directory `/home/crunchbang/sim/omnetpp-4.2.2/samples/sockets'
make[1]: Leaving directory `/home/crunchbang/sim/omnetpp-4.2.2'

Now you can type "omnetpp" to start the IDE
crunchbang@crunchbang:~/sim/omnetpp-4.2.2$
```

Figure A.2.: Terminal after make command

When ./configure has finished, you can compile OMNeT++. Type in the terminal:

```
$ make
```

(Note that this process may take up to a few minutes)

To take advantage of multiple processor cores, add the -j2 option to the `make` command line.

Verifying the Installation

You can now verify that the sample simulations run correctly. For example, the dyna simulation is started by entering the following commands:

```
$ cd samples/dyna
$ ./dyna
```

By default, the samples will run using the Tcl/Tk enviroment. You should see nice gui windows and dialogs

Starting the IDE

You can launch the OMNeT++ Simulation IDE by typing the following command in the terminal:

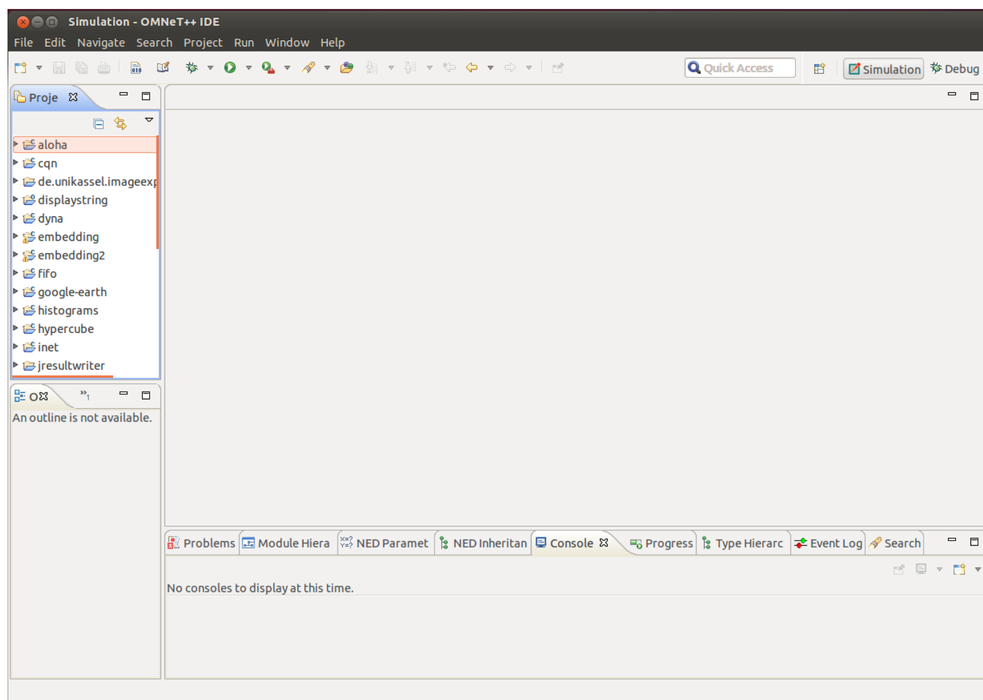

Figure A.3.: Example of a freshly opened OMNeT++ Session

```
$ omnetpp
```

If you would like to be able to access the IDE from the application launcher or via a desktop shortcut , run one or both of the commands below:

```
$ make install-menu-item
$ make install-desktop-icon
```

Or add a shortcut that points to the omnetpp programm in the IDE subdirectory by other means, for example using the Linux desktop's context menu.

Reconfiguring the Libraries

If you need to recompile the OMNeT++ components with different flags (e.g different optimazisation), then change to the top-level OMNeT++ directory, edit `configure.user` accordingly, then type:

```
$ ./configure
$ make cleanall
```

```
$ make
```

If you want to recompile just a single library, then change to the directory of the library (e.g cd sry/sim) and type:

```
$ make clean
$ make
```

Installation of the Gnutella module

The source files for the Gnutella implementation for OMNeT++ and OverSim consists of the `gnutella` folder, the `gnutellasearchapp` folder, `omnetpp.ini, default.ini, ExtAPIMessagesGnutella_m.h`, `ExtAPIMessagesGnutella_m.cc` and `ExtAPIMessagesGnutella.msg`

Put the `gnutella` folder into `\OverSim\src\overlay`.
Put the `gnutellasearchapp` folder into `\OverSim\src\applications`.
Delete the `gia` folder in the `\OverSim\src\overlay` directory.
Delete the `giasearchapp` folder in the `\OverSim\src\applications` directory.
Put the `omnetpp.ini` and `default.ini` files into `\OverSim\simulations`.
Put the `ExtAPIMessagesGnutella` files into `\OverSim\src\common`.
You can now select and start the gnutella configuration directly from the simulator using the `omnetpp.ini` file.

If this does not work start the terminal and switch your directory into the gnutella folder and type:
`opp_msgc GnutellaMessages.msg`
Then switch into the `common` folder where the ExtAPIMessagesGnutella files are located and type:
`opp_msgc ExtAPIMessagesGnutella.msg`

This should recompile your message files and it is required in order to run the simulaton on other versions than OMNeT++ 4.4.1.

It is also possible to run the simulation without having to use the graphical user inferface.
Start your terminal and switch into the `/oversim` subfolder and type:

