UNIVERSITY OF OSLO
Department of Informatics

Introducing name resolution into OLSR

Master thesis

Øyvind Spigseth

11th January 2008

Abstract

Mobile Ad-hoc Networks (MANETs) are wireless networks consisting of autonomous nodes that exist in an infrastructure-less environment. There are no centralized servers and no hierarchy of nodes. The *Domain Name System* (DNS) is important to million of nodes that are located in the Internet. Through a huge amount of DNS servers spread around the world, they serve a name resolution service to all nodes existing in Internet. Users of the Internet have the advantage of using hostnames or fully qualified domain names to communicate with another host. Because of the fast growth of MANETs, where they tend to be large and dense, a name resolution service is also desired for these networks. Since MANETs are characterized by unstable topology the existing DNS can not work in MANETs. *Optimized Link State Routing* (OLSR) is a routing algorithm designed for MANETs that propagate IP addresses through periodically emitted control messages. These control messages are broadcasted through the *Multipoint Relay* (MPR) flooding scheme, which reduces the total amount of overhead. This thesis investigates the possibilities of designing and implementing a distributed name resolution service for MANETs based on the existing *Optimized Link State Routing* (OLSR) algorithm by extending the existing control messages and data sets. The efficiency and generated overhead are measured to compare this solution together with existing solutions.

Preface

This thesis is a part of my Master Degree at the University of Oslo, Department of Informatics. The work was carried out for the Distributed Multimedia Systems (DMMS) group, during the autumn semester of 2007. My teaching supervisors were Professor Thomas Plagemann, the leader of the DMMS group, and Matija Pužar. I would like to thank Plagemann for assisting me giving me feedback and structuring this document and Pužar who has helped me a lot with technical problems and giving me other useful hints during this semester.

I would also like to thank my family which has supported me through a semester with great challenges. Without them I could not have reached this far. Ulrich Schumacher also deserves an acknowledgement for reading through this report and giving valuable feedback.

Øyvind Spigseth
January 2008

Contents

1 Introduction ... 1
1.1 Problem description and motivation ... 1
1.1.1 Mobile Ad-hoc networks ... 1
1.1.2 Domain Name System ... 2
1.1.3 DNS in Mobile Ad-hoc Networks ... 3
1.1.4 Claims ... 3
1.2 Terminology ... 4
1.3 Methodology ... 4
1.4 Organization of the report ... 5

2 Background ... 7
2.1 Related work ... 7
2.1.1 Name resolution system based on a reactive routing protocol ... 7
2.1.2 Partly distributed name resolution system with clusters ... 10
2.1.3 MIDAS – Naming by extending OLSR ... 11
2.1.4 OLSRd plug-in ... 12
2.2 Existing solutions versus our claims ... 12
2.3 Optimized Link State Routing – OLSR ... 13
2.3.1 OLSR terminology ... 14
2.3.2 OLSR Information Repositories ... 14
2.3.3 OLSR Packet Format ... 15
2.3.4 OLSR Message Types ... 17
2.3.4.1 Multiple Interface Declaration message format ... 17
2.3.4.2 HELLO message format ... 17
2.3.4.3 Topology Control message format ... 19
2.3.5 OLSR functionality ... 19
2.3.5.1 Multiple interfaces ... 19
2.3.5.2 Link sensing ... 20
2.3.5.3 Neighbour detection ... 20
2.3.5.4 Two-hop neighbour detection ... 21
2.3.5.5 Multipoint relaying ... 22
2.3.5.6 MPR Selector detection ... 23
2.3.5.7 Distribution of Topology Control messages ... 24
2.3.5.8 Forwarding of OLSR control messages ... 25
2.3.5.9 Route calculation ... 25
2.3.6 OLSR Summary ... 26
2.4 Background summary ... 26

3 Design ... 27
3.1 Design claims and assumptions ... 27
3.2 Extending the control messages ... 28
3.2.1 TC messages ... 28
3.2.2 HELLO messages ... 30
3.2.3 TC-messages and HELLO-message working together ... 31
3.3 Extending data structures ... 32
3.4 Generating and parsing HELLO messages and TC messages ... 33
3.4.1 Naming ... 33
3.4.2 Generation and parsing of HELLO messages ... 34

3.4.3 Generation and parsing of TC messages ... 35
3.5 Name resolution ... 35
3.6 Summary ... 37

4 Implementation ... 39
4.1 OLSRd dataflow ... 39
4.1.1 Dataflow of generating HELLO messages ... 39
4.1.2 Dataflow of parsing HELLO messages ... 41
4.1.3 Dataflow of generating TC messages ... 42
4.1.4 Dataflow of parsing TC messages ... 43
4.1.5 Summary code analysis ... 44
4.2 Implementation of the extensions ... 45
4.2.1 Implementing the name table ... 45
4.2.2 Extending HELLO messages with names ... 47
4.2.3 Parsing the new HELLO message ... 50
4.2.4 Extending TC messages with names ... 52
4.2.5 Parsing the new TC messages ... 56
4.2.6 Other changes in OLSRd ... 59
4.3 Testing the implementation ... 59
4.3.1 Test environments ... 60
4.3.1.1 Real nodes ... 60
4.3.1.2 NEMAN ... 60
4.3.2 Test scenario 1 ... 63
4.3.2.1 Results with `TcRedundancy` equal to zero ... 64
4.3.2.2 Results with `TcRedundancy` equal to two ... 66
4.3.2.3 Other results of the first test scenario ... 68
4.3.3 Test scenario 2 ... 69
4.3.4 Test scenario 3 ... 70
4.3.5 Test scenario 4 ... 72
4.3.6 Conclusion – testing functionality ... 73
4.4 Summary ... 73

5 Evaluation ... 75
5.1 Test methods ... 75
5.2 Measured overhead in a 500x400 area ... 78
5.3 Measured overhead in a 1000x800 area ... 81
5.4 Measured overhead when hostname in OLSRd control message increases ... 84
5.5 Summary ... 84

6 Conclusion ... 87
6.1 Summary of the report ... 87
6.2 Claims versus our solution ... 87
6.3 Further work ... 89

Appendix A ... 91

Bibliography ... 93

List of Figures

Figure 1.1: A small part of the domain namespace. 2

Figure 2.1: Extension of a reactive protocol's route requests and route replies. 8

Figure 2.2: A fully distributed name resolution system. 9

Figure 2.3: A partially distributed name service. 10

Figure 2.4: A MANET partitioned into clusters. 11

Figure 2.5: The OLSR information repositories. 14

Figure 2.6: The basic layout of any packet in OLSR. 16

Figure 2.7: The OLSR MID message format. 17

Figure 2.8: The format of the OLSR HELLO message. 18

Figure 2.9: The 8-bit Link Code field in the OLSR HELLO message. 18

Figure 2.10: The format of the OLSR Topology Control message. 19

Figure 2.11: The process of creating a symmetric link between two neighbours. 21

Figure 2.12: The difference between classical flooding and MPR flooding. 23

Figure 3.1: A MPR node with 4 MPR selectors. 29

Figure 3.2: The new design of the OLSR TC-message. 30

Figure 3.3: The name field used in the control messages. 30

Figure 3.4: The new HELLO message contains just the name of the originator of the HELLO message. The name is stored in a name field as explained in Figure 3.3. 31

Figure 3.5: Name distribution with HELLO messages and TC messages working together... 32

Figure 3.6: A shared data structure to store names from HELLO messages and TC messages. 33

Figure 3.7: A local name resolution with a DNS server running on a local node. 36

Figure 3.8: A local name resolution with the *hosts* file on a local node. 36

Figure 4.1: Dataflow for HELLO message generation. If an interface is going to be initialized with OLSR, a scheduler event is registered on that new interface. This event is for HELLO message generation, and is triggered on a fixed time interval. 39

Figure 4.2: Dataflow when parsing a HELLO message. 41

Figure 4.3: Dataflow of generating TC messages. 42

Figure 4.4: Dataflow of parsing TC messages. 43

Figure 4.5: The control buttons in the graphical user interface of NEMAN. 60

Figure 4.6: The graphical user interface after the scenario file is loaded, and the "Prepare" has been pushed. At the top we can see the control buttons. Below them we can see the geographical area where the nodes are simulated to be. 61

Figure 4.7: The topology of the first test scenario. 63

Figure 4.8: Topology used in the second test scenario. The topology shown here is a snapshot from the NEMAN GUI. 69

Figure 4.9: Topology used in the third test scenario. The topology shown here is a snapshot from the NEMAN GUI. 71

Figure 4.10: Topology used in the fourth test scenario. The topology shown here is a snapshot from the NEMAN GUI. 72

Figure 5.1: The experimental design with all input parameters and their values.......... 76

Figure 5.2: This diagram shows us how many packets that were captured during the experiment when nodes were stored in an area of 500x400. 79

Figure 5.3: The measured overhead where nodes exist in a 500x400 area. 80

Figure 5.4: This diagram shows us how many packets that were captured during the experiment when nodes were stored in a area of 1000x800. 82

Figure 5.5: The measured overhead where nodes exist in a 1000x800 area. 83

Figure 5.6: This diagram shows how the total amount of overhead increases with number of characters in the hostname. 84

List of Tables

Table 2.1: This table shows how existing solutions meet our requirements. 13

Table 3.1: A comparison of overhead with different solutions to include the name into the TC message. 29

Table 4.1: Functions that need modifications and/or extensions regarding HELLO messages. 44

Table 4.2: Data structures that need modifications and/or extensions regarding HELLO messages. 44

Table 4.3: Functions that need modifications and/or extensions regarding TC messages. 45

Table 4.4: Data structures that need modifications and/or extensions regarding TC messages. 45

Table 4.5: The new name table is an extended data structure of OLSR. 46

Table 4.6: Functions that is needed for the nametable. 46

Table 4.7: Configuration of the nodes in the first test scenario. 64

1 Introduction

Rapid growth of *Mobile Ad-hoc networks* – MANETs – makes it hard not to have a name resolution service. The number of nodes that participate in a Mobile Ad-hoc network can be high. It can then be difficult to remember all the network addresses to the corresponding nodes.

In the Internet we have the *Domain Name System* – DNS – that is able to serve all hosts a name resolution service. Problems are met when we try to use DNS in MANETs. This is because of the special characteristics that these networks have.

The challenge of implementing a name resolution in MANETs is that we need to create a whole new design in order to make it work properly.

1.1 Problem description and motivation

In this section we first look at the characteristics of *Mobile Ad-hoc networks*. Then we present the *Domain Name System* and discuss why it does not function in these networks.

1.1.1 Mobile Ad-hoc networks

MANETs are wireless networks where nodes use the `IEEE 802.11` set of standards for communication. These networks are formed by an arbitrary number of autonomous nodes. When a node comes in range of another node, they create a link and are ready to communicate. If we compare MANETs with other wired networks, we got some important differences that characterize a MANET:

- *Autonomous nodes* – All nodes are independent of other nodes. They work on their own, and are not dependent on others to be able to participate in the MANET.
- *No infrastructure* – MANETs have no infrastructure that organizes the nodes as in the Internet.
- *Multi-hop environment* – All nodes acts as a router. The structure of a MANET is therefore flat. A path between the source and the destination node can consist of an arbitrary number of hops of intermediate nodes.
- *Unstable topology* – MANETs are designed for mobile nodes that can move around within a geographical area with an arbitrary speed.
- *Power constraints* – Mobile nodes use battery power and have thereby limitations of their power sources.
- *CPU constraints* – In some MANETs, the nodes are small devices that do not have the same CPU speeds as normal computers.
- *Bandwidth constraints* – In a given geographical area there may be obstacles between nodes that limit the bandwidth on links between them. There can also be other environmental factors like signal interference and atmospheric disturbance.

When designing software for MANETs, these characteristics must be taken into consideration. This often makes it harder to develop new software for MANETs than for applications that are intended to work on other networks.

1.1.2 Domain Name System

The *Domain Name System* (DNS), which is the existing name resolution system, is widely used on Internet today. DNS is built on a strong hierarchy that is composed by a tree-structured name space called the *Domain Name Space*. RFC 1034 [1] illustrates a small part of the *Domain Name Space* shown in Figure 1.1. This tree is organized such that each leaf has one or more resource records. These resource records hold information associated with the domain name.

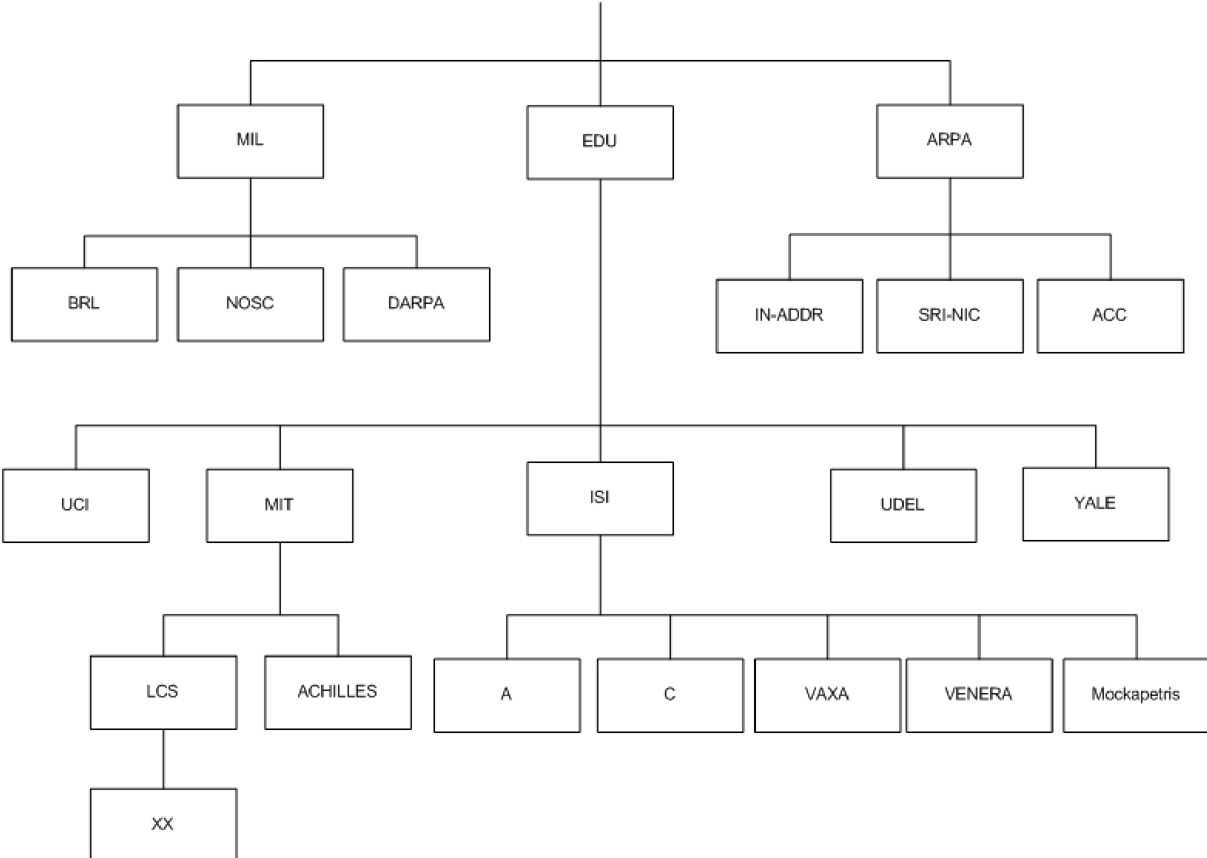

Figure 1.1: A small part of the domain namespace.

Name servers hold the information about their domain tree structure. A *name server* has in general complete information about a subset of the domain space.

Nodes that want information from a DNS server have to know how to communicate with it. They send DNS requests to the name servers and receive and process DNS responses. A node which is able to communicate with a name server in this manner is called a *resolver*. Resolving usually entails iterating through several name servers to find the needed information.

Name construction in DNS reflects the structure of the domain namespace. A name consists of two parts: The hostname and the domain suffix. The user configures the hostname as well

as the name server in the Internet. This very often leads to an organizational structure. For instance, all nodes that are related to the University of Oslo use the `uio` domain. This domain is part of the `no` sub domain space. The domain suffix consists of all domains that lead towards the root of the domain space. Hence we get the `uio.no` domain suffix for all nodes that are related to the University of Oslo.

Another aspect of DNS is that domains are often related to a block of network addresses. The most common network address that is in use today is Internet Protocol (IP) addresses. An organization reserves a set of IP addresses, which they are free to partition in the way they want. These partitions define sub networks and reflect often sub domains in the domain namespace.

DNS is a very hierarchal system when we look at the aspects of domain name space, naming and organizational structure. If any of the name servers which hold information about a leaf in the domain space go down, the system does not function properly or not function at all. If a name server goes down, the consequence becomes greater the more central the name server is in the domain space.

1.1.3 DNS in Mobile Ad-hoc Networks

The Domain Name System (DNS) is a service that is heavily used in the Internet. Without this service the Internet would have been much more difficult to use. The alternative is to use IP addresses in order to connect to other host on a network. This is not intuitive for people mostly. People often get confused when they are presented a network address like an IP address. In addition it is much easier to remember a name like `www.ifi.uio.no` than `129.240.64.24`.

The existing DNS do not work in a MANET because of its characteristics as mentioned in Section 1.1.1. DNS is a service that is strongly bound to a client-server relationship. Whenever an application on a node wants to translate a name into an IP address a DNS request is sent to a central DNS server. We can not rely on such a relationship, since the topology is highly dynamic. There can not be a guarantee that the DNS server gets out of range from the other nodes. As mentioned in Section 1.1.2 the DNS is dependent on stable name servers. If a DNS server runs on an arbitrary node and gets out of range, the whole service fails to operate. This thesis investigates how a service like DNS can be designed for MANETs.

1.1.4 Claims

In order to serve MANETs a name resolution service, we must think of alternative solutions. In this section we present claims that our solution needs to meet.

- *Distributed service*. Unstable topology makes it impossible to base our solution on a centralized server approach.
- *Reduce overhead*. Nodes in MANETs often have constraints in their link speed, CPU speed and battery lifetime. The traffic on the network must therefore be minimized as much as possible.

- *Efficiency*. Bindings of IP addresses and names are desired to be made as fast as possible.
- *Application layer transparency*. Web browsers, e-mail clients and other applications should not have to be redesigned in order to work with our solution. Our design must be made in such a way that existing applications are not affected.
- *Independency*. Our solution must not depend on a huge number of existing solutions. This might lead to an unpredictable solution, because of increased complexity. In addition such solution might also not be an efficient solution.

These claims are referred to throughout the report. They are used in the evaluation of our solution and other existing solutions.

1.2 Terminology

This section describe a few terms that are used throughout this report:

- When talking about OLSR, it is meant the OLSR routing algorithm that is specified in the RFC 3626 specification.
- OLSRd is the implementation of the OLSR, where the RFC 3626 is the base of the design.
- Hosts and nodes on a network are words that are synonyms. It is an abstract term of devices that can exist on a network. Such devices can be ordinary computers, laptops, mobile phones, sensors etc.

1.3 Methodology

The methodology for this thesis can be divided into the following steps:

- *Literature study* – Consists of reading research papers, documentation for existing software and software that is used as basis for the design in the thesis.
- *Code study* – Source code study of OLSRd.
- *Design* – A development of a rough design that make a base for the implementation process.
- *Implementation and functionality testing* – The design of a solution is implemented based on the design. In addition the functionality is tested, in order to know if our solution works as intended.
- *Evaluation* – The implementation is more thoroughly analysed.

All the steps include documentation and work on the report. The organization of the report reflects the methodology used for this thesis, as we see in the next section.

1.4 Organization of the report

The report is organized as follows:

- Chapter 2 presents background material that is necessary in order to continue with further work on this report.
- Chapter 3 discusses how we develop a design for our solution.
- Chapter 4 describes how we implement our design and how we perform different functionality tests.
- Chapter 5 performs an analysis on our implementation. It designs different experiments, present measured results and analyses them.
- Chapter 6 summarizes the report and concludes how our solution fulfilled our claims declared in Section 1.1.4.

With this report a CD is attached as appendix. A full description of its content is given in Appendix A.

2 Background

In this chapter we get an overview of the background material that is related to this report. In Section 2.1, we look at related work that has been done on the problem that this thesis investigates. Then, in Section 2.2 we compare our claims with the solutions found. Further, the structure of this chapter reflects a discussion of background material that we are going to use to design a solution to our problem. Found solutions and their weaknesses is our base for choosing a solution.

2.1 Related work

The findings from the literature study are presented in this section. Selecting papers was a difficult process. A subject might be discussed in hundreds of related papers. When searching for related work on our subject, two libraries were used: *ACM digital library* and *IEEE xplore*. In addition, *Google Scholar* was used to find other interesting papers which were not found through the other libraries.

The keyword phrases that are used when searching through the libraries and *Google Scholar* are "name resolution MANET" and "naming MANET". The papers were selected based on the title. Reading through different papers gave an idea of what has been solved, and what kind of ideas that lay behind the solution.

When the search through the libraries was finished, suggestions on papers from the staff at the University played an important role. With various suggestions it is possible to see aspects of a problem.

As a result from this process a set of papers were selected. Section 2.1.1 and 2.1.2 describes related work found from library search and literature study. Section 2.1.3 is related work that was found through suggestions from the staff at the University and own findings from literature study. Through code study an additional solution was found on our topic. This finding is presented in Section 2.1.4.

2.1.1 Name resolution system based on a reactive routing protocol

There exist many reactive routing protocols for MANETs [2]. Ad-hoc On-demand Distance Vector (AODV) and Dynamic Source Routing (DSR) are two examples of such routing protocols, and act similar. In AODV a node broadcasts a *Route Request* (`RREQ`) when a route to another node does not exist. Every intermediate node maintains a reverse path towards the destination. The destination node unicast a *Route Response* (`RREP`) packet along the reverse path after the `RREQ` is received.

Engelstad proposes in [3] a mechanism that extends the traditional `RREQ/RREP` message with DNS query message. This extension is illustrated in Figure 2.1. A source node puts the *Name Resolution Request* (`NREQ`) into the `RREQ` packet that is spread throughout the network. This source node is now referred to as a *Name Resolver* (NR). These packets are flooded according to the arrows that are not dotted in Figure 2.1. If the node receiving the `RREQ` does not

understand the NREQ extension, it ignores the subsequent process and just forwards it to other neighbour nodes.

When the destination node receives the request, it generates the piggyback *Name Resolution Reply* (NREP) message as well as RREP, and unicast them back to the source node. The destination node is referred to as a *Name Server* (NS), since it is able to resolve the received request. In Figure 2.1 the reply is sent along the path described by the dotted arrows.

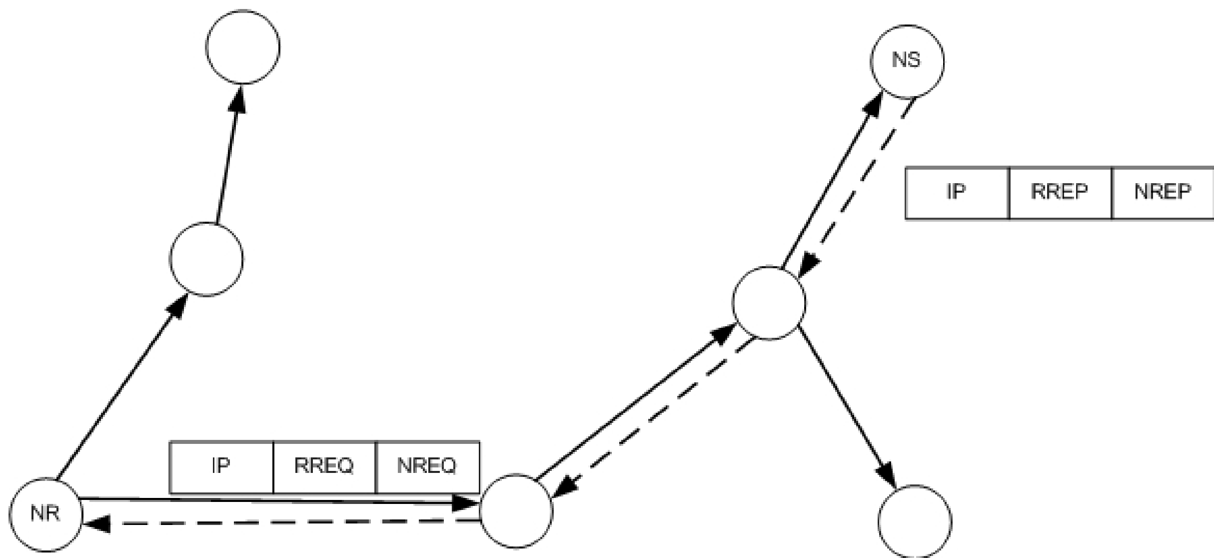

Figure 2.1: Extension of a reactive protocol's route requests and route replies.

This solution made a base for two solutions in [3]:

- A fully distributed system.
- A partially distributed system.

A fully distributed name resolution system (see Figure 2.2) means that all nodes participating in the MANET are independent of other nodes to make the name resolution system work. This means that all nodes are a potential *Name Server*. In addition, all nodes are responsible for their own participation in the name resolution system. For a MANET this is suitable because that all nodes are autonomous.

The fully distributed system requires two processes:

- *Find* – A name resolver needs to find the name server by flooding name requests as illustrated in Figure 2.1.
- *Bind* – When the name server is found a binding must be created between the name server's name and the resolved IP address.

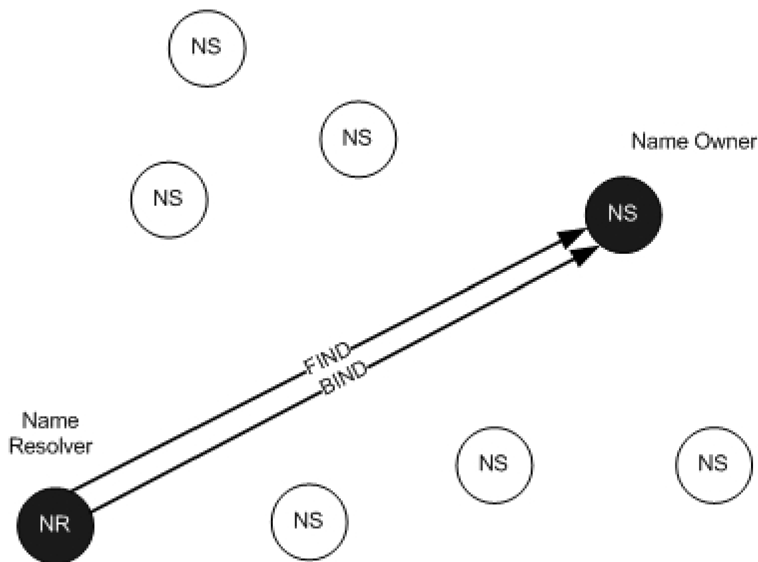

Figure 2.2: A fully distributed name resolution system.

Nevertheless, a fully distributed name resolution system generates much overhead because of flooding. A partially distributed name resolution was therefore also proposed in [3].

In the partially distributed approach, illustrated in Figure 2.3 , there are only a few numbers of elected *Name Servers*. When we discuss the partially distributed service the *Name Server* is called a *Name Coordinator*. A *Name Owner* (a MANET node that wants to make its name discoverable by other MANET nodes) must first find a name coordinator in its surroundings and register with it. Hence, with this approach name resolution comprises three phases:

- *Registration* – A name owner must find a name coordinator in its surroundings and register with it.
- *Find* – A name resolver needs to find the name server by flooding name requests as illustrated in Figure 2.1.
- *Bind* – When the name server is found a binding must be created between the name server's name and the resolved IP address.

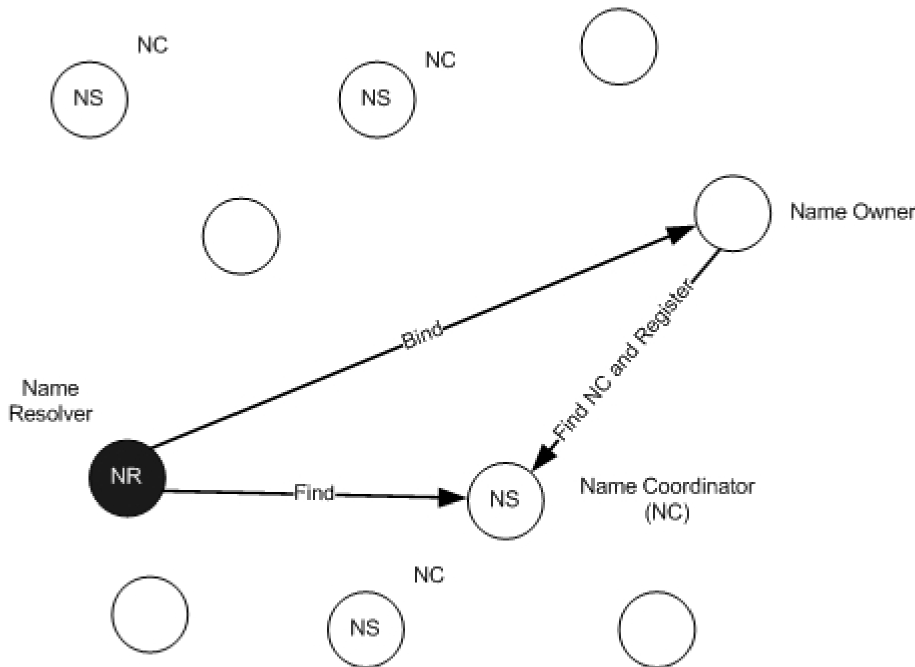

Figure 2.3: A partially distributed name service.

Figure 2.3 describes how a name owner first must find and *Register* with a *Name Coordinator*. Then a *Name Resolver* can query the *Name Coordinator* to *Find* the *Name Owner*. Finally the *Name Resolver* is able to create a *binding* between the name of the name owner and the resolved IP address.

The solutions in Figure 2.2 and Figure 2.3 both provide a name resolution system for a MANET. Engelstad concludes in [3] that there are a couple of potential problems according to the partially distributed system. One of the problems is due to the characteristics mentioned in Section 1.1.1. The partially distributed solution involves using a centralized name service. In a MANET we can not guarantee the stability of such a service because of dynamic topology.

Another problem is that the route discovered at the name resolver is not a direct route to the *Name Owner*, but through the *Name Coordinator*. In order to find a direct route, the *Name Resolver* must broadcast a new route request to the resolved IP address.

The last disadvantage is that the complexity increases since the partially distributed system involves more processes to make it work than the fully distributed system.

2.1.2 Partly distributed name resolution system with clusters

This idea is based on MANETs that are partitioned into clusters. The cluster mechanism is well described in [4] (chapter 4).

A cluster is composed of an arbitrary number of nodes, where each node plays different roles. The most central node is called the cluster-head, and is the node that defines the cluster. From a given node in a MANET the shortest distance to one of the surrounding cluster-heads decides which of the cluster the given node should belong to. If a node has equal distance between two cluster-heads in it surroundings, it is a member of both of these clusters.

There are several algorithms to elect cluster-heads. Some of them are mentioned in [5] and [6].

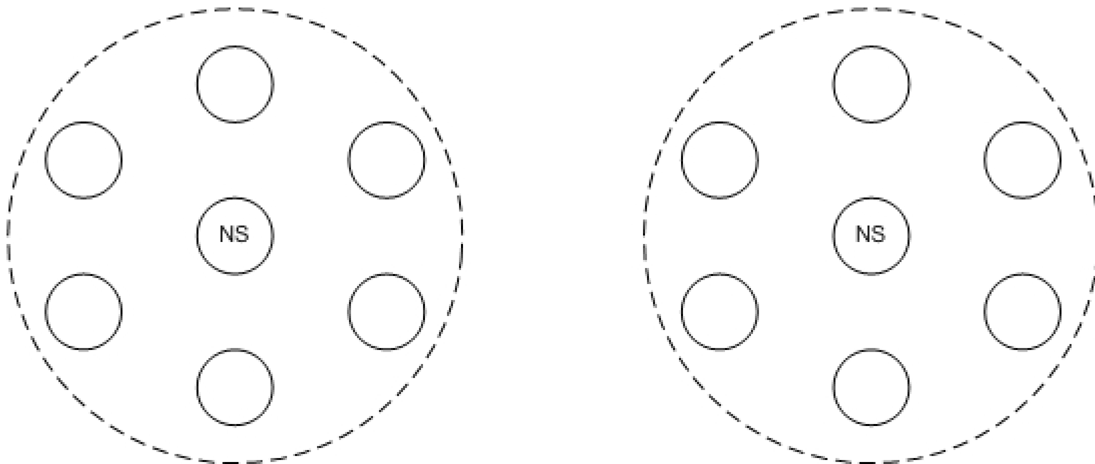

Figure 2.4: A MANET partitioned into clusters.

Hong et al. proposes in [7] a name resolution system that focuses on a partial distributed name resolution system using clusters and cluster-heads to elect name servers. Figure 2.4 illustrates the idea of this solution. This is a partial distributed service, because only elected nodes play the role as a name server. These servers must announce themselves as name servers. Other nodes are then able to detect existing name servers, and query them for name resolution.

The main advantage of this solution compared to Engelstad's solution [3] is that a partial distributed system generates less overhead.

The disadvantage is that the complexity of a partially distributed name resolution system does not necessarily decrease. A main challenge is to choose a cluster algorithm that works fine with an unstable topology as we find in MANETs. Choosing wrong cluster algorithms might lead to frequent re-election of cluster-heads if we have an unstable topology. Nodes in a cluster might then be confused about which node is the actual name server at a given time. In addition a partial distributed service involves a centralized client-server relationship. In MANETs, there is always a risk in depending on such a relationship because of the dynamic topology.

2.1.3 MIDAS – Naming by extending OLSR

MIDAS [8], [9] is a project initiated and funded by the IST programme of the European Commission. It started in Jan '06 and will finish June '08. One of the project's tasks is to design middleware for connectivity and information-sharing over hybrid networks. Such networks include MANETs.

The MIDAS working group has proposed, in [10], a design for opportunistic connectivity services. In this design they have issued the naming aspect of addressing in networks like MANET. As routing protocol they have selected the Optimized Link State Routing (OLSR) protocol [2]. The solution of a name resolution system described in [10] is build on OLSR. The MANET scope of this solution involves extension of OLSR to propagate both names and IP addresses. How this is done is not specified in [10]. The idea is that all names should be stored in an extended version of the OLSR routing table.

In contrast to the solutions in Section 2.1.1 and 2.1.2 that are build on a reactive routing protocol, this is a solution that is built on a proactive routing protocol (OLSR). Since this approach has not come further than the design declaration, it can be an interesting starting point for our work in this thesis.

The main advantage is that all traffic that needs to be flooded is forwarded through the *Multipoint Relay* (MPR) flooding scheme. This flooding scheme reduces greatly overhead generated by control traffic. The disadvantage is that if we extend the OLSR protocol compared to the RFC 3626 [11], older versions might not be compatible if they exist on the same network.

2.1.4 OLSRd plug-in

OLSRd [12] is the implementation of OLSR. It is implemented according to the specifications in RFC 3626 [11]. This implementation has support for plug-ins that are extensions to the functionality of the protocol itself.

A plug-in called *Nameservice*, developed by Bruno Randolf [12], distributes the name of the node to all other nodes on the network by special name messages. (More information can also be read in the `README` file that belongs to the OLSRd plug-in. This file can be found in the CD-ROM content in Section 0). When a node receives a name message, it stores the name together with the source address in a table. If all nodes on a MANET uses OLSRd with this plug-in, each node is able to maintain a table of bindings between IP addresses and names. The plug-in is also able to write necessary information to files so that each node can act as a name server in a fully distributed manner.

The advantage of the plug-in is that it takes advantage of the OLSR *Multipoint Relay* (MPR) flooding scheme when it broadcasts the name messages. Overhead generated by flooding is greatly reduced with this mechanism.

The broadcasting of a node's name is only done every second minute, and is a strategy to reduce overhead. A disadvantage of this is that it will take quite a long time to discover all names on the network. Another disadvantage is that the validity time of each entry in the table is one hour. If no new name message is received from a given node for a while, it is a high possibility that this entry is not valid in reality, because of the unstable topology of the MANET. Nevertheless, if the topology is stable, this plug-in works fine.

2.2 Existing solutions versus our claims

This section describes how existing solutions meet our claims from Section 1.1.4 of a name resolution in MANETs. Table 2.1 gathers all solutions found in Section 2.1, and point out which of the claims they meet (+) and do not meet (-). (+/-) means that the solution does not fulfill the claim 100 percent.

Solutions/Claims	Distributed service	Reduce overhead	Efficiency	App. layer transparency	Independency
Engelstad's extension of a reactive protocol	+	-	+	+	+
Hong et al.'s clustering approach in a partially distributed service	+/-	+/-	+	+	-
Extension of OLSRd control messages and data sets.	+/?	+/?	+/?	+/?	+/?
OLSRd name service plug-in	+	+	-	+	+

Table 2.1: This table shows how existing solutions meet our requirements.

Table 2.1 has a row that describes "Extension of OLSRd data sets and control messages". This solution is the proposed solution in [10] which was discussed in Section 2.1.3. Since it has not yet been implemented it is impossible for us to say whether or not it meets our claims. The (+/?) symbol has therefore been used to indicate this.

Nevertheless, this solution is an interesting approach as a starting point of view for developing a new solution. This involves extending OLSR in such a way that we will not end up with the same disadvantages that we have already found. The solution in Section 2.1.4 is interesting with respect to comparing our solution, since it already has been implemented and is ready for testing.

The next section presents some background information on OLSR. We need this information in order to start designing an own name resolution system based on an extended version of OLSR.

2.3 Optimized Link State Routing – OLSR

The Optimized Link State Routing (OLSR) is a proactive routing protocol built on an ordinary link state algorithm. Unlike a reactive routing protocol like AODV, OLSR is table driven. The behaviour of a reactive protocol and a proactive (table driven) protocol is completely different. A main difference is that a proactive routing protocol periodically broadcasts control messages to gather information from all nodes. It is then able to know the structure of the whole topology. A reactive protocol only acts if there is no route to host and a message is to be sent. Only if that occur, it broadcasts a route request.

All presented material in this section is based on the RFC 3626 [11] specification of OLSR and another master thesis [13] written by Andreas Tønnesen who is also responsible for the implementation of OLSRd.

2.3.1 OLSR terminology

OLSR deals with some terms that need some explanation:

- A *main (interface) address* is the IP address that is used in the control traffic as *originator address*. If a node only has one interface, this must be set to be the main address. A node with multiple interfaces must choose one of these interfaces to be its *main interface address*.
- The *originator address* is the same as the source address of a packet

.

2.3.2 OLSR Information Repositories

OLSR uses a set of control messages to exchange information between nodes. In order to keep all the information from OLSR control messages, OLSR has different databases where the information is recorded. This information is recorded or updated every time a control message is received. The databases are in OLSR referred to as sets. These sets are illustrated in Figure 2.5.

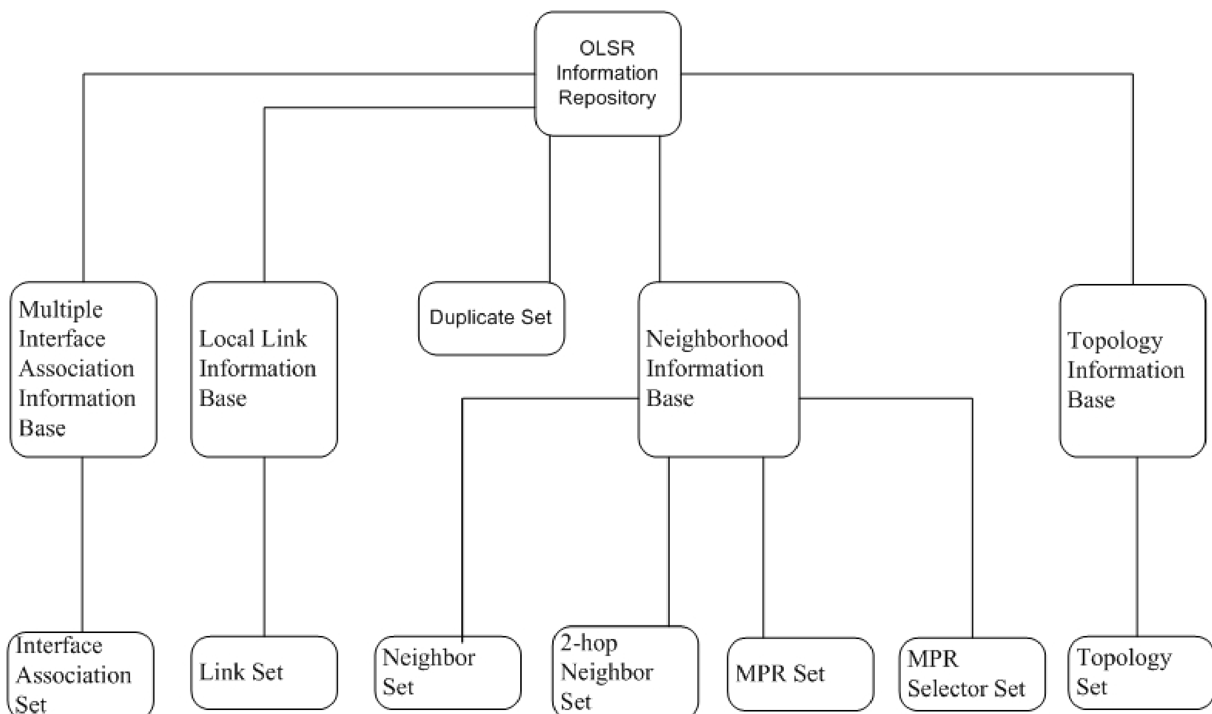

Figure 2.5: The OLSR information repositories.