```
$ make makefiles
$ make
```

This compiles every file in your oversim subfolder including your module.

You can now start your simulation with and without gui.

```
$ cd oversim/simulations
$ ./src/oversim -uCmdenv -cMyConfig
$ ./src/oversim -uTkenv -cMyCondig
```

Appendix B.

Simulation Result Tables

Sucessfully delivered QueryHits

	Number of Neighbors							
	3		4		5		10	
Number of Terminals	Mean Value	Deviation	Mean Value	Deviation	Mean Value	Deviation	Mean Value	Deviation
42	0,6	1	2,07	3,02	2,4	4,18	3,9	7,64
140	0,28	0,78	0,96	1,66	1,59	2,89	4,79	7,01
700	0,06	0,28	0,28	0,73	0,84	1,69	3,48	5,74
1400	0,03	0,2	0,17	0,58	0,49	1,18	exceeded computation time	

Forwarded Query Messages

	Number of Neighbors							
	3		4		5		10	
Number of Terminals	Mean Value	Deviation	Mean Value	Deviation	Mean Value	Deviation	Mean Value	Deviation
42	1093,5	556,52	2451,7	1156,91	3387,72	1580,61	7599,62	3408,72
140	1773,83	1067,89	7244,68	3535,77	11116,08	4990,31	25862,19	11047,86
700	2246,19	1,21E+3	14683,86	7,99E+3	39589,01	18890,29	119036,44	53050,17
1400	2024,2	1,37E+3	14771,38	8,93E+3	51029,36	23497,42	exceeded computation time	

Forwarded QueryHits Messages

	Number of Neighbors							
	3		4		5		10	
Number of Terminals	Mean Value	Deviation	Mean Value	Deviation	Mean Value	Deviation	Mean Value	Deviation
42	33,52	67,6	33,54	52,33	40,16	59,86	35,21	47,92
140	63,4	79,45	277,78	2527,66	270,81	2528,43	100,61	137,08
700	121,84	1616,07	280,43	2780,44	286,3	2771,01	186,82	1974,67
1400	156,4	1968,27	239,12	2544,6	125,58	1616,26	exceeded computation time	

Bibliography

[1] “System/application designs, optimizations and implementations on overlay networks,” http://web.cse.ohio-state.edu/hpcs/WWW/HTML/internet-p2p.html.

[2] “Free software foundation, regarding gnutella,” http://www.gnu.org/philosophy/gnutella.en.html.

[3] “Gnutella 0.4 specification,” http://rfc-gnutella.sourceforge.net/developer/stable/index.html.

[4] G. Tribhuvan, “A brief introduction and analysis of the gnutella protocol,” http://archive.cone.informatik.uni-freiburg.de/teaching/seminar/p2p-networks-w06/submissions/gnutella.pdf.

[5] “Omnet credits,” http://omnetpp.org/index.php/documentation/3636.

[6] “Omnet manual,” http://www.omnetpp.org/doc/omnetpp/Manual.pdf.

[7] G. Gross Wehrle, “Modeling and tools für netzwerk simulation,” Springer , 2010.

[8] I. Baumgart, B. Heep, and S. Krause, “OverSim: A flexible overlay network simulation framework,” in *Proceedings of 10th IEEE Global Internet Symposium (GI ’07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pp. 79–84, May 2007.

[9] “Oversim wiki,” http://www.oversim.org/wiki/OverSimChurn.

[10] Z. Yao, “Modeling heterogeneous user churn and local resilience of unstructured p2p networks,” in *Proceedings of the 2006 14th IEEE International Conference on Network Protocols*, pp. 32–41, 2006.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 03.September 2014 Stefan Schmid

Please add here

the DVD holding sheet

This DVD contains:

- A *pdf* Version of this bachelor thesis
- All LaTeXand grafic files that have been used, as well as the corresponding scripts
- **[adapt]** The source code of the software that was created during the bachelor thesis
- **[adapt]** The measurment data that was created during the evaluation
- The referenced websites and papers