As we can see from Figure 2.5, almost each set belongs to an information base. An exception is the duplicate set. All information bases have only one set, except from the neighbourhood information base which has four sets bound to it. This organisation of the database sets is mainly to get an intuitive structure of what kind of information a node keeps track of. We now describe in detail the purpose of each set:

- *Interface Association Set* – Holds information about nodes that have multiple interfaces that participate on the same network.
- *Link Set* – This set keeps track of all links and their state of nodes running OLSR.
- *Neighbour Set* – This database records all neighbours found.
- *2-hop Neighbour Set* – Records all 2-hop neighbours.
- *MPR Set* – Holds information about all nodes this node has chosen as a MPR node.
- *MPR Selector Set* – Records every node that has chosen this node as a MPR node.
- *Topology Set* – Records all possible destination nodes on the network. The information stored here reflects the topology the network.
- *Duplicate Set* – Keep track of all incoming packets to ensure that only packets with new information will be forwarded from a node.

This is a very general introduction of the OLSR information repositories. We later in Section 2.3.5 look at how they actually are used, when we describe more of the OLSR algorithm. We then get more detailed information about the different technical terms that are used in this section.

2.3.3 OLSR Packet Format

In OLSR, the control messages are an important part of the algorithm. OLSR has defined its own packet format that is used to transfer the different control messages in a standardized way. The basic layout of any packet in OLSR is illustrated in Figure 2.6(omitting IP and UDP headers).

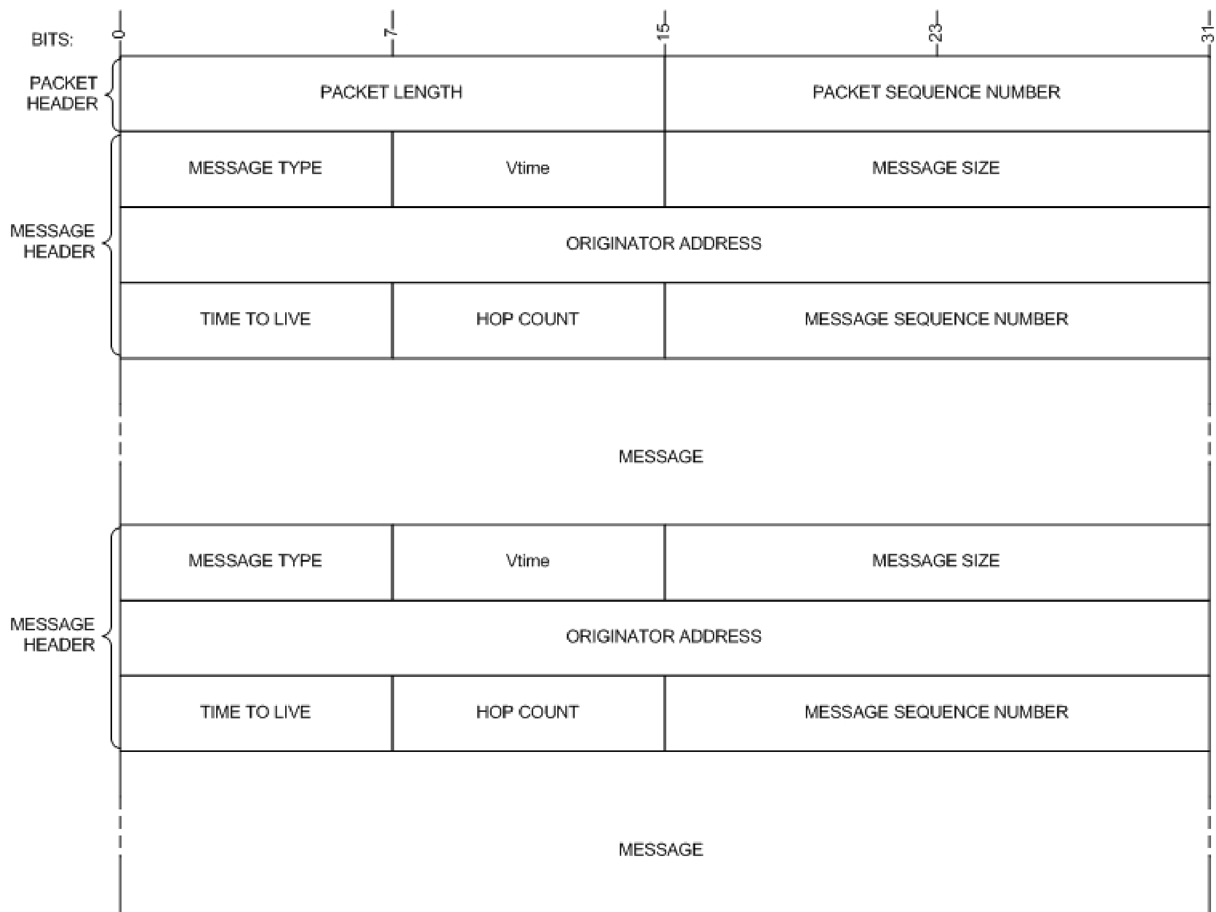

Figure 2.6: The basic layout of any packet in OLSR.

As we can see from Figure 2.6, the header is divided into two parts. Packet header is the header which tells us information about the whole packet. The message header is information about the message. A packet can contain multiple messages. This is indicated with the dotted line in the message field in Figure 2.6.

The packet header consists of the following fields:

- *Packet Length* – Is the length of the whole packet in bytes.
- *Message Type* – Is the type of the control message included in the packet.

The message header consists of the following fields:

- *Vtime* – The validity time of the message. This field indicates how long the information carried in the message after reception shall be considered as valid.
- *Message Size* – The size of this message including the message header.
- *Originator address* – Main address of the originator of the message.
- *Time To Live* – This field indicates how many times the message can be forwarded.
- *Hop Count* – The number of times the message has been forwarded.

- *Message Sequence Number* – Each time a new OLSR packet is being sent from a node, this field is incremented by one to ensure freshness of the information in the message.

This packet format is used whenever a control message is generated and sent. The messages carried in the packet have their own format. These are discussed in the next section.

2.3.4 OLSR Message Types

This section presents the three most important control messages. All these messages are encapsulated into the message part of the general OLSR packet format as discussed in Section 2.3.3.

2.3.4.1 Multiple Interface Declaration message format

The Multiple Interface Declaration (MID) message has the format illustrated in Figure 2.7.

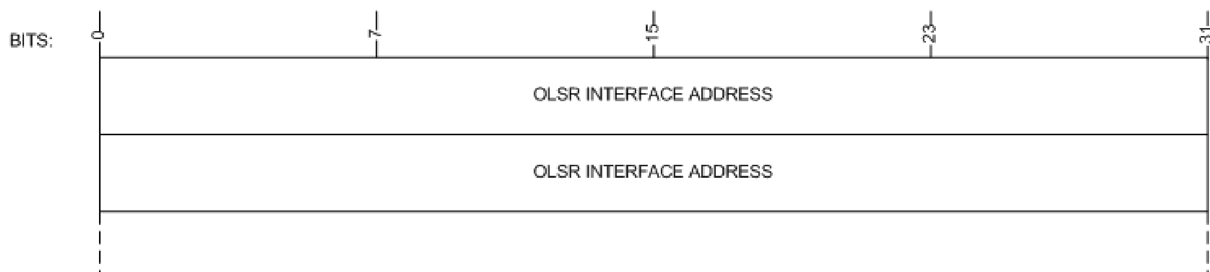

Figure 2.7: The OLSR MID message format.

This message is sent in the message part of the general OLSR packet format and only contains the interface addresses of the node. If a node does not have multiple interfaces, MID messages are not sent or generated.

2.3.4.2 HELLO message format

The message part of the general OLSR packet format when sending a HELLO message has the format illustrated in Figure 2.8:

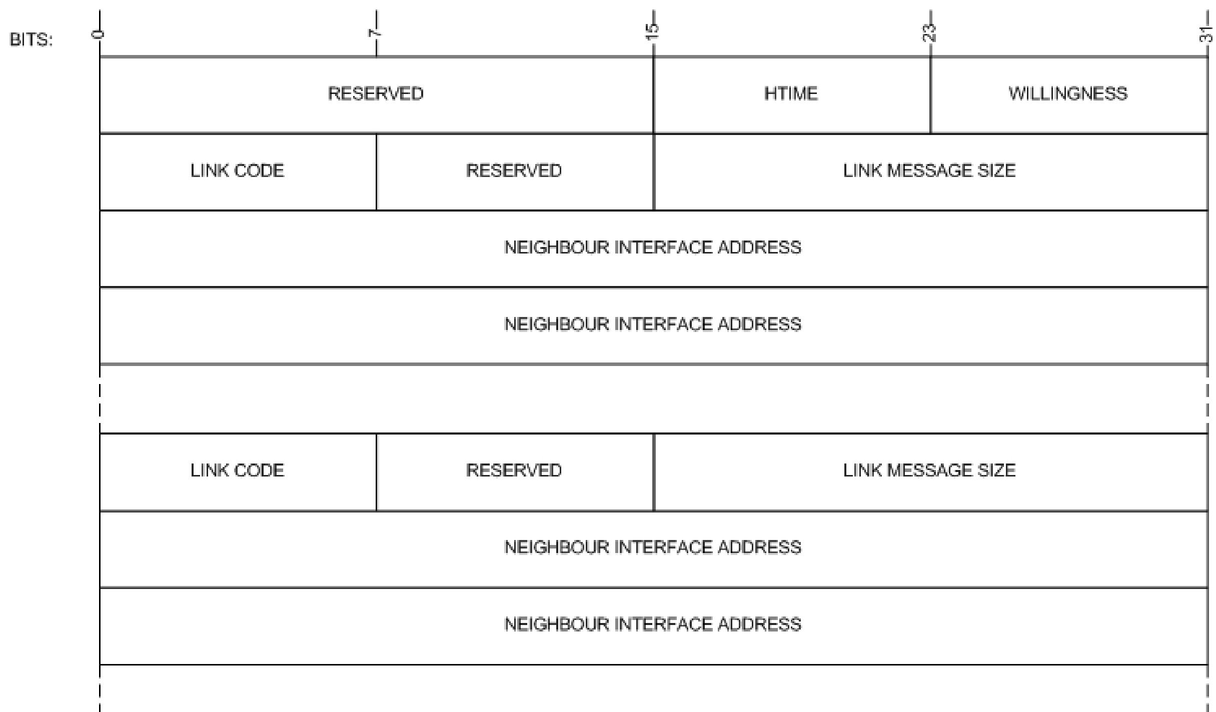

Figure 2.8: The format of the OLSR HELLO message.

As we can see from Figure 2.8, the first 32 bits are only sent in the first portion of the message. The next 32 bits are header values that are sent in each portion. After these headers the neighbour interface addresses are stored. If the message has to be split up, it continues with the second of the 32 bits headers followed by the next neighbour addresses.

The *Link Code* field has a special format that announces the *Neighbour Type* and the *Link Type*. This format is illustrated in Figure 2.9.

Figure 2.9: The 8-bit Link Code field in the OLSR HELLO message.

We now look at the HELLO message in more detail; it contains the following fields:

- *Reserved* – There are two fields that have this name. These are fields that are reserved for future possible extensions. The implementation of OLSR also needs to have these fields in order to meet the RFC 3626 [11] specification.
- *Htime* - This field specifies the HELLO emission interval used by the node on this particular interface, i.e., the time before the transmission of the next HELLO.
- *Willingness* - Specifies the willingness of a node to carry and forward traffic for other nodes.
- *Link Code* - Specifies information about the link between the interface of the sender and the following list of neighbour interfaces. It also specifies information about the status of the neighbour.

- *Link Message Size* - The size of the link message counted in bytes and measured from the beginning of the *Link Code* field and until the next *Link Code* field.
- *Neighbour Interface Address* – The address of an interface of a neighbour node.

2.3.4.3 Topology Control message format

The format of a Topology Control message is illustrated in Figure 2.10.

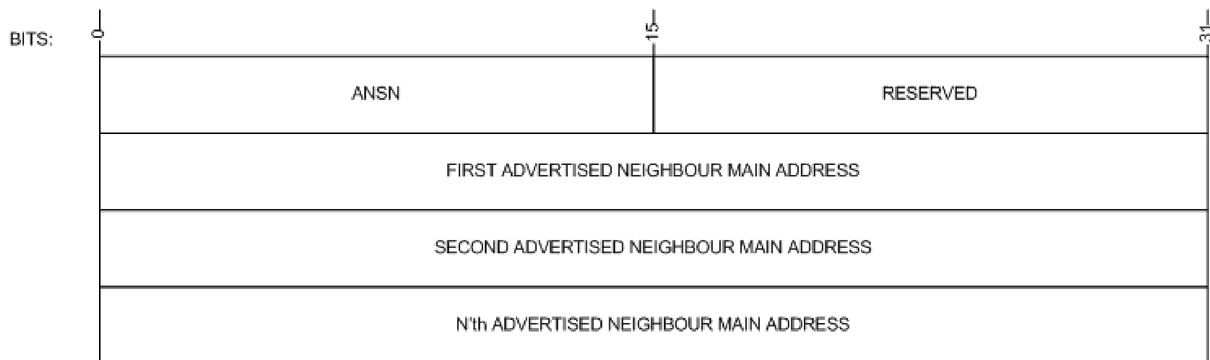

Figure 2.10: The format of the OLSR Topology Control message.

This is sent in the message part of the general OLSR packet format. The members in this message format are now presented in more detail:

- *ANSN* – This is a sequence number that the originator of the message increases each time it detects changes in its neighbour set. In that way nodes can always get the most recent information from the originator node of the TC message.
- *Reserved* – According to the specifications in RFC 3626 [11], this field must be set to only zeroes.
- *Advertised Neighbour Main Address* – The main address of a neighbour of the originator of the message.

2.3.5 OLSR functionality

Previous section described the three most important OLSR control messages. This section discuss how these elements work together to achieve the goal of routing.

2.3.5.1 Multiple interfaces

Nodes on a network can have more than one interface which is connected to the same MANET. The interfaces will have their own address. A mechanism for other nodes to understand that these addresses belong to the same node is needed in OLSR. *Multiple interface declaration* (MID) messages, presented in Section 2.3.4.1, are used to diffuse information about multi-homed nodes. A MID message contains a list of all interface

addresses the originator has. Upon receiving a MID message, a node updates the Multiple Interface Association Information Base according to the information carried in the message. All OLSR interfaces listed in the MID message are registered on the main address of the originator. The main address is the address among all interface addresses. This is chosen arbitrarily from the node itself. When adding a route to a node, OLSR adds routes to all addresses of other interfaces on which the remote node runs OLSR, using the same path.

2.3.5.2 Link sensing

Link sensing is done to keep up to date information about links that exists between a node and its neighbours. The link sensing is done with periodical exchange of HELLO messages, presented in Section 2.3.4.2. By exchanging information in HELLO messages, the link set is updated in each node. The main goal with link sensing is to find out whether or not it is possible to send a message on a link. Each link to a neighbour gets an associated state of either symmetric or asymmetric. Symmetric state indicates, that the link to that neighbour node has been verified to be bi-directional. This means that it is possible to transmit data in both directions. Asymmetric state indicates that HELLO messages from the node have been heard, however it is not confirmed that this node is also able to receive messages.

During the process of link sensing a node records a set of *Link Tuples* in the *Link Set*:

`(L_local_iface_addr,L_neighbour_iface_addr,L_SYM_time,L_ASYM_time,L_time)`

Where `L_local_iface_addr` is the interface address of the local node and `L_neighbour_iface_addr` is the interface address of the neighbour node. These to interface addresses define the two endpoints of a link.

`L_SYM_time` is the time in which the neighbour interface is considered symmetric and `L_ASYM_time` is the time in which the neighbour interface is considered heard. `L_SYM_time` is used to decide the Link Type declared for the neighbour interface. If `L_SYM_time` is not expired, the link must be declared symmetric. If `L_SYM_time` is expired, the link must be declared asymmetric. If both `L_SYM_time` and `L_ASYM_time` are expired, the link must be declared lost.

The last member of the tuple is `L_time`, which specifies the time at which this record expires and must be removed.

The information stored here is used for population the neighbour set. This is to be discussed in the next section.

2.3.5.3 Neighbour detection

As classical link state routing algorithms, OLSR also needs a mechanism to detect neighbours. In OLSR, neighbours are only nodes that can be reached within a one-hop radius. A neighbour can have the state of symmetric or asymmetric. The information that tells the node if a link is symmetric is found in the *Link Set* that is populated through the link sensing as described in Section 2.3.5.2. HELLO messages (Section 2.3.4.2) are periodically sent to

accomplish this task. A simplified explanation of how this is done in OLSR is illustrated in Figure 2.11.

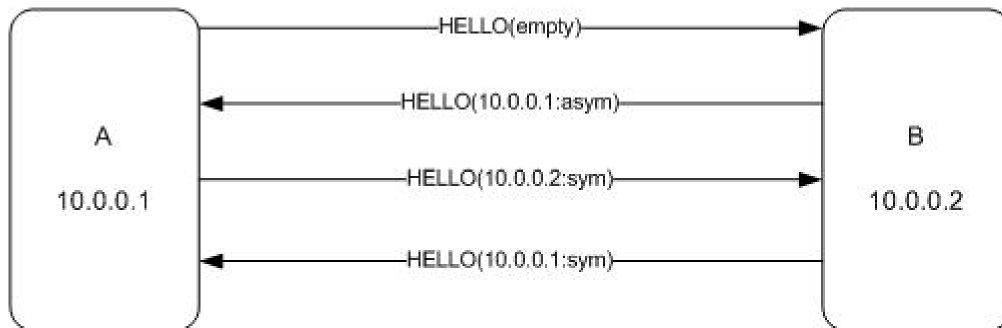

Figure 2.11: The process of creating a symmetric link between two neighbours.

The process of neighbour detection in Figure 2.11 can be summarized as follows: *A* sends an empty HELLO message. *B* receives it and registers *A* as an asymmetric neighbour. The reason is that *B* can not yet find its own address in the HELLO message. *B* sends a HELLO message to *A*, where it tells *A* that *B* has registered it as an asymmetric neighbour. *A* will in the HELLO from *B* find its own address and will now register *B* as a symmetric neighbour. *A* sends a HELLO to *B* again which tells *B* that *A* has declared *B* as a symmetric neighbour. *B* will now register *A* as a symmetric neighbour, and tell *A* about this in a HELLO message.

Neighbour detection populates the *Neighbour Set*. Link sensing and neighbour detection are closely related. Whenever there is created a new link entry, a corresponding neighbour entry is recorded in the *Neighbour Set*. The difference is that a node can have several link entries describing different links to the same neighbour, but there is only one neighbour entry per neighbour. Every neighbour entry must only be recorded with the neighbour node's main address.

In addition, there is also up to date information about the status of a node when there are changes in the link set. A neighbour is said to be a symmetric neighbour if there exists at least one link entry where the symmetric timer is not timed out. When a link entry is deleted, the corresponding neighbour entry is also removed if no other link exists for this neighbour.

A node records a set of *Neighbour Tuples* in the *Neighbour Set*:

```
(N_neighbour_main_addr, N_status, N_willingness)
```

Where `N_neighbour_main_addr` is the main address of a neighbour. `N_status` specifies if the node is `NOT_SYM` (not symmetric) or `SYM` (symmetric). `N_willingness` is an integer between 0 and 7, and specifies the node's willingness to carry traffic on behalf of other nodes.

2.3.5.4 Two-hop neighbour detection

This mechanism entails updating the *2-hop Neighbour Set*. Two hop neighbours are nodes that are reachable through symmetric neighbours. Upon receiving a HELLO message from a symmetric neighbour, all announced neighbours are added or updated in the two hop neighbour set.

A node records a set of *2-hop tuples* in the *2-hop Neighbour Set*:

```
(N_neighbour_main_addr, N_2hop_addr, N_time)
```

`N_neighbour_main_addr` is the main address of a neighbour. `N_2hop_addr` is the main address of a 2-hop neighbour with a symmetric link to `N_neighbour_main_addr`. `N_time` specifies the time at which the tuple expires and must be removed.

2.3.5.5 Multipoint relaying

Like other link state routing algorithms, OLSR also need to flood the network with link state declaration packets. The purpose of these packets is that they announce to all other nodes which neighbours they have. In OLSR these packets are called *Topology Control* Messages (TC messages) and were presented in Section 2.3.4.3.

OLSR takes advantage of *Multipoint Relaying* (MPR) when flooding the network. The main purpose with MPR is to reduce the total amount of overhead generated from the control messages when they are broadcasted.

Each node that uses OLSR keeps track of all its two hop neighbours. A subset of the nodes of the symmetric neighbour is maintained. Through this subset of nodes, the node, which maintains this subset, can reach all two hop neighbours. The optimization is illustrated in Figure 2.12.

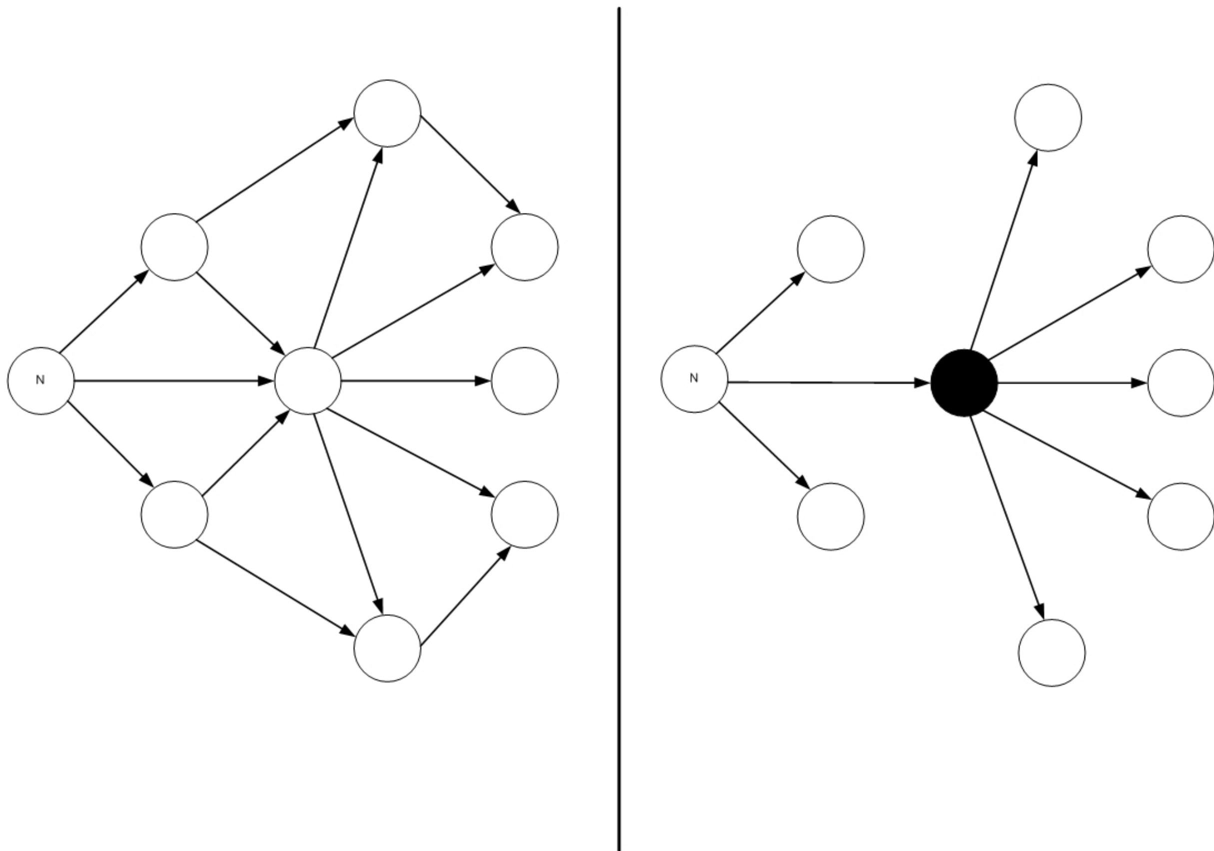

Figure 2.12: The difference between classical flooding and MPR flooding.

The left part in Figure 2.12 illustrates a classical flooding scheme and the right part the MPR flooding scheme. The node coloured in black is a node which node N has chosen as a MPR node. As we see the overhead is greatly reduced. It is reduced even more if the density of the nodes is high, compared to the classical flooding scheme.

Each node announces its willingness to forward messages in HELLO messages to their neighbours. If a node announces that it never forwards a message, this node must never be chosen as MPR. Based on the different node's willingness a calculation is performed to make the MPR set as small as possible. This means that every two hop neighbour can be reached with a minimum number of 1-hop neighbours.

Based on the willingness in the HELLO messages and the MPR calculation algorithm, the *MPR set* gets populated. This set is maintained by each node to store the set of neighbours which are selected as MPR. The *MPR Set* only records the neighbours' main addresses.

2.3.5.6 MPR Selector detection

The MPR flooding scheme, that we discussed in 2.3.5.5, requires that nodes register which neighbours have chosen them as MPR. This is done by setting the *Neighbour Type* in Figure 2.9 to `MPR_NEIGH` in *Link Code* field of the HELLO message. Upon receiving a HELLO message, a node checks the announced neighbours in the message (Figure 2.8) for entries matching one of the addresses used by the local node. If an entry has a matching address and

the neighbour type of that entry is set to `MPR_NEIGH`, then an entry is updated or created in the MPR selector set using the main address of the sender of the HELLO message.

This mechanism populates the *MPR Selector Set* with *MPR selector tuples*:

```
(MS_main_addr, MS_time)
```

`MS_main_addr` is the main address of a node, which has selected this node as MPR. `MS_time` specifies the time at which the tuple expires and must be removed.

2.3.5.7 Distribution of Topology Control messages

Link state routing algorithms entail that all nodes flood their information about their local links. This is done by using *Topology Control* (TC) messages.

Based on the information received in a TC message, a node can get an overview of the whole topology. TC messages are, as mentioned, flooded using the MPR flooding scheme. In OLSR, based on the RFC 3626 specification [11], only the nodes that are chosen as a MPR node are generating TC messages. The result is less overhead, since there is a high possibility that there are less MPR nodes, than ordinary nodes. In addition, the TC messages are flooded throughout the network using the MPR flooding scheme. With these two techniques OLSR is greatly optimized compared with ordinary link state routing algorithms.

TC messages are generated on a regular interval and immediately when changes are detected in the *MPR selector set*.

In order to keep track of the freshness of the information contained in the message, we need to have some sort of a counter which indicates this. In OLSR, the TC messages contain a sequence number called *Advertised Neighbour Sequence Number* (ANSN). Whenever a node detects a change in its advertised neighbour set, the ANSN is increased.

As a result from exchanging TC messages to nodes all over the network, the topology set is populated. Upon receiving a TC message, the header values are inspected and processed from the following rules:

- If no entry is registered in the *Topology Set* on the address of the originator, one is created with validity time and *ANSN* set according to the TC message header.
- If an entry is registered in the *Topology Set* on the address of the originator and with ANSN lower than the received *ANSN*, then that entry is updated according to the received TC message.
- If an entry is registered in the *Topology Set* on the address of the originator with an ANSN equal to the received ANSN, then the validity time of the entry is updated.

Topology Tuples are recorded in the *Topology Set* and have these members:

```
(T_dest_addr, T_last_addr, T_seq, T_time)
```

`T_dest_addr` is the main address of a node, which may be reached in one hop from the node with the main address `T_last_addr`. Typically, `T_last_addr` is a MPR of `T_dest_addr`. `T_seq` is a sequence number, and `T_time` specifies the time at which this tuple expires and must be removed.

2.3.5.8 Forwarding of OLSR control messages

Based on the information in the OLSR *Information Repositories* and header fields of an incoming OLSR packet, a special forwarding algorithm is used:

1. If the link on which the messages arrived is not considered symmetric, the message is silently discarded. To check the link status the set is queried.
2. If the TTL carried in the message header is 0, the message is silently discarded.
3. If this message has already been forwarded the message is discarded. To check for already forwarded messages, the *Duplicate Set* is queried.
4. If the last hop sender of the message, not necessarily the originator, has chosen this node as MPR, then the message is forwarded. If not, the message is discarded. To check this, the MPR selector set is queried.
5. If the messages are to be forwarded, the TTL of the message is reduced by one and the hop count of the message is increased by one before broadcasting the message on all interfaces.

The *Duplicate Set* is an OLSR repository for storing messages that have already been forwarded. The message itself is not stored. Just the originator address, message type and sequence number is recorded and is enough to identify the message. This data is cached for a fixed time of `DUP_HOLD_TIME` suggested to be 30 seconds. If any messages are forwarded from a node, a new entry in this repository is created. Then, if the node receives the message again after a while, the node finds it in this repository and discards it. In this way, a node reduces the number of unnecessary retransmissions.

2.3.5.9 Route calculation

This mechanism is common for all link state routing protocols. OLSR parses information from the *Information Repository* and calculates a routing table which contains a path to all nodes that is registered there. The routing table record tuples with the following members:

```
(R_dest_addr, R_next_addr, R_dist, R_iface_addr)
```

`R_dest_addr` is a destination address to a node which is `R_dist` number of hops away. `R_next_addr` is the address of the next node or first hop in the path towards the destination node. `R_iface_addr` is the address of the local interface which `R_next_addr` is reachable through.

From the specification in RCF-3626 [11], the algorithm to calculate the routing table is a shortest path algorithm. This shortest path can be summarized very simplified as follows:

1. First all one hop symmetric neighbours are added.
2. Then all two hop neighbours are added which have not already been added from the one hop neighbour set. A node can in theory exist in both the one-hop *Neighbour Set* and in the *2-hop Neighbour Set* which depends on the topology.
3. Finally, the Topology Set is iterated, and entries made in the routing table out of nodes that not already has been added.

2.3.6 OLSR Summary

As a summary, we can mention that OLSR is a link state routing protocols that is optimized by using the MPR flooding scheme. By using MPR, the overhead generated from control messages compared to classical link state routing algorithms is reduced.

The three most important control messages in OLSR are:

- Multiple Interface Declaration (MID) messages are generated by a node, if a node has multiple interfaces. MID messages are used to announce these addresses which should be mapped to the same node at another node.
- The purpose of HELLO messages is to perform link sensing, neighbour detection and MPR selection.
- Topology Control (TC) messages announce a node's link state. TC messages are emitted through the MPR flooding scheme.

In order to keep track of the information which the control messages give the nodes, different information repositories are maintained. These information repositories are also used when the control messages are generated. The content in the control messages are based on the information recorded in the OLSR information repository.

2.4 Background summary

The background chapter of this report has first presented related work on the problem of name resolution in MANETs. We gathered all the solutions in a table that shown us that none of them met all our claims to a name resolution system. The MIDAS working group has proposed a solution in a design document [10] that we base our further work on and try to design a new name resolution system to try to fulfil all our claims. OLSR was therefore briefly explained in order to be able to create a design.

3 Design

This chapter of the thesis discusses how our design of a name resolution system by extending OLSR is developed. First we present some design requirements. Then we go in more depth into what we need to do technically in order to be able to implement a name service for nodes on a MANET.

3.1 Design claims and assumptions

In Section 1.1.4, we declared a set of claims that our solution has to meet. The characteristics of a MANET that were described in Section 1.1.1 needs us to design a service that is a distributed service. This service must transfer data with a minimal cost and a high efficiency.

Section 2.1.3 discusses an alternative way of designing a name resolution system for MANETs by extending OLSR to propagate both IP addresses and names. Every node is then able to create bindings between IP addresses and names of all the nodes on the MANET.

In Section 2.1.4 we saw that OLSRd already has a plug-in extension that distributes names through new defined name messages. The main disadvantage with this solution is that the efficiency is poor, since the name messages are only broadcasted every second minute.

The solution proposed in this thesis aims to solve the efficiency problem. We want to be able to offer a node a binding of name and IP addresses of the other nodes whenever there is a route to a node. The most logical way to attack this problem is to extend the control messages, as we identified in Section 2.3.4. This is because that route discovery is done through these control messages. Nevertheless, we must take into consideration the problem of overhead that this approach creates. Our solution should try to keep a balance of overhead at a minimum and efficiency at a maximum.

Although this solution seems to be a good approach compared to other solutions, there are some consequences by extending the OLSR control messages. We therefore need to declare the following assumptions:

- All nodes on the MANET must use our extension of OLSR in order to make our name resolution system working.
- Compatibility between our solution and the standard version or older versions of OLSR can not be guaranteed.

With these claims and assumptions we have created a base for our solution. The next sections discuss the design of our solution in more detail and create a foundation of the implementation process.

3.2 Extending the control messages

As mentioned in Section 2.3.4 the three most important types of control messages in OLSR that offer us route discovery are HELLO, TC and MID messages. This section discusses which of the control messages need extensions and how it can be done to reach our goals that defined in Section 3.1.

3.2.1 TC messages

As explained in Section 2.3.5.7, Topology Control (TC) messages are used to announce neighbours of a node to all other nodes on the MANET.

By extending the TC messages to contain the name of the originator, all nodes on the network are able to know the name of nodes that are chosen as a MPR node. The reason is that all nodes that are chosen as MPR generate TC messages, which are flooded to all possible destinations.

In an ordinary link state routing algorithm, all nodes broadcast their link states. As we have already discussed in Section 2.3.5.7, this is not the case in OLSR. A node only generates TC messages if it has been chosen as MPR by another node(s). We have therefore yet to design a mechanism to distribute the names of the nodes which are not selected as MPRs. This is discussed further in Section 3.2.2.

TC messages have, as all of the other types of OLSR control messages, a message body. In TC messages the announced neighbours is stored as we can see in Figure 2.10. A node that is not chosen as MPR chooses another node as MPR if the MANET consists of at least one two-hop neighbour. The MPR node includes the main address of the MPR selector into the TC message. In order to distribute names of the nodes that are not MPR, the TC message body must also be extended to contain the name as well as the IP address.

The easiest way to design and implement this extended version of TC message is to statically allocate space for the name with a fixed number of bytes. In the standard C library there is defined a constant called `MAXHOSTNAMELEN` set to `256 bytes`. If we should allocate such a great space for the originator name and all of the neighbour names, the TC message could be potential very large.

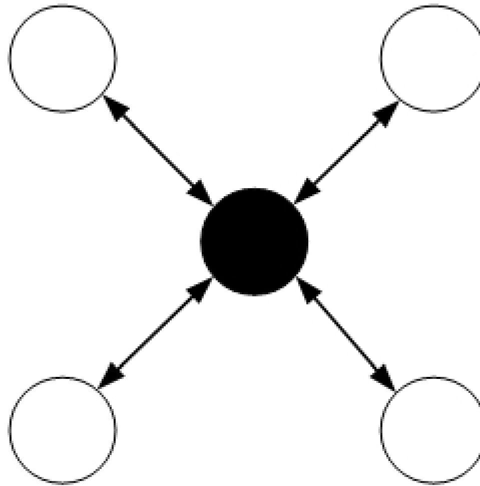

Figure 3.1: A MPR node with 4 MPR selectors.

As an example we can imagine a MANET that has a MPR node with four MPR selectors, visualized in Figure 3.1. For simplicity we assume that all hostname of all nodes have a length of 15 characters which means that all names need 15 bytes. In the TC messages, this means that only 15 out of 256 bytes are used. Nevertheless 256 bytes are sent.

The optimal solution only needs `15*5=75 bytes` to store all the names in a TC message. The solution from the last paragraph uses `256*5=1280 bytes`. Out of these `1280 bytes`, only `75 bytes` are used, and `1205 bytes` are transferred as empty information for the receiver of the TC message.

This example proves that such a design of a new TC message does not meet our design requirements in Section 3.1 to minimize the overhead in control messages as much as possible. A new design of a TC message must be developed to meet our requirements.

The next idea is to include a byte for each name field that tells us the length of the name. In that way we can dynamically allocate storage space for the name. As a result, the total overhead becomes almost optimal as described above. The names claim 75 bytes. In addition a byte is used for each name to tell us the length under processing of the TC message. The total extra overhead for this imagined TC message is 80 bytes.

If we look at the two different solutions regarding the design of the TC message, we can compare them together in a table with respect to the total overhead. The UDP-, IP- or the OLSR-headers are not included in this calculation. The example as shown in Figure 3.1 is used. There is one MPR node that has four neighbours connected to it.

Original TC-message:	Static name storage in the TC message.	Dynamic name storage in TC-message
160 bytes	1440 bytes	240 bytes

Table 3.1: A comparison of overhead with different solutions to include the name into the TC message.

If we have a MPR node with 4 neighbours the message size has the values shown in Table 3.1 dependent on the design of the TC message. We assume that all neighbours have an announced hostname of 15 bytes. We see if we for each name field should have allocated 256 bytes for a name, we get a very large TC-message.

As a conclusion from Table 3.1, the dynamic name storage must be chosen. The difference regarding the size of a TC message is that high, that we do not have any choice.

The new TC message is illustrated in Figure 3.2. The `ANSN` and the `RESERVED` fields are untouched and still existing. Next there is a name field where the name of the originator is stored. The name field is explained in Figure 3.3. Then the main address of the first announced neighbour is stored followed by the name field of this neighbour. Then the main address of an eventual second neighbour is stored, followed by its name field. The TC message body is generated in this manner until all neighbours are represented.

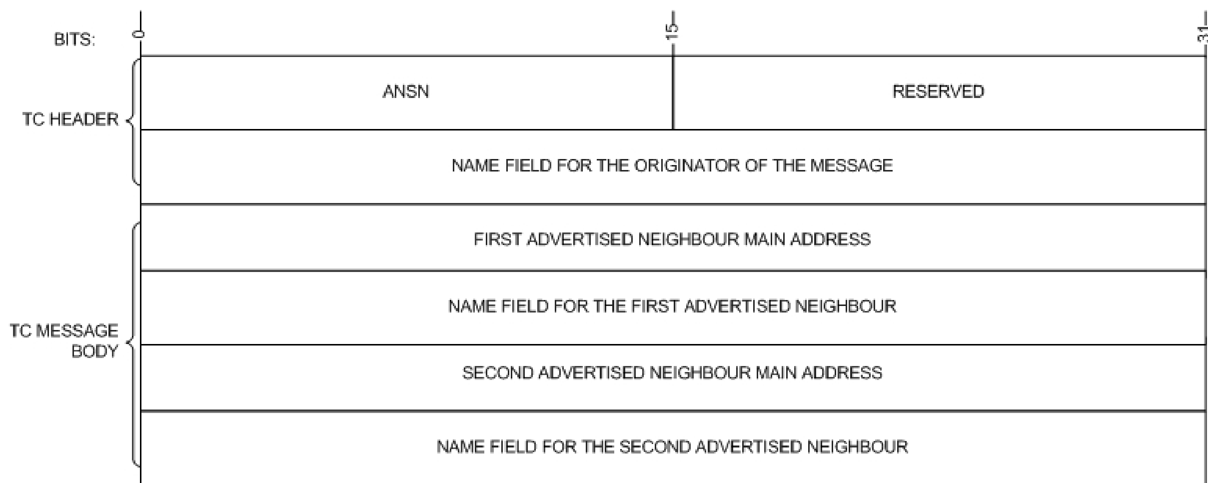

Figure 3.2: The new design of the OLSR TC-message.

The name fields used in the body of the TC message is described in Figure 3.3. One byte is used to tell the receiving node the length of the name. These fields of the TC message have variable lengths, which depends on the length of the hostname that is stored there. The length can have values between 0 and 255. 0 means that there no name represented for the node.

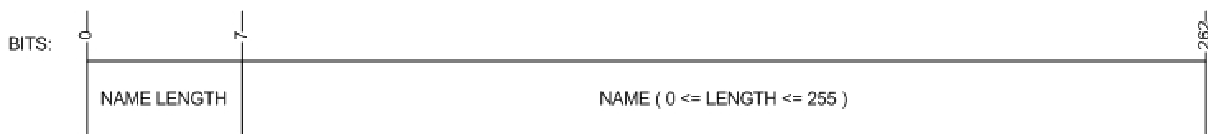

Figure 3.3: The name field used in the control messages.

3.2.2 HELLO messages

In Section 3.2.1 we described how we can distribute names with OLSR TC-messages. There is still one unsolved challenge: A node that is not chosen as MPR from any other nodes on the MANET is not generating TC-messages. We therefore also have to extend HELLO messages in order to distribute the name between neighbours. If one of these neighbours is chosen as MPR of one node, it begins generating and sending TC messages where all of its neighbours with names are announced.

As explained in Section 2.3.5, HELLO messages are used for link sensing, neighbour detection and MPR calculation. They are exchanged within the one-hop neighbourhood from

a node to accomplish these tasks. A HELLO message is never forwarded, since the information only makes sense between two neighbours.

The extension of HELLO messages is almost the same as for the TC message in Section 3.2.1. The difference is that we only have to include the name of the originator. We do not need to include the names of all of the neighbours that are announced in the message body.

The purpose of announcing neighbours through HELLO messages are just for making the receiver able to keep track of all two-hop neighbours and for later MPR calculation. Nevertheless, the design of this name resolution service based on OLSR focuses on distributing names through TC-messages. TC-messages are flooded to all possible destinations, and it therefore gives us no meaning of sending the name of the neighbours in the HELLO messages. We also minimize the overhead with this solution.

The new design of a HELLO message, shown in Figure 3.4, consists therefore only of a name field, as explained in Figure 3.3, that contain the name of the originator of the HELLO-message.

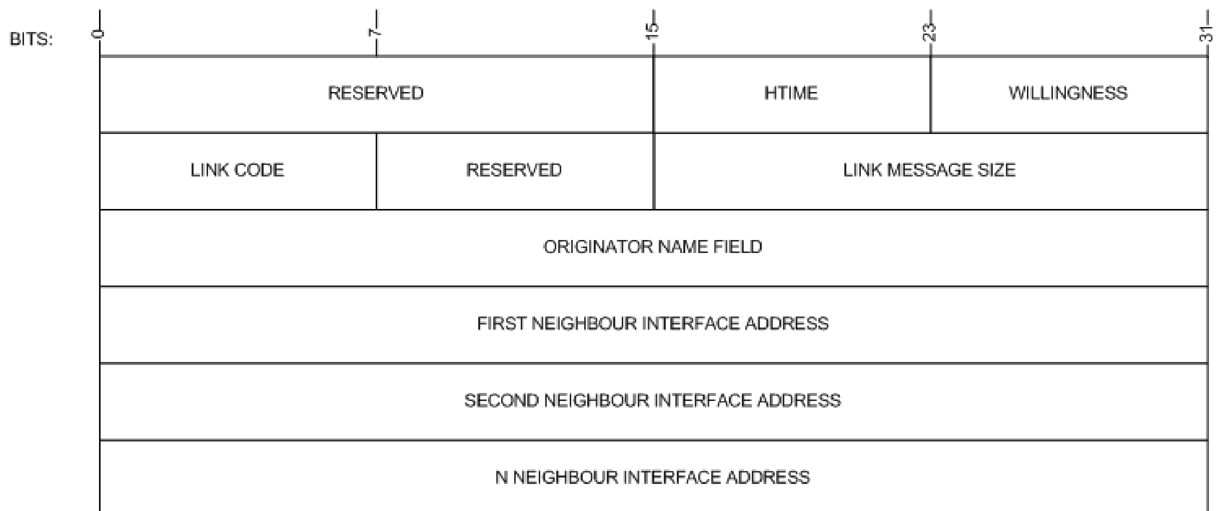

Figure 3.4: The new HELLO message contains just the name of the originator of the HELLO message. The name is stored in a name field as explained in Figure 3.3.

3.2.3 TC-messages and HELLO-message working together

In Section 3.2.1 and 3.2.2, we have focused on how we can extend the messages. But as mentioned in Section 3.2.2, an extension of TC messages can not serve a distribution of names alone. We need help of HELLO messages as we can see in Figure 3.5.

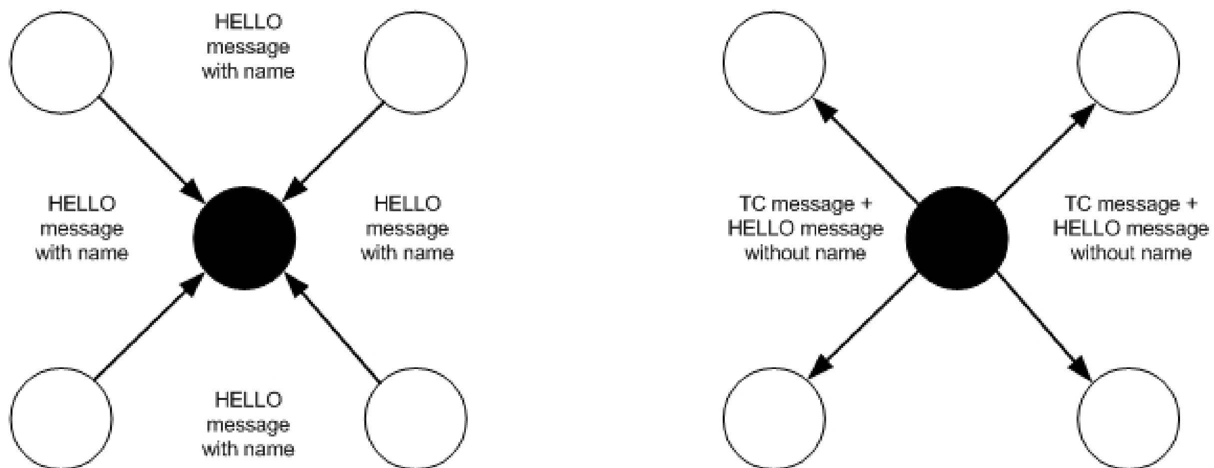

Figure 3.5: Name distribution with HELLO messages and TC messages working together.

TC messages are generated, based on the information from the HELLO messages. During exchanging HELLO messages, the *Neighbour Set* is populated. When the MPR node in Figure 3.5 generates a TC message it includes all of its neighbours. Our extension in Section 3.2.1 provides that all the names of each neighbour are included. To keep the overhead at a minimum, we not include names in the HELLO messages for a node which is generating TC messages. If a node generates TC messages, the name are only transmitted through TC messages to avoid wasting of bandwidth capacity.

3.3 Extending data structures

In order to store the extended information from the control messages in memory of each node, we also need to extend the data structure of OLSR. This can be done in two ways: Either by extending the routing table, or defining a new data structure that is shared and used upon receiving both HELLO and TC messages.

The most logical solution is to extend the existing OLSR information repositories. Upon receiving HELLO messages, the names are stored in the *Neighbour Set*. Likewise, upon receiving TC-messages the names are stored in an extended version of the *Topology Set*. From these sets, an extended version of the routing table is created, which holds all bindings to all possible destinations. There are two arguments against doing it this way:

If we should extend the routing table, the routing table calculation algorithm must be modified as we shortly explained in Section 2.3.5.9. In addition the two-hop neighbour set must also be extended with name, since the second step in the routing table calculation algorithm is to populate the routing table with two hop neighbours. This means that we also have to store names about every announced neighbour in the HELLO messages. The result is that unnecessary much overhead is generated. In addition this does not coincide with our earlier discussed design in Section 3.2.2.

The second argument against extending the existing data structure to store names is that names can be faster and more independent detected between neighbours. If we create an own name table for OLSR, that is used upon processing HELLO messages and TC messages as we can see in Figure 3.6, a node can record the names from both of them without going through the routing table.

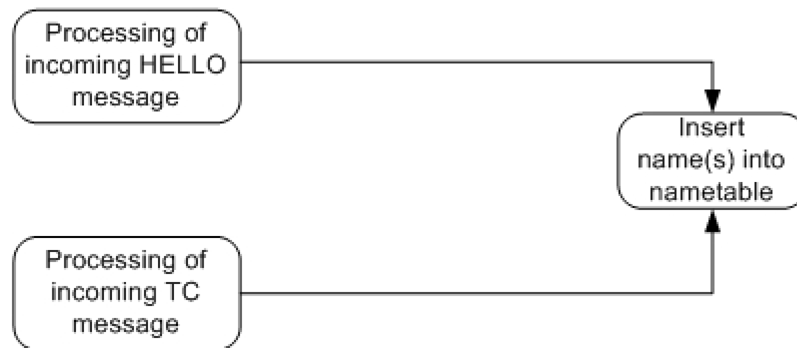

Figure 3.6: A shared data structure to store names from HELLO messages and TC messages.

Figure 3.6 shows us how we can store names in memory of a node without going around the routing table. The name table is populated under parsing of a HELLO message or a TC message. The name table records tuples with the following information:

```
(IP address, Name, Validity time)
```

`IP address` is the IP address that corresponds to the `Name` and vice versa. `Validity time` is a timer which decides when the tuple's information expires and is considered invalid. The tuple should then be removed from the name table. The purpose of having a validity timer for names is that we never can guarantee for how stable the topology is. A validity timer ensures that we always have a certain freshness of the information.

The name table can technically be created using a linked list. Different functions for inserting, deleting, lookups etc. must be created.

3.4 Generating and parsing HELLO messages and TC messages

Generation of HELLO messages and TC messages is done on a fixed time interval. The parsing technique used in the implementation of OLSR, OLSRd [12], uses a common parsing function for all messages that are being received on the socket. The message type is extracted out of the OLSR header. The message is then forwarded internally to the corresponding parsing function of that control message type.

In this section we focus on the naming of the nodes. This is an important aspect since it is a policy of how we identify nodes by names. Then we talk about the changes in the algorithms of generation and parsing of the HELLO messages and TC messages.

3.4.1 Naming

We described in Section 1.1.2 that the DNS naming convention very much reflects the organisational structure of a domain. MANETs are characterized by a flat structure where there is no hierarchy. We therefore meet problems if we use the same name convention as DNS does in MANETs.

Internet is grouped in blocks of IP addresses that are reserved for an organisation. These blocks are mostly connected to a domain and sub domains. This is a structure that MANETs never have. Nevertheless, there can be a policy that all nodes in a MANET in Oslo should

have predefined domain suffix. An example of such naming can be: `hostname.oslo-manet.no`. Again this way of naming requires that the `oslo-manet.no` domain is a reserved domain for MANET users. This raises the question about security. There must be restrictions of when you are allowed to use a domain suffix. In Oslo there might be one domain suffix, and in Bergen another one. Anyway, there is no guarantee, that the user of the nodes follows these rules by them selves. Automatic configuration of the domain suffix parameter should be configured automatically, to ensure that nodes always belong to the correct domain suffix. Such automatic configuration is though difficult to make work in MANETs. *Dynamic Host Control Protocol* (DHCP) servers are used for configuration like this in Internet. We then face the problem with the client-server relationship, which can not work on MANETs.

To solve this problem, is out of scope of this thesis. It is though interesting to see that existing naming convention is not as intuitive to include in MANETs as it looks. When it comes to naming nodes in this thesis, we keep our selves to that MANETs have a flat structure and that domain suffixes have no meaning. None of the nodes are organised in a hierarchical manner. A node's hostname is therefore used for naming the nodes. We must therefore assume that all nodes have unique names. This is also an assumption in DNS [1], since the naming convention there is `<hostname>.<domain suffix>`.

Later in this report, we test our solution in NEMAN [14], which is a program for emulation MANETs. Using only the hostname as the name of a node cause problems, since all the emulated nodes get the same name, since they physically run on the same node.

There is one way of making a difference in their name. Each emulated interface are assigned an own interface. Including the name together with the hostname causes the names to be unique in our emulation test bed.

The name identifier for each node is therefore defined as: `<hostname>-<interface-name>`.

3.4.2 Generation and parsing of HELLO messages

The HELLO messages are generated on a fixed time interval. Default emission interval for HELLO messages is two seconds. All headers that existed before will coexist with our extension, as discussed in Section 3.2. In Section 3.4.1 we decided that the name should be the hostname of a node in addition to the name of the interface. The hostname has been configured by the user and can be extracted with the `gethostname()` function in C, while the name of the interface can be extracted from the OLSRd.

The already implemented version of OLSR, OLSRd [12], uses two types of HELLO message structures: One type for use internally at a node, and one packet structure which is in use right before sending the packet of the HELLO message. This is more detailed discussed in Section 4.2.2. After all, both of these structures need extensions to include the name of the originator. The internal structure might only have a buffer, since it is only used for internal processing. In the packet, we must dynamically allocate the size it needs in the header for the name in the HELLO message. The name is stored in the name field together with its name length as we saw in Figure 3.2. All functions that work on the internal structure and the packet structure before sending of a HELLO message must therefore be modified.

When it comes to parsing of HELLO messages, the function that parses HELLO messages needs to be extended. This is because of the new header where the originator's name is stored. The name length is extracted, and the name is copied from the given position to the name length's number of bytes from the input buffer. Next, the name extracted is inserted into the name table.

As a summary we now know that the existing algorithms are still used for the generation and the parsing. They are just extended or modified in order to store and read the names in a HELLO message.

3.4.3 Generation and parsing of TC messages

As discussed in Section 3.4.1, the name of a node is set to be `<hostname>-<interface>` to make it work in NEMAN. This rule must also be followed in TC messages.

TC messages are generated on a time interval. The default time interval is every fifth second. We have already mentioned that default behaviour is that a node does not send a TC message, if it has not been chosen as an MPR node. This is though configurable in OLSRd. A node can be set to include either just the MPR selectors (default), both MPR selectors and MPR nodes or all of the neighbours. The last option results in more redundancy but also much more overhead on the network. The recommended behaviour is that the `TcRedundancy` in `olsrd.conf` that set this option should be kept on the default value which is `0`.

The originator's name is stored in the header together with its name length in the name field. When the generation algorithm adds all the neighbours, names to them are also added as well. They are stored in the message body as show in Figure 3.2.

The parsing is done likewise. First the originator's name is extracted. Then, in addition to the neighbours IP addresses their names are also extracted. Later, when all header values and names are extracted out from the message, all the names found in the message are inserted into the name table.

The generation and parsing algorithm of TC messages are extended to store and read names. The extended parts of the algorithm do not affect the existing storing and extraction parts of the algorithm.

3.5 Name resolution

The sections in this chapter have until now only focused on how the names should be distributed and stored in memory on each node. The next step is to make the names exchanged from HELLO messages and TC messages useful for name resolution.

In Section 3.1 one of the design requirements is that the service is application layer transparent. This because that software should avoid being special designed for use on MANETs. Software that work in the Internet, should also work on MANETs.

In the Internet we have DNS servers that maintain database files over the nodes which the server is responsible for. The DNS service is based on a client-server relationship which does not work in a MANET as already mentioned in Section 1.1.3.

In this design chapter we have been discussing how all nodes can know the names of all of the other nodes. We then have the opportunity to make all nodes acting as a name server.

Each time an application perform a DNS request, an operating system function is called. In a Unix/Linux environment this function is `gethostbyname()`. This function first parses the file `/etc/hosts` to try to find requested IP address that correspond to the name. If there was no match, the function tries to find a name server configured in `/etc/resolv.conf` and forward the request to that server. In Windows environment we also have a similar file. The path to that file is `C:\WINDOWS\system32\drivers\etc\hosts` if `C:` is the drive that Windows is installed on.

To serve name resolution it is enough to only perform maintenance on this hosts file and store all information from the name table there. Any modifications to the `/etc/resolv.conf` file are unnecessary.

Figure 3.7: A local name resolution with a DNS server running on a local node.

An alternative solution, as shown in Figure 3.7, is that we install and run a name server such as BIND [15] on each node. This is perhaps a more robust solution, but requires more configuration before our name service can be used. Regardless, the focus on this thesis is how we can distribute the names. We therefore choose to perform the easiest solution and perform maintenance on the *hosts* file in the operating system. The solution can be visualized as it is done in Figure 3.8.

Figure 3.8: A local name resolution with the *hosts* file on a local node.

The *hosts* file in both Unix/Linux and Windows platforms has this structure:

```
<IP address>      <Corresponding name>    <An eventual alias>
```

As discussed in Section 3.4.1 the names are only the hostname without any domain suffix. The structure of our hosts file will therefore be:

```
<IP address>      <Corresponding name>
```

Aliases are often just the hostname, and used to create shorter names out of the fully qualified domain name of a specified node.

In order to make maintenance of this hosts file, a new timer entry must be registered into the scheduler of OLSRd. This timer function is called every time the scheduler starts a new loop in the main loop of the scheduler. As a result we can therefore guarantee that we have a well updated hosts file. The timer function empties the file each time it is called by the scheduler, and rewritten, based on the current information stored in the name table.

This solution serves us an application transparent solution. The applications do not have to care about where the names are coming from. They just perform DNS requests as they use to do in the Internet.

3.6 Summary

In this chapter we have discussed the design of a fully distributed name service to nodes in a MANET. This is done by extending the TC messages and HELLO messages to include names. A new data structure, a name table, is created in OLSRd to store bindings between names and IP addresses. This is updated upon receiving HELLO messages and TC messages, and makes names fast to detect between neighbours. The generation and parsing techniques also have to be extended. Last, we need a way to make the information in the name table available to the applications running on the node. This is done by maintaining the `hosts` file that is located in `/etc/` in Unix/Linux environments.

4 Implementation

This chapter discusses the implementation details that are needed to be done in OLSR in order to distribute names between nodes. This implementation part consists of analysing the dataflow of the existing OLSRd implementation. The version used of OLSRd is 0.4.10. After this analyse is done, we know where to implement the extensions that we explained in the design chapter. We then look at, and discuss how the implementation is done. At the end we test our implementation in the NEMAN emulation environment.

4.1 OLSRd dataflow

In Section 3.4.2 and 3.4.3 of the design chapter, we talked about which elements that needed to be modified in order to reach our goal. To find out exactly which elements that needs to be modified we look at the dataflow that the OLSRd implementation has.

4.1.1 Dataflow of generating HELLO messages

The dataflow for generating HELLO messages is illustrated in Figure 4.1.

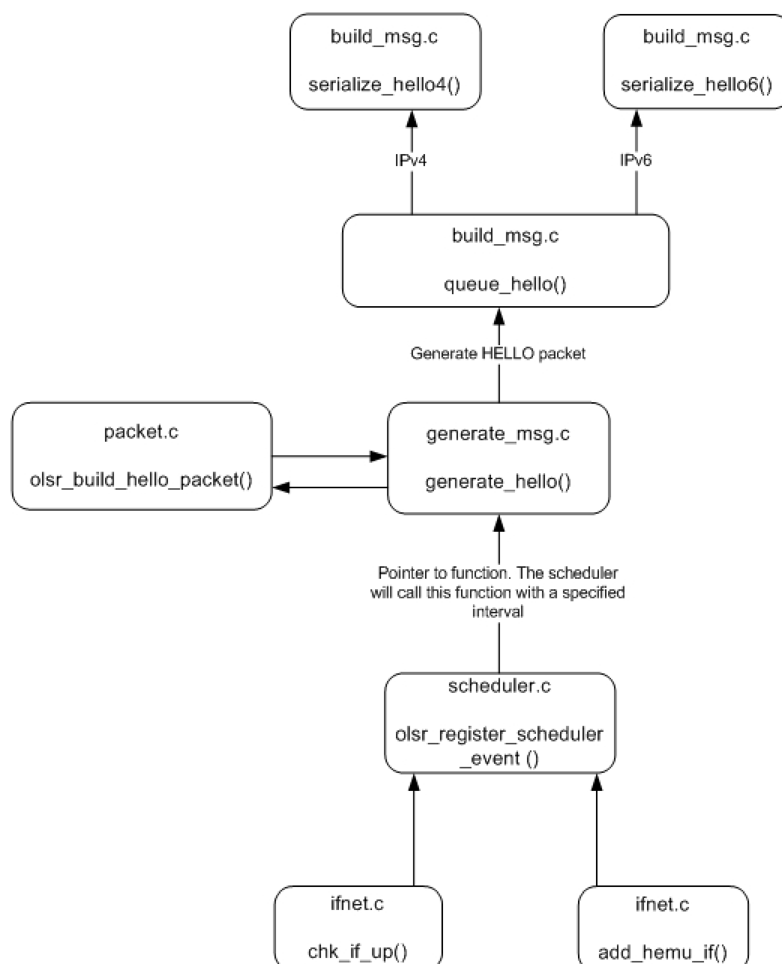

Figure 4.1: Dataflow for HELLO message generation. If an interface is going to be initialized with OLSR, a scheduler event is registered on that new interface. This event is for HELLO message generation, and is triggered on a fixed time interval.

From Figure 4.1 we see that if a new interface is going to be initialised in OLSRd, a scheduler event is registered on that interface. This procedure is done when OLSRd is started. A scheduler event is triggered for generating HELLO messages on a fixed time interval of every 2nd second. The scheduler function `olsr_register_scheduler_event()` registers a pointer to the `generate_hello()` function. In the main loop of the scheduler, the timer of the HELLO message event is checked. If the timer of two seconds has expired, the `generate_hello()` function is called.

As mentioned in the design chapter, the implementation of OLSRd uses two types of a HELLO message:

- An *internal structure*.
- A *packet structure*.

The internal structure of the HELLO message is used for calculation of header values and message body. In addition the internal structure is used when processing a HELLO message. The packet structure is the structure of the HELLO message that is sent over the network. This structure is encapsulated into an OLSR packet structure and takes into consideration correct byte order of header values and message body values. It also has the responsibility to store header members and message body members in correct order, so the receiver can parse the packet after it has been received. The internal structure has a more intuitive outline, while the packet structure is a more complex version of the HELLO message. Therefore, all processing on a HELLO message is done on an internal structure.

The `generate_hello()` function only uses the internal structure of the HELLO message. `generate_hello()` calls `olsr_build_hello_packet()` to fill the internal HELLO message with correct values of header values and message body values. In a HELLO message the message body consists of IP addresses to neighbours with corresponding information like the link state (symmetric or asymmetric link).

Our extension of a HELLO message consists of only adding the originator's name into the header. `The olsr_build_hello_packet()` must therefore be modified as well as the internal structure of the HELLO message.

From Figure 4.1, we can see that after the internal HELLO message is filled with correct information, the message is forwarded to a function that converts the message to the packet format and sends it. These functions are `serialize_hello4()` and `serialize_hello6()`. Which of them that is called, depends on which IP version that is used. When using IPv4, `serialize_hello4()` is called, and when using IPv6 `serialize_hello6()` is called.

`serialize_hello4()` and `serialize_hello6()` must also be modified and/or extended to store correct information into the HELLO message. The packet structure of the HELLO message must be extended as well, in order to be able to put the information about the originator's name and name length as discussed in Section 3.2.2.

We have now analysed which of the functions that need modifications or extensions with respect to the generation of HELLO messages. In Section 4.2.2 we describe how the implementation is done.

4.1.2 Dataflow of parsing HELLO messages

Parsing of HELLO messages is done as illustrated in Figure 4.2.

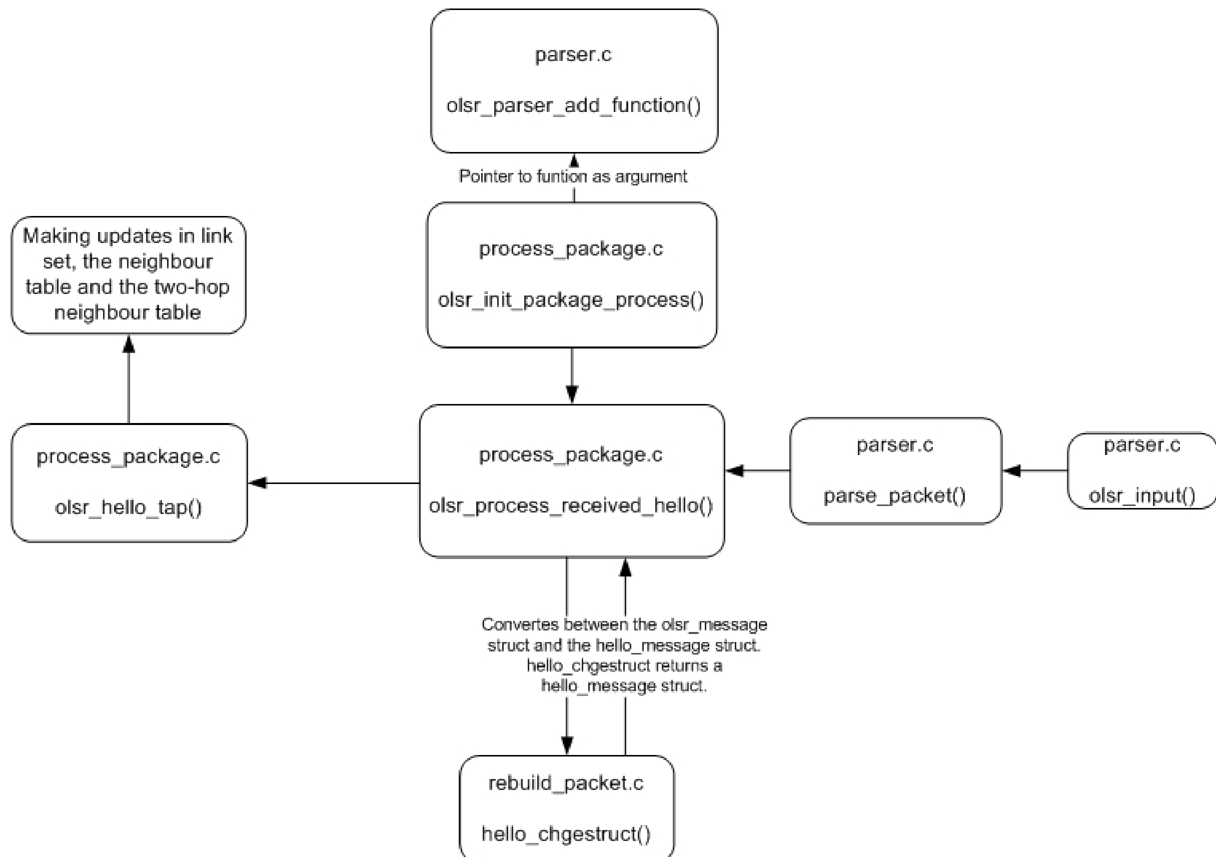

Figure 4.2: Dataflow when parsing a HELLO message.

When OLSRd starts, a parser functions to all known message types registered. This is also done for HELLO messages. The function `olsr_process_received_hello()` is registered as function for parsing of HELLO messages. When a HELLO message is detected as the incoming message type through `olsr_input()` and `parse_packet()`, `olsr_process_received_hello()` is called, and the whole message is forwarded to that function. The first task this function has, is to convert the packet version of the HELLO message back to the internal message format. This is done in `hello_chgestruct()`. We need to do some modifications in this function to parse the new message format correct. Neither the name length nor the name is extracted from the message without any modifications. Parsing of other header values also fails, since we redesign the HELLO message as mentioned in Section 3.1 and 4.1.1.

After the process of parsing and converting from packet format to internal HELLO message format is done, the HELLO message is ready for processing. This is done in `olsr_hello_tap()`. Since this function does not take care of our name extension, we need to modify this to store the name as discussed in Section 3.3. This should not be a comprehensive extension. It should only imply calling a function for the new data structure, the name table, for inserting or updating an entry. This is further discussed and shown how is implemented in Section 4.2.3.

4.1.3 Dataflow of generating TC messages

The dataflow of generating TC messages can be illustrated as it is done in Figure 4.3.

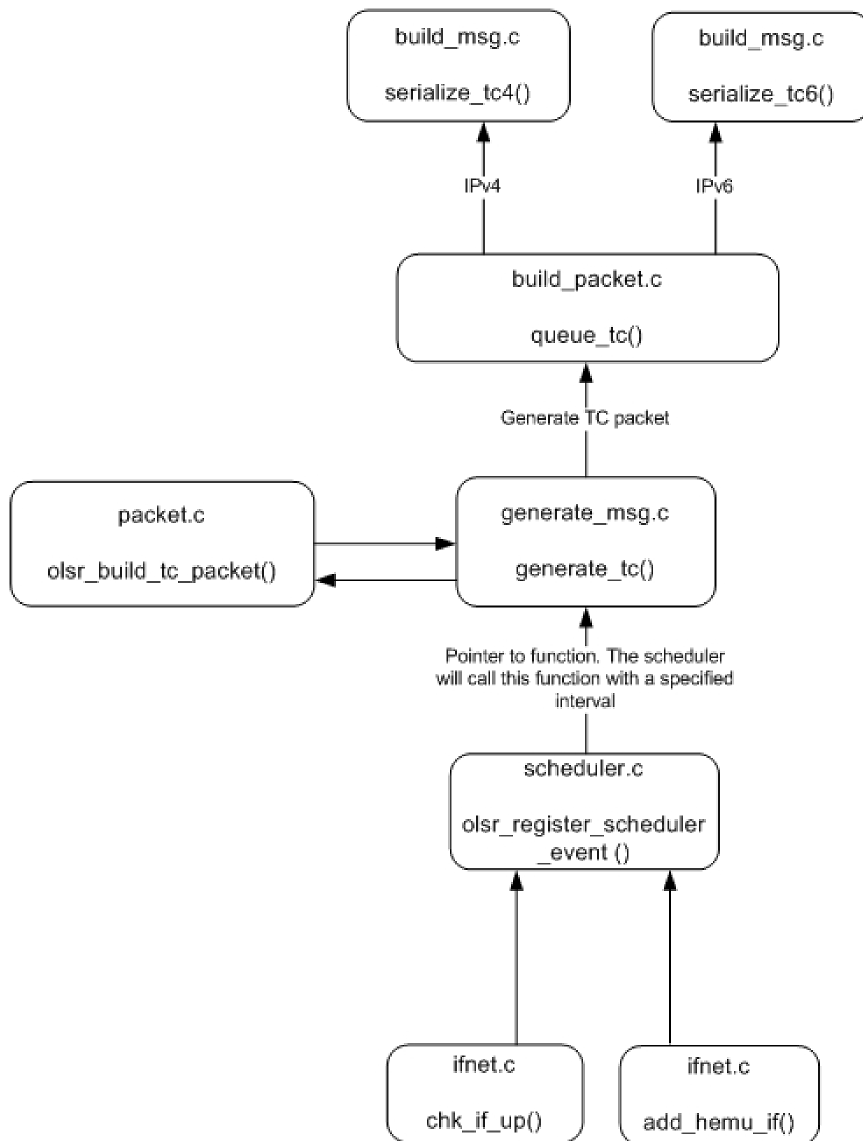

Figure 4.3: Dataflow of generating TC messages.

As an initial procedure, OLSRd registers a scheduler event to generate TC messages on each interface. A TC message is generated on a fixed time interval of every fifth second.

OLSRd uses two types of a TC message:

- An *internal structure*.
- A *packet structure*.

Their purposes are the same as for the HELLO messages as we described in Section 4.1.1.

The function that is called from the scheduler each time this timer expires, is the `generate_tc()` function. It then calls `olsr_build_tc_packet()` to create an internal structure of the TC message.

There are three modifications that we need to do: First, the internal structure needs to be extended to include the originator's name. Second, the internal structure needs to be extended, so that every announced neighbour's main address also includes their corresponding names. Third, the `olsr_build_tc_message()` needs to be extended to store the names in the extended internal structures of the TC message.

A TC message contains information about all necessary names of the originator and all MPR selectors/neighbours that is to be announced. When an internal TC message is filled up with correct information, `olsr_build_tc_message()` calls `queue_tc()` which again calls `serialize_tc4()` or `serialize_tc6()`. `serialize_tc4()` is called when IPv4 addresses are used and `serialize_tc6()` is called when IPv6 addresses are used. These two functions build and send the TC message in the external format. We need here to modify and or extend the `serialize_tc4()` and `serialize_tc6()` functions as well as the packet format of the TC message.

4.1.4 Dataflow of parsing TC messages

The dataflow of parsing incoming TC messages can be illustrated as in Figure 4.4.

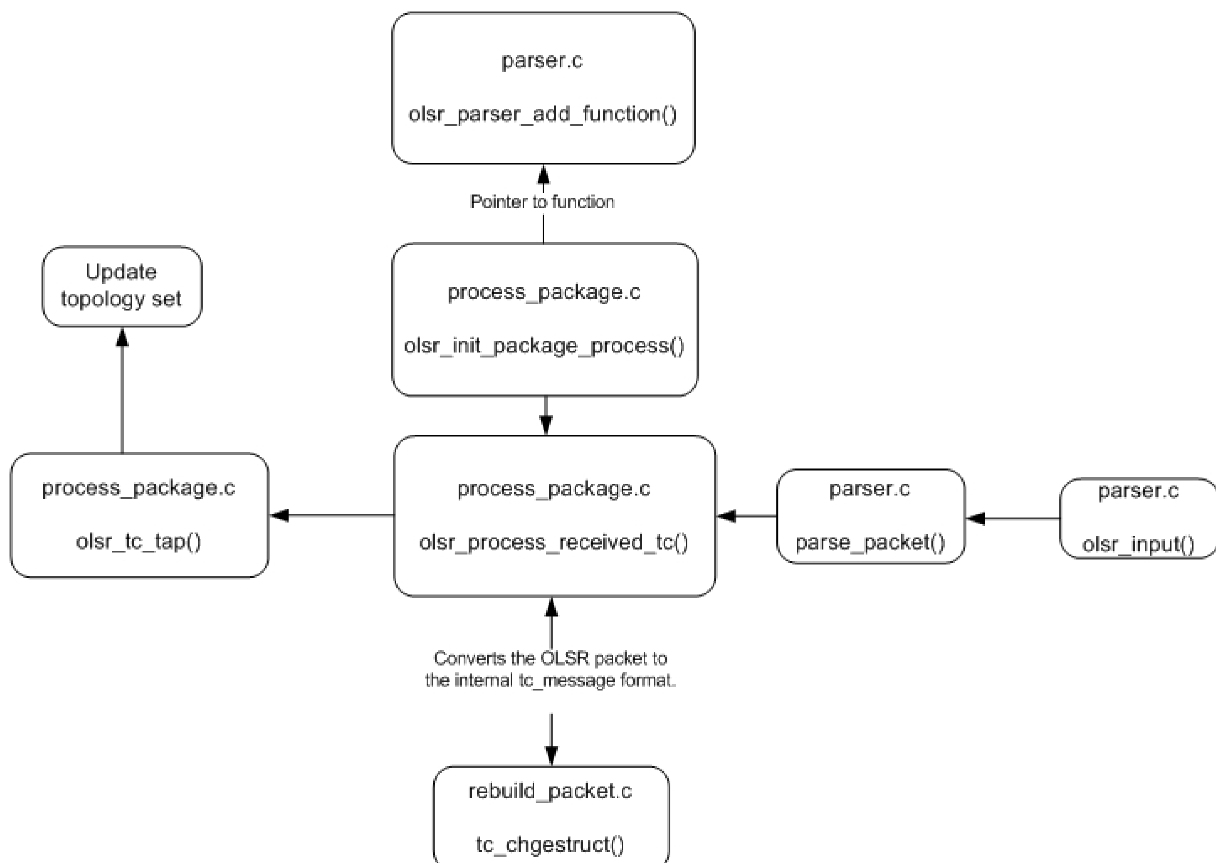

Figure 4.4: Dataflow of parsing TC messages.

When OLSRd starts, `olsr_init_package_process()` is called to add parser functions for all known message types. This is also the case when coming to TC messages. `olsr_init_package_process()` calls `olsr_parser_add_function()` with `olsr_process_received_tc()` as an argument. The parser of all incoming messages is then

registering a new parser function for the TC message type. When a TC message is detected during parsing, `olsr_process_received_tc()` is called for further processing. We do not need to do any modifications here.

When `olsr_process_received_tc()` is called from the `parse_packet()`, `olsr_process_received_tc()` converts the received external version of the TC message format to the internal message format. This is done in `tc_chgestruct()`. We need modification and/or extensions in order to read all new members of the external TC message. The internal TC message is already changed as discussed in Section 4.1.3.

When the message has been converted back to the internal format, olsr_process_received_tc() calls olsr_tc_tap() for further processing. olsr_tc_tap() will update the topology set with new known destinations, or update the timer on the existing ones. We must here also insert or update our name table in order to keep track of new or known names. As discussed in Section 3.3 the name table entry consists of a timer that will expire. We want to remove nodes that do not exist in the MANET due to frequent changes of the topology.

4.1.5 Summary code analysis

We now have pointed out which elements that we need to extend and/or modify. In Section 4.2, we will look at how we can implement these extensions. As a summary from this section, some tables are presented in order easily to get an overview of which elements that we need to modify or extend.

Elements regarding HELLO messages:

Functions	Located in:
`olsr_build_hello_packet()`	`packet.c`
`serialize_hello4()`	`build_msg.c`
`serialize_hello6()`	`build_msg.c`
`hello_chgestruct()`	`rebuild_packet.c`
`olsr_hello_tap()`	`process_packet.c`

Table 4.1: Functions that need modifications and/or extensions regarding HELLO messages.

Data structures	Located in
`struct hello_message` (internal HELLO message)	`packet.h`
`struct hellomsg` (packet structure of the IPv4 HELLO message)	`olsr_protocol.h`
`struct hellomsg6` (packet structure of the IPv6 HELLO message)	`olsr_protocol.h`
`struct hellomsg` (packet structure of the HELLO message)	`olsr_protocol.h`

Table 4.2: Data structures that need modifications and/or extensions regarding HELLO messages.

Elements regarding TC messages:

Functions	**Located in:**
olsr_build_tc_packet()	packet.c
serialize_tc4()	build_msg.c
serialize_tc6()	build_msg.c
tc_chgestruct()	rebuild_packet.c
olsr_tc_tap()	process_packet.c

Table 4.3: Functions that need modifications and/or extensions regarding TC messages.

Data structures	**Located in**
struct tc_messsage (internal TC message)	packet.h
struct tc_mpr_addr (list of MPR selectors/neighbours in the internal TC message)	packet.h
struct tcmsg (packet structure of the IPv4 TC message)	olsr_protocol.h
struct neigh_info (list of MPR selectors/neighbours in the packet structure of the IPv4 TC message	olsr_protocol.h
struct tcmsg6 (packet structure of the IPv6 TC message)	olsr_protocol.h
struct neigh_info6 (list of MPR selectors in the packet structure of the IPv6 TC message)	olsr_protocol.h

Table 4.4: Data structures that need modifications and/or extensions regarding TC messages.

4.2 Implementation of the extensions

This section describes how the implementation is done of the elements pointed out in Section 4.1.5. The code shown is only an abstract of the whole code, since it will most likely confuse the reader if the whole code is shown here together with the explanation. The whole code can be read in the CD contents described in Appendix A. When the code differs regarding to the different IP versions, only IPv4 code is shown. This is to limit the code here and is an agent to get a nice overview in this section.

When new code is presented, all extensions are commented as "`Extended by Øyvind`" or "`Extension`".

4.2.1 Implementing the name table

As discussed in Section 3.3, we need a dedicated data structure to store all names received from within HELLO messages and TC messages. This data structure is hereby called the name table. Our design of the name table was that it will record tuples with these fields:

```
(IP address, Name, Validity time)
```

Since we want to include the extensions that our name service requires into existing code as much as possible, we choose to include functions and declarations of the name table into an existing C source file. The name service explained in this thesis is from the writer's point of view more related to TC messages than the HELLO messages. We therefore choose to add the name table into where all information about the TC messages is stored. This is in the file where the topology set is implemented, located in the `tc_set.h` and `tc_set.c` files. All extensions that are related to the name table is therefore written in these two files.

The C structure of the name table is:

```
struct name_entry
{
      union olsr_ip_addr destination;
      char name[255];
      clock_t            val_time;
      struct name_entry *next;
};
```

The IP address that correspond to the name vice versa, are two members of this structure. Third member of this structure is the timer that tells when the validity of an entry expires and must be removed. Fourth member is a next pointer, which indicates that the data structure of the name table is a linked list. Table 4.5 describes how and where the name table is declared.

Data structures	**Located in**
`struct name_entry`	`tc_set.h`
`extern struct name_entry* nametable` (The name table)	`tc_set.h`

Table 4.5: The new name table is an extended data structure of OLSR.

We need functions to manipulate the name table. These functions must handle initialisation of the name table, inserting of new entries, deleting of expired entries, lookup of entries, printing out information to standard output for debugging purposes and writing of a hosts file as described in Section 3.5. Functions that corresponds to these functions and that will be created are presented in Table 4.6.

Functions	**Located in:**
`olsr_init_nametable()`	`tc_set.c`
`insert_name_entry ()`	`tc_set.c`
`delete_name_entry ()`	`tc_set.c`
`find_name_entry()`	`tc_set.c`
`write_hostsfile()`	`tc_set.c`
`time_out_nametable ()`	`tc_set.c`

Table 4.6: Functions that is needed for the nametable.

The first function `olsr_init_nametable()` has the responsibility to initialise the name table. A timeout function is registered in the scheduler from `olsr_init_nametable()`. This timeout function, named `time_out_nametable()`, has the responsibility to iterate through the name table and delete expired entries by calling `delete_name_entry()`. It also has the responsibility to write the hosts file by calling `write_hostsfile()`. `write_hostsfile()` might not be in use under all tests of the implementation. This is discussed in Section 4.3.1.2. `insert_name_entry()` does as the name says. It inserts entries into the name table if no such

entry exists identified by the IP address. This function also updates timers on existing entries. To find existing entries, the `find_name_entry()` function is used.

4.2.2 Extending HELLO messages with names

In Section 4.1.1 and 4.1.5 we pointed out the functions and the data structures that need to be modified or extended for generating our new design of HELLO messages.

The first step in our implementation is to extend the internal HELLO message structure located in `packet.h`. This is done as follows (The extensions are commented with "`Extension by Øyvind`"):

```
struct hello_message
{
  double                 vtime;
  double                 htime;
  union olsr_ip_addr     source_addr;
  olsr_u16_t             packet_seq_number;
  olsr_u8_t              hop_count;
  olsr_u8_t              ttl;
  olsr_u8_t              willingness;
  struct hello_neighbor  *neighbors;
  char name[MAXHOSTNAMELENGTH]; /* Extension by Øyvind */
};
```

The change to the internal structure is minimal. Only the originator's name is additionally stored here. The `MAXHOSTNAMELENGTH` macro is also a new extension and defined in `olsr_protocol.h`. This macro is set to 255.

The next step is to store the hostname into the internal message when `olsr_build_hello_message()` is called. This function is called every time a new HELLO message is created to create the HELLO message out of a totally empty internal `hello_message` structure:

```
/* Extension by Øyvind - Ask the OS for the hostname and store it in
 * the TC message.Max hostnamelength is 255 bytes
 */
 gethostname(message->name, MAXHOSTNAMELENGTH);
 /* End of extension */
```

With these changes the name is stored in the internal structure of a HELLO message. `gethostname()` is a build-in function in C that asks the operating system after the current configured hostname. The maximum number of characters read from the hostname returned by the operating system is 255, which is the value of `MAXHOSTNAMELENGTH`. The function stores the name into name member of the `hello_message` structure.

The next step is to build the packet version of the HELLO message. As pointed out in Section 4.1.5, that code is located in the `serialize_hello4()` and `serialize_hello6()` functions. The different functions are called dependent on what IP version the interface is used.

Before we start implementing the extensions of `serialize_hello4()` and `serialize_hello6()`we need to do some changes with the packet structure of the HELLO

message. This structure is located in olsr_protocol.h. The modifications are done as shown below:

```
struct hellomsg
{
  olsr_u16_t      reserved;
  olsr_u8_t       htime;
  olsr_u8_t       willingness;
  olsr_u8_t name_len; //Extension
  struct hellinfo hell_info[1];
} __attribute__ ((packed));
```

It is important to notice that the name itself is not stored in this structure. Only the byte that contains the name length is stored here. Since the structures have the attribute "packed" the structure is stored in memory such that all structure members are aligned after each other in memory. In our case we want a char pointer that can have dynamic length that depends on the name. Though, a structure that has the "packed" attribute, a char pointer will only refer to one byte in memory. If we try to write more than one byte, other information in the structure/message will be overwritten. When we build the message on the output buffer, we therefore need to add the name manually at the end of the message. Even if the name field of the originator is added at the end of the message, this field is still considered as a header value as in Figure 3.4.

If we read olsrd.conf located under /etc/ on Unix/Linux platforms, we can locate a variable named TcRedundancy also mentioned in Section 3.4.3. This variable tells us what kind of neighbours that we distribute through TC messages. The default value is 0, which means that only the MPR selectors are announced. MPR selectors are as mentioned in Section 2.3.5.6, neighbour(s) of a node that has/have chosen this node as a MPR node. This is one part of the optimizations regarding to overhead in OLSR compared to other link state routing algorithms.

We shortly summarize the other values of the TcRedundancy option of OLSRd: If TcRedundancy is 1, both MPR selectors and MPR nodes are announced as neighbours in the TC message. Last, if TcRedundancy is 2, all neighbours are announced in the TC message. This means that whenever a neighbour is detected, a TC message will be generated. Hence, we do not add names into HELLO messages if TcRedundancy is 2 to minimize overhead, as discussed in Section 3.2.3. If TcRedundancy is either 0 or 1, names must be stored into HELLO messages in order to distribute the names to all nodes on the MANET.

This analysed problem is taken into consideration when building HELLO messages. Hence, a Boolean variable is created at the top of build_msg.c:

```
/* Shared variable which is used by the serialize_hello[4/6] and
 * serialize_tc[4/6] function to decide wheter or not to put the
 * name in the HELLO messages. Extension by Øyvind
 */
static olsr_bool mpr_found;
```

This variable is set to true, if and only if the node has begun to distribute TC messages in serialize_tc[4/6](). If TcRedundancy is 2 or mpr_found is true, we do not add the

name into the HELLO message, since it will be distributed through TC messages whenever a neighbour is detected:

```
/* Extension by Øyvind
 * The name is also sent in HELLO messages if TC Redundancy is less
 * than 2. If the TC Redundancy is 2 or that this node has generated
 * a TC message with data the name in HELLO messages is not added.
 * This to reduce overhead.
 */

if( olsr_cnf->tc_redundancy == 2 || mpr_found == OLSR_TRUE )
{
      h->name_len = 0;
      /* Even if we not need to send the name, the byte
       * which indicates the length is still in the header.
       */
      curr_size++;
}
else
{
      namelen = strlen(message->name);
      message->name[namelen] = '-';
      strcpy(&message->name[namelen+1], ifp->int_name);
      h->name_len = strlen(message->name);
      curr_size += h->name_len+1;
  }
  /*End of extension*/
```

As we can see from the code, the `name_len` member of the external HELLO message structure always exists. This implies that the message size always is one byte greater than the original HELLO message at this point. If the name is stored into the HELLO message, the message size will increase with the size of the name plus this `name_len` byte.

In Section 3.4.1 we decided to change the naming convention to be like this:

`<hostname>-<interface-name>.`

This is handled in the code above after the `else` statement.

As mentioned, we had to add the name manually at the end of the message. This is done right before the HELLO message is sent by calling `net_outbuffer_push()`. We therefore have to get the pointer in the output buffer of the current position.

The `curr_size` variable keeps track of this position all the time. Whenever new data is added to the message, this variable increases. We can see that the name length is already added to the `curr_size` variable at this point. This is to define an empty message. As we can see from the whole code in Appendix A, a partial packet is only sent if the message contains data. An empty message is in addition to the original HELLO message header also including the `name_len` member of the packet structure.

The correct position at the end of the message is therefore `curr_size` minus the name length. The `name_len` byte is added before the message body, so that byte must not be taken into consideration here.

If `TcRedundancy` is less than two and the name length is greater than zero, we add the name behind the external HELLO message into the message buffer as follows:

```
/* Extension by Øyvind. We shall add the name if the tc_redundancy
 * is less than 2. We do not add the name if h->name_len has the
 * value 0. This means that mpr_found is FALSE or that we simply
 * do not have any name to add.
 */
if( olsr_cnf->tc_redundancy < 2 && h->name_len > 0)
{
      name = (char *)&msg_buffer[curr_size-h->name_len];
      strncpy(name, message->name, h->name_len);
}
/* End of extension */
```

The function `serialize_hello6()` is modified similarly, but not shown here because of the reasons stated in the beginning of Section 4.2.

4.2.3 Parsing the new HELLO message

Parsing of a message received from the network implies to read it in a predefined way. The structure of the message must be known in order to do the parsing correct. In Section 4.2.2 we changed the external structure of the HELLO message that is sent between neighbours on a MANET, as well as the internal message format.

In `hello_chgestruct()` the HELLO message is converted back to the internal format, which implies modification and extension of this function. First we declare a new char pointer at the top of this function:

```
char *name; //Extension by Øyvind
```

This pointer is to be set at the position at the end of the message, minus the value of the `name_len` member of the packet structure of the HELLO message. We can then easily copy the name from the input buffer and into the internal HELLO message at the receiver side:

```
/*Extension by Øyvind - Extract the name in the HELLO message */
if( m->v4.message.hello.name_len > 0 )
{
      name = (char *)m + (ntohs(m->v4.olsr_msgsize) -
                  m->v4.message.hello.name_len);
      strncpy(hmsg->name, name, m->v4.message.hello.name_len);
      hmsg->name[m->v4.message.hello.name_len] = '\0';
}
else
{
      hmsg->name[0] = '\0';
}
/* End of extension */
```

Since the name is stored at the end of the message, the message body does not last to message size which is the end of the message. We must therefore alter the for-loop, to last from the message body position to the message size minus the length of the name:

```
/* Changed by Øyvind. The name from the originator node carried
 * in the HELLO message is stored after all of the neighbor
 * nodes. Must therefore substract the name length
 */
 for  (hinf = m->v4.message.hello.hell_info;
      (char *)hinf < ((char *)m +
      (ntohs(m->v4.olsr_msgsize))-m->v4.message.hello.name_len);
      hinf = (struct hellinfo *)((char *)hinf + ntohs(hinf->size)))
{

      for (hadr = (union olsr_ip_addr  *)&hinf->neigh_addr;
          (char *)hadr < (char *)hinf + ntohs(hinf->size);
           hadr = (union olsr_ip_addr *)&hadr->v6.s6_addr[4])
      {
            nb = olsr_malloc(sizeof (struct hello_neighbor),
                                                  "HELLO chgestruct");

            COPY_IP(&nb->address, hadr);

            /* Fetch link and status */
            nb->link = EXTRACT_LINK(hinf->link_code);
            nb->status = EXTRACT_STATUS(hinf->link_code);

            nb->next = hmsg->neighbors;
            hmsg->neighbors = nb;
      }
}
```

When the process of converting from packet format to internal message format is finished, the message will be processed. To HELLO messages this means to update the link set, the neighbour set, the two-hop neighbour set, the MPR set and the MPR selector set. This is all done with `olsr_hello_tap()`.

In `olsr_hello_tap()` we only need to add the functionality of updating the name table. This can easily be done with one single call on the `insert_name_entry()` available from the `tc_set.h` as declared in Table 4.6. The call is implemented like this:

```
/*Exension by Øyvind-Insert the name into the nametable*/
insert_name_entry(message->source_addr, message->name, message->vtime);
```

The IP address and the name of the originator of the HELLO message are taken as the first and the second argument. The third argument is the validity time. This is declared as an argument to `insert_name_entry()`, since HELLO messages and TC messages operate with different validity times. We also choose to use the same validity time from the received control message, since we must rely on that we get a new similar control message from the source node, as the one we newly have received.

By setting these rules we can always keep the name table updated as good as the routing table of the node. It is important that the name table and the routing table are synchronized in this way. Whenever a name in the name table is recorded, there should be a route in the routing

table to the same host and vice versa. Our claim about efficiency will then be fulfilled. We will get a binding between the IP address and the names as fast as possible.

4.2.4 Extending TC messages with names

Modification of TC messages and HELLO messages is done in nearly the same way. The difference is that in TC messages, we also want to include the announcement of the neighbours' names.

The internal structure of the TC messages is located in `packet.h` and is extended as follows:

```
struct tc_message
{
  double              vtime;
  union olsr_ip_addr  source_addr;
  union olsr_ip_addr  originator;
  char name[MAXHOSTNAMELENGTH]; //Extended by Øyvind
  olsr_u16_t          packet_seq_number;
  olsr_u8_t           hop_count;
  olsr_u8_t           ttl;
  olsr_u16_t          ansn;
  struct tc_mpr_addr  *multipoint_relay_selector_address;
};
```

The `tc_message` structure is extended only with the originator's name.

The last member of this structure has a pointer to a `tc_mpr_addr` structure. Here all neighbours that are to be announced in the TC message are stored. The `tc_mpr_addr` pointer can be considered as the TC message body as illustrated in Figure 3.2.

Next task is to extend the neighbour information. We include the names for each of them. This is done in this way:

```
struct tc_mpr_addr
{
  double             link_quality;
  double             neigh_link_quality;
  union olsr_ip_addr address;
  char name[MAXHOSTNAMELENGTH]; //Extended by Øyvind
  struct tc_mpr_addr *next;
};
```

`MAXHOSTNAMELENGTH` is an extended macro defined to 255 bytes. A name has like HELLO messages a maximum length of 255 bytes.

As mentioned a timer is registered in the scheduler to generate TC messages. When the timer expires, `generate_tc()` is called, as seen in Figure 4.3. `generate_tc()` calls `olsr_build_tc_message()` to build an empty internal version of the TC message out of an empty one. `olsr_build_tc_message()` first adds all header values as it did originally. Then it inserts the originator's name:

```
/* Extension by Øyvind - Ask the OS for the hostname and store
 * it in the TC message. Max hostnamelength is 255 bytes
```

```
 */
gethostname(message->name, MAXHOSTNAMELENGTH);
/* End of extension */
```

This is done in the same way as with HELLO messages in Section 4.2.2. `gethostname()` is a build-in function in C that asks the operating system after the current configured hostname. The maximum number of characters read from the hostname returned by the operating system is 255, which is the value of `MAXHOSTNAMELENGTH`. The function stores the name into the name member of the `tc_message` structure.

We now want to insert all of the neighbours in the TC message. As we can see from the next abstract of the code in `olsr_build_tc_message()`, is that the number of neighbours depends on the value of `TcRedundancy` in `olsrd.conf` (see also Section 3.4.2 and Section 4.2.2).

In the for-loop that iterates through all neighbours there is a switch-case statement. This code illustrates how this is done:

```
switch(olsr_cnf->tc_redundancy)
{
	case(2):
	{
		/* 2 = Add all neighbors */
		break;
	}
	case(1):
	{
		/* 1 = Add all MPR selectors and selected MPRs */
	break;
	}
	default:
	{
		/* 0 = Add only MPR selectors(default) */
		break;
	}
}
```

If `TcRedundancy` is `2`, all neighbours are added. When `TcRedundancy` is `1`, `olsr_build_tc_message()` must perform a lookup in the MPR selector set. If the neighbour was not found there it performs a lookup into the MPR set. If not found in neither of these two sets, the loop starts over with the next neighbour, if any, without adding the current one. If `TcRedundancy` is `0`, which is default, only a lookup into the MPR selector set is performed. If the current neighbour was not found there, the loop starts over with the next neighbour, if any, without adding the current one.

In all of these cases, if the requirement for adding the neighbour into the TC message is fulfilled, we perform a lookup into the name table in order to find the neighbours' names.

Normally, the names of the neighbours' names are transported in HELLO messages. Though, this is not the case if `TcRedundancy` on a neighbour is set to `2`. The name in this scenario is transported through a TC message from that node as discussed in Section 3.2.3. A name should therefore be present in the name table, when coming to this point of generating TC messages.

Next, we perform a call on the function `find_name_entry()`, to find the name of the current neighbour in the loop. If the name is found, which is almost guaranteed every time, the name is added. Otherwise the string end terminator character is set at the beginning of the buffer:

```
/* Extension by Øyvind */
mpr_hostname = find_name_entry(&entry->neighbor_main_addr);
if(mpr_hostname != NULL)
{
      strcpy(message_mpr->name, mpr_hostname->name);
}
else
{
      message_mpr->name[0] = '\0';
}
/* End of extension */
```

The internal TC message is now created. We now discuss how the packet structure of the message must be extended, and how we store all nodes from the internal TC messages into the output buffer. The `neigh_info` structure can be considered as the message body where the neighbours are located. The `tcmsg` structure can be considered as the header of the TC message:

```
struct neigh_info
{
  olsr_u32_t       addr;
  olsr_u8_t mpr_name_len; //Extension
} __attribute__ ((packed));


struct tcmsg
{
  olsr_u16_t        ansn;
  olsr_u16_t        reserved;
  /* Extended by Øyvind. In order to carry name from the origintator
   * node in the TC message we need the length of it.
   *The name itself is hooked on TC message after all MPR nodes.
   */
  olsr_u8_t name_len;
  /* End of extension */
  struct neigh_info neigh[1];
} __attribute__ ((packed));
```

The packet structures that compose the message body and the header of the TC message are only extended with the name lengths. The name themselves must be put into the output buffer manually. This is the same problem we met when building HELLO messages in Section 4.1.1.

The "`packed`" attribute organizes the memory so that all members of the packet are stored right after each other. A char pointer always just allocate 1 byte in memory. The names therefore overwrite other header values or message body values. To easily integrate our extension of the TC message, we choose to store the originator's name at the end of the message. The originator's name is still considered as a header value as declared in Figure 3.2.

In this section we only present an abstract of the code. `serialize_tc4()` and `serialize_tc6()` can be seen in Appendix A. The source code shown is only the code related to IPv4.

The first thing that is done in `serialize_tc[4/6]()` is to insert all header values that are stored in the internal TC message and into the packet structure of the message. This also includes storing the name length into the header. The name itself is stored in the end of message, because of the problem we meet with the attribute "`packed`" on packet format of the TC message structure.

The current size of the message will be the TC message's IPv[4/6] header, plus the length of the host name, plus the byte that hold the name length. After this is added to the `curr_size` variable, this is the message length that is considered as an empty TC message. An empty TC message means a TC message without anything stored in the message body, which in TC messages are information about neighbour nodes.

```
/* Extension by Øyvind - Get the name of the hostename.
 *
 * Name length is the string length of the hostname.
 */
namelen = strlen(message->name);
message->name[namelen] = '-';
strcpy(&message->name[namelen+1], ifp->int_name);
tc->name_len = strlen(message->name);
/* Current size is now the TC IPv4 header + the string length + the
 * byte which tells us the length of the string. */
curr_size += tc->name_len+1;
/* End of extension */
```

The next step is the iteration through the announced neighbours. In the code, these neighbours are referred to as MPR selectors, since default behaviour is that only the MPR selectors are added.

The loop that iterates through the MPR selectors is modified quite much, as we can see from the code. All the names to all neighbours in the internal TC message must now be stored into the output message buffer. This requires modification of the pointers that manipulates the message output buffer.

First the IP address is added, and the current size of the message is increased by either 32 bits or 128 bits depending on which IP version that is used. An IPv4 address is always 32 bits long, while an IPv6 is always 128 bits long:

```
COPY_IP(&mprsaddr->addr, &mprs->address);
curr_size += ipsize;
```

The second step is to insert the length of the name to the current MPR selector node. The length of the name is stored in one byte. This is enough to represent the name length in decimal notation, since the maximum name length is 255 bytes. The current size of the message is then increased by one byte:

```
/* Extension and modifications by Øyvind */
mprsaddr->mpr_name_len = strlen(mprs->name);
curr_size += 1;
```

After the IP address and the name length of a neighbour, the name is stored. In order to manage this, the pointer to the output buffer must point at the memory address right after where the name length was stored. The pointer that now points at the output buffer is currently a pointer type of a `neigh_info` structure, and is called `mprsaddr`.

This structure has been extended with the byte to include the name length. The length of this structure is therefore now 5 bytes if we use IPv4 addresses and 17 bytes if we use IPv6 addresses.

In order to point at the correct position in the output buffer we increase this `neigh_info` pointer by one. It will then point at the position right after currently added neighbour's name and name length:

```
mprsaddr++;
```

Next, we want to add the name of the MPR selector. We create a char pointer that points to the location where `mprsaddr` now points to. This pointer is called `mpr_name`. The name from the internal TC message is copied into the output buffer where `mpr_name` points to.

Then this pointer must be moved to point at the location after the name. The `neigh_info` pointer, mprsaddr, is now set to point at the same location at the output buffer as the `mpr_name` does, which is right after the current MPR selector's name in the loop.

```
mpr_name = (char *)mprsaddr;
strncpy(mpr_name,mprs->name,strlen(mprs->name));
mpr_name += strlen(mprs->name);
curr_size += strlen(mprs->name);
mprsaddr = (struct neigh_info *)mpr_name;
/*End extension */
```

The last extension is to store the originator's name into the message. We have already discussed that the name will be stored at the end of the message, likewise as with HELLO messages in Section 4.2.2. This procedure is done right before the message is sent. The name length has already been added to the current size. Therefore the location where the name is to be stored is current size minus the name length:

```
name = (char *)&msg_buffer[curr_size-tc->name_len];
strncpy(name, message->name,tc->name_len);
```

These are the main extensions and modifications that are needed to be done in order to store all names with corresponding name lengths into the TC message that is to be sent. There are also some other minor modifications. These can be seen in the Appendix A.

4.2.5 Parsing the new TC messages

The structure of the TC message that is sent over the network has now changed. When an incoming message is identified as a TC message, `olsr_process_received_tc()` is called by `parse_packet()` as we can see from Figure 4.4.

`olsr_process_received_tc()` will first call `tc_chgestruct()` to convert from the packet format to the internal format, before any further processing. Since we have altered the design of the TC message, we need to do some extensions and modifications here to read the TC message correctly.

In the top of the `tc_chgestruct()` function, we declare some new variables we need further down in the function:

```
char *name, *mpr_name; //Extension by Øyvind
olsr_u8_t mpr_name_len; //Extension by Øyvind
```

The char pointers are set to positions where the names are stored in the input message buffer. The `name` pointer will be set to point at the location where the name of the originator is, while the `mpr_name` is set to point at the names of the announced neighbours in the TC message during the iteration of the message body. `mpr_name_len` is a temporary variable used for extracting the name length of the neighbours.

First the name of the originator is extracted. This is done similarly as with HELLO messages. The position of the name in the input message buffer is the messages size, minus the length of the name. We can now, after setting the correct position of this `name` pointer, easily copy it into the internal message format:

```
/*Extension by Øyvind */
name = (char *)m + (ntohs(m->v4.olsr_msgsize)-tc->name_len);
strncpy(tmsg->name, name, tc->name_len);
tmsg->name[tc->name_len] = '\0';
/* End of extension */
```

Next, a minor modification is done in the for-loop that iterates through the message body. Since we have added the originator's name at the end of the message, we must extend the for-loop to read the message including the message size plus the length of the name to the originator.

As illustrated in Figure 3.2, each announced neighbour is stored after each other in the body of the TC message. A neighbour is in our new design of a TC message represented with first the IP address, then a name field which consist of the name length and the name. In Section 4.2.4, we extended the `neigh_info` structure with the name length. After this byte, the name of the neighbour is stored. The `mpr_name` char pointer must therefore be updated every time the for-loop starts over to point at the correct position after `name_len` byte.

The whole for-loop will as a consequence from the two paragraphs above, be extended and modified quite much:

```
/* Changed by Øyvind. The name from the originator node carried in
 * the TC message is stored after all of the MPR nodes. Must
 * therefore substract the name length (-tc->name_len)
 */
for ( maddr = mprsaddr;
      (char *)maddr < ((char *)m +
                        (ntohs(m->v4.olsr_msgsize))-tc->name_len); )
{

      mprs = olsr_malloc(sizeof(struct tc_mpr_addr), "TC chgestruct");
      COPY_IP(&mprs->address, &maddr->addr);
```

```
        mpr_name_len = maddr->mpr_name_len;

        /* Extension by Øyvind */
        maddr++;
        mpr_name = (char *)maddr;
        strncpy(mprs->name, mpr_name, mpr_name_len);
        mprs->name[mpr_name_len] = '\0';
        mpr_name += mpr_name_len;
        maddr = (struct neigh_info*)mpr_name;
        /* End of extension */

        mprs->next = tmsg->multipoint_relay_selector_address;
        tmsg->multipoint_relay_selector_address = mprs;
}
```

The IP address and the `mpr_name_len` are extracted first from the `maddr` structure which is of the `neigh_info` type. The `maddr++` statement that originally was an increase statement of the for-loop is moved down to the body. The pointer is increased by one, which means that we move 5 bytes forward in the memory where the message input buffer is stored.

We move 5 bytes forward in memory if IPv4 addresses are used and 17 bytes with IPv6 addresses. This is the same technique used when building TC messages in Section 4.2.4.

We set the `mpr_name` char pointer to the same location as the `maddr`. From there we can extract a neighbour's name and copy it into the internal message format. After that has been done we increase the mpr_name pointer's memory location to point at the length of the neighbour's name further ahead in memory.

The location is now either the end of the TC message body or at the next neigbour's IP address and `nam_len` values. The `maddr` pointer is therefore set to the same location as the `mpr_name` after it has been increased. We are now finished with the extension that regards to parsing of the new TC message.

A similar extension is done for IPv6 parsing of TC messages, and can be seen in Appendix A.

When the converting of the internal TC message format is finished, `olsr_process_received_tc()` forwards it to `olsr_tc_tap()` for further processing. Originally, this code nearly mere updates the topology set. We also need to update the name table with the originator's name and the all the names of the neighbours.

The validity time is the same as stored in the TC message. In that case, the name table and routing table will be synchronized when it comes to updated information. To update the name table, the following implementation has been done in `olsr_tc_tap()`:

```
/* Øyvind - If SYM neighbor, extract the information from
 * the TC header to update the name entry
 */
insert_name_entry(message->originator, message->name, message->vtime);

/* We also insert the mpr node names */
mpr = message->multipoint_relay_selector_address;

while(mpr!=NULL)
{
```

```
      insert_name_entry(mpr->address, mpr->name, message->vtime);
      mpr=mpr->next;
}
//End of extension
```

With these extensions to the `olsr_tc_tap()`, all names have been taken care of and we are finished with the processing of the TC message.

4.2.6 Other changes in OLSRd

In order to initialise the name table, the `olsr_init_nametable()` must be called when the initialisation of OLSRd is happening. These calls are done in `olsr.c` in the `olsr_init_tables()` function. This initial procedure is done when OLSRd is starting. `olsr_init_tables()` needs only one extension at the bottom after other initialisation calls:

```
/* Initialize nametable (Øyvind) */
olsr_init_nametable();
```

During the implementation process, the need to have an overview of the name table has been present. This is quite useful for debugging purposes. When printing out information to standard output about neighbours and topology the `print_nametable()` in `tc_set.c`, is used to view information about the name table on the screen. This function is designed in the same way as the other print function for showing information about other tables to the screen. `print_nametable()` is implemented to be called right after all of the other print functions in `olsr_process_changes()` located in `olsr.c`. The only extension of this function was to implement a call to the `print_nametable()` function:

```
print_nametable(); //Extension by Øyvind
```

4.3 Testing the implementation

This section tests if our implementation works as expected according to our design.
We first describe our test environments in Section 4.3.1. Further in Section 4.3.2, 4.3.3, 4.3.4 and 4.3.5 we describe the different tests.

Our tests do not take into consideration environmental factors, such as node mobility, grey zones etc. We only want to test the functionality in these tests. If the tests pass without respect to environmental factors, we also expect that our solution behave similar as OLSRd acts with these factors. The OLSRd program has run through many tests earlier, and we expect that our extension not affects its behaviour. We have only extended the control messages hence the behaviour should be the same as before, regarding to packet loss and other failures that occurs from environmental factors.

As a start a simple test is done between three one-hop neighbours in Section 4.3.2. Then we start changing the topology to be more and more complex to see if the names are successfully distributed to all nodes independently of the complexity of the topology.

4.3.1 Test environments

To test our implementation we have two test environments. These are described in the next subsections.

4.3.1.1 Real nodes

We can run OLSRd on real hosts. A main limitation is that we can not hire people at the university campus to run around with mobile nodes. The alternative is therefore to use only a number of nodes that are connected to each other to test basic functionality of our implementation.

4.3.1.2 NEMAN

NEMAN [14] is an emulation environment that is able to emulate a MANET with hundreds of nodes on the computer that runs it. This is done by creating emulated interfaces, where each emulated node is assigned an interface.

NEMAN consists of two parts: A background process, called `topoman`, which emulates the interfaces, and a GUI to interact with this process.

The GUI has to be run on a different machine than the machine that the background process of the emulation runs at. In order to send control messages, from the GUI at one machine to the other machine that runs the emulation, a listener is needed to receive NEMAN control messages and to interact with `topoman`. This is accomplished with a shell script called `listen.sh`, and has to be run all the time where `topoman` is running.

The GUI of NEMAN has these control buttons as illustrated in Figure 4.5.

Figure 4.5: The control buttons in the graphical user interface of NEMAN.

OpenFile opens a scenario file. This is a file that describes how the topology initially is going to be, which implies setting correct links between neighbours and coordinates in the emulated area. An example of such a scenario file is shown below:

```
# Coordinates of the nodes
$node_(0) set X_ 100.0
$node_(0) set Y_ 100.0
$node_(0) set Z_ 0.000000000000
$node_(1) set X_ 250.0
$node_(1) set Y_ 100.0
$node_(1) set Z_ 0.000000000000
$node_(2) set X_ 400.0
$node_(2) set Y_ 100.0
$node_(2) set Z_ 0.000000000000
```

```
# Links definitions
$god_ set-dist 0 1 1
$god_ set-dist 1 2 1
```

This is the scenario file used in test presented in Section 4.3. Appendix A contains scenario files on this format for each test.

If we push *Prepare* after we have opened a scenario file, the GUI sends control messages to `topoman` at the other machine. This control message contains information about all the links and which nodes that exist in the MANET. Figure 4.6 presents an illustration of the GUI after the scenario file has been loaded and the *Prepare* button has been pushed. After the *Prepare* button has been pushed, we can start the OLSR routing protocol implementation on each emulated interface by pushing *Start OLSRDs*.

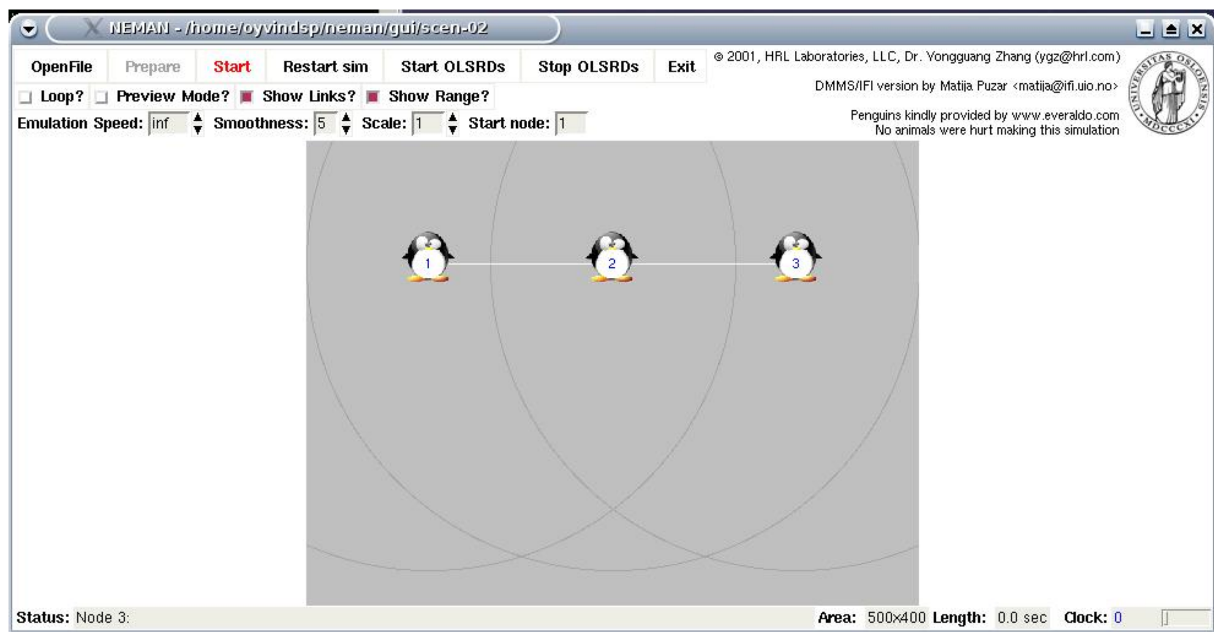

Figure 4.6: The graphical user interface after the scenario file is loaded, and the "Prepare" has been pushed. At the top we can see the control buttons. Below them we can see the geographical area where the nodes are simulated to be.

For testing our implementation, NEMAN gives us one challenge that we must solve. We do not have the possibility to read the standard output on the emulated nodes. In addition we do not have the possibility to write hosts files from all the emulated nodes, since they run at the same host. The hosts file will be shared among all of the emulated nodes, which means that this will not work. When we use NEMAN to test our implementation, we must be able to read information that the nodes get during exchanging the OLSR control messages in another way.

OLSRd has support for so-called IPC connections, and is used by different plug-ins and the OLSRd GUI front-end. The code for the IPC front-end is located in `ipc_frontend.c` (see Appendix A). By default, a TCP socket is bound to port 1212. Information about new routes is printed out as messages on this socket. A function in `ipc_frontend.c`, named `ipc_route_send_rtentry()` is responsible for this task. Originally it prints out routing table information on this form:

```
1,10.0.0.2,10.0.0.2,1,eth0
```

This line tells us that there is a route to destination `10.0.0.2` through `10.0.0.2` that is 1 hop away.

We must extend this message, to also print out the hostname of the destination node. The new design of each message that is sent on TCP port 1212 is then:

```
1,10.0.0.2,10.0.0.2,1,eth0,<destination host name>
```

In order to do this, we must include `tc_set.h` in the header of `ipc_frontend.c` to be able to read our name table:

```
#include "tc_set.h"
```

Then we start with the extension of `ipc_route_send_rtentry()`. We first ask the name table with the `find_name_entry()` function to find the name of the destination node of the routing table entry. We then need a `name_entry` pointer and a char buffer to store the name. This is declared among the other variable declarations in the top of the `ipc_route_send_rtentry()` function.:

```
struct name_entry *dest_name_entry; //Extension by Øyvind
char *tmp, tmp2[10], dst_hostname[255]; //New char buffer by Øyvind
```

If no name is found to the destination address, we print out an appropriate message. This can be the case, since it can take some time to transport the TC messages to all possible nodes on the MANET. The extension is implemented as follows:

```
/* Extension by Øyvind */
dest_name_entry = find_name_entry(dst);
if(dest_name_entry == NULL)
      strcpy(dst_hostname, "No name found yet");
else
      strcpy(dst_hostname, dest_name_entry->name);

/* Modified by Øyvind to include the name or
 * the "No name found yet" string of the message:
sprintf(packet2, "%d,%s,%s,%s,%d,%s\n", add, olsr_ip_to_string(dst),
        dst_hostname, tmp, met, tmp2);
/* End of extension */
```

If we connect to the TCP port 1212 with a program like `netcat` onto a node that runs OLSR, we are able to read all routing destinations together with the hostname. This is the way we read information from the name table when using NEMAN to test our implementation.

There is still one challenge left regarding the connection to the TCP port 1212. All the nodes emulated in NEMAN run on one computer. Since they run on the same computer, the TCP port can only be bound to one interface at a time. A shared library has been created to accomplish this task. This library is called `libsocktap`, and is developed by Matija Pužar and Sergio Cabrero Barros at the University of Oslo, Department of Informatics.

In order to make the testing easy, a shell script is made. This script takes an interface name as an argument. The shell script binds the TCP socket on port 1212 to that interface given as argument, and display the information in `netcat`. The shell script is called `check_neighbours.sh` (see Appendix A) and implemented as follows:

```
#!/bin/sh

LD_PRELOAD=/usr/local/lib/libsocktap.so SO_BINDTODEVICE=$1
CONVERT_PORT=1212 nc localhost 1212
```

With this shell script together with the extended IPC front-end we can easily see all routing destinations with corresponding destination names at each emulated interface in NEMAN.

The nodes are identified by the interface which they are running on. They are named `tap1`, `tap2`, `tap3` to `tapn`, and have to corresponding IP addresses `10.0.0.1`, `10.0.0.2`, `10.0.0.3` to `10.0.x.x`. To check the information from the node with IP address `10.0.0.2`, we run `check_neighbours.sh` like this:

```
./check_neighbours.sh tap2
```

As an example we will get the following on the standard output :

```
1,10.0.0.1,10.0.0.1,1,tap2,dmms-lab119-tap1
```

The output can be read as: We have a route to destination address `10.0.0.1` through node with IP address `10.0.0.1` from interface `tap2` with destination name `dmms-lab119.ifi.uio.no-tap1`.

4.3.2 Test scenario 1

The first test consists of a simple topology which is illustrated in Figure 4.7. It is an important test, because we can test to see whether or not names are transported correctly through HELLO messages. In this topology we expect that there will not be exchanged any TC message if `TcRedundancy` is `0`. If we set this value to `2`, we expect that TC messages will be sent between the neighbours, and that HELLO messages are generated without names. This is the main purpose of this test, to see whether or not this functionality is working as expected.

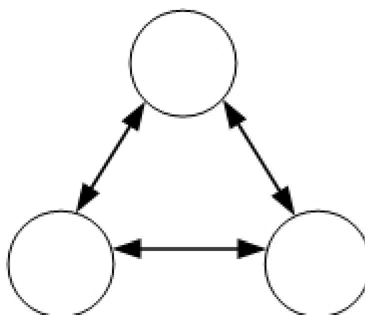

Figure 4.7: The topology of the first test scenario.

Since we are only having one-hop neighbours, we have the ability to use real nodes with wired connection between them as described in Section 4.3.1.1. We can then test the `print_nametable()` function that prints out the name table to standard output. In addition we can also test to see if the `write_hostsfile()` function works.

The topology in Figure 4.7 consists of the nodes that are described in Table 4.7.

IP address	OS hostname	Interface name	OLSR hostname
10.0.0.11	debiandesktop	eth0	debinandesktop-eth0
10.0.0.3	debianlaptop	eth0	debinalaptop-eth0
10.0.0.150	debianserver	eth0	debianserver-eth0

Table 4.7: Configuration of the nodes in the first test scenario.

We will now run OLSRd on each of these hosts. The test scenario in this section will be used to perform two individual tests. In the first test, we will set `TcRedundancy` to `0` which is the default value. Then, in the second test we set it to `2`.

The next two section prints out the information that are printed to standard output on each of the hosts.

4.3.2.1 Results with `TcRedundancy` equal to zero

This section presents the results when `TcRedundancy` is set equal to zero. The standard output that OLSRd generates is the result that we use in the evaluation. The nodes used in this test are the nodes that are declared in Table 4.7.

Standard output on debiandesktop (10.0.0.11):

```
       *** olsr.org - 0.4.10 (Nov 20 2007) ***

--- 17:13:52.11 ---------------------------------------------------- LINKS

IP address       hyst   LQ     lost    total  NLQ    ETX
10.0.0.150       0.875  0.000  0       0      0.000  0.00
10.0.0.3         0.938  0.000  0       0      0.000  0.00

--- 17:13:52.11 ------------------------------------------------ NEIGHBORS

IP address       LQ     NLQ    SYM   MPR   MPRS  will
10.0.0.3         0.000  0.000  YES   NO    NO    3
10.0.0.150       0.000  0.000  YES   NO    NO    6

--- 17:13:52.11 ------------------------------------------------- TOPOLOGY

Source IP addr   Dest IP addr     LQ     ILQ    ETX

--- 17:13:52.11 ------------------------------------------------- NAMETABLE

Destination IP addr   Destination Name
10.0.0.3              debianlaptop-eth0
10.0.0.150            debianserver-eth0
```

Standard output on debianlaptop (10.0.0.3):

```
       *** olsr.org - 0.4.10 (Nov 20 2007) ***

--- 17:13:44.90 ----------------------------------------------------- LINKS

IP address       hyst   LQ     lost    total  NLQ    ETX
10.0.0.11        0.938  0.000  0       0      0.000  0.00
10.0.0.150       0.984  0.000  0       0      0.000  0.00

--- 17:13:44.90 ------------------------------------------------- NEIGHBORS

IP address       LQ     NLQ    SYM   MPR   MPRS  will
10.0.0.11        0.000  0.000  YES   NO    NO    3
10.0.0.150       0.000  0.000  YES   NO    NO    6

--- 17:13:44.90 -------------------------------------------------- TOPOLOGY

Source IP addr   Dest IP addr     LQ     ILQ    ETX

--- 17:13:44.90 -------------------------------------------------- NAMETABLE

Destination IP addr   Destination Name
10.0.0.150            debianserver-eth0
10.0.0.11             debiandesktop-eth0
```

Standard output on debianserver (10.0.0.150):

```
       *** olsr.org - 0.4.10 (Nov 20 2007) ***

--- 17:14:33.16 ----------------------------------------------------- LINKS

IP address       hyst   LQ     lost    total  NLQ    ETX
10.0.0.11        0.938  0.000  0       0      0.000  0.00
10.0.0.3         0.984  0.000  0       0      0.000  0.00

--- 17:14:33.16 ------------------------------------------------- NEIGHBORS

IP address       LQ     NLQ    SYM   MPR   MPRS  will
10.0.0.3         0.000  0.000  YES   NO    NO    3
10.0.0.11        0.000  0.000  YES   NO    NO    3

--- 18:14:33.16 -------------------------------------------------- TOPOLOGY

Source IP addr   Dest IP addr     LQ     ILQ    ETX

--- 17:14:33.16 -------------------------------------------------- NAMETABLE

Destination IP addr   Destination Name
10.0.0.3              debianlaptop-eth0
10.0.0.11             debiandesktop-eth0
```

From these results, we can see that all names are found on each node. HELLO messages distribute names correctly, when they shall do it. We can also see that the topology table is empty on each host. This means that none of the hosts have received a TC message, which is the case when we only have a MANET with one-hop neighbours and `TcRedundancy` set to `0`.

As a conclusion of this test, we can say that sending and receiving HELLO messages with our extension works.

4.3.2.2 Results with `TcRedundancy` equal to two

This section presents the results when `TcRedundancy` is set equal to two. The standard output that OLSRd generates is the result that we use in the evaluation. The nodes used in this test are the nodes that are declared in Table 4.7.

Standard output on debiandesktop (10.0.0.11):

```
       *** olsr.org - 0.4.10 (Nov 20 2007) ***

--- 17:21:27.70 ---------------------------------------------------- LINKS

IP address       hyst   LQ      lost    total  NLQ     ETX
10.0.0.150       0.992  0.000  0       0       0.000  0.00
10.0.0.3         0.999  0.000  0       0       0.000  0.00

--- 17:21:27.70 ------------------------------------------------ NEIGHBORS

IP address       LQ      NLQ     SYM   MPR   MPRS  will
10.0.0.3         0.000  0.000  YES   NO    NO    3
10.0.0.150       0.000  0.000  YES   NO    NO    6

--- 17:21:27.70 ------------------------------------------------- TOPOLOGY

Source IP addr   Dest IP addr     LQ      ILQ     ETX
10.0.0.3         10.0.0.150       0.000  0.000  0.00
10.0.0.3         10.0.0.11        0.000  0.000  0.00
10.0.0.150       10.0.0.11        0.000  0.000  0.00
10.0.0.150       10.0.0.3         0.000  0.000  0.00

--- 17:21:27.70 ------------------------------------------------- NAMETABLE

Destination IP addr   Destination Name
10.0.0.3              debianlaptop-eth0
10.0.0.150            debianserver-eth0
```

Standard output on debianlaptop (10.0.0.3):

```
       *** olsr.org - 0.4.10 (Nov 20 2007) ***

--- 17:21:20.59 ----------------------------------------------------- LINKS

IP address       hyst   LQ     lost   total  NLQ    ETX
10.0.0.150       0.992  0.000  0      0      0.000  0.00
10.0.0.11        0.998  0.000  0      0      0.000  0.00

--- 17:21:20.59 ------------------------------------------------- NEIGHBORS

IP address       LQ     NLQ    SYM   MPR   MPRS  will
10.0.0.11        0.000  0.000  YES   NO    NO    3
10.0.0.150       0.000  0.000  YES   NO    NO    6

--- 17:21:20.59 -------------------------------------------------- TOPOLOGY

Source IP addr   Dest IP addr     LQ     ILQ    ETX
10.0.0.11        10.0.0.150       0.000  0.000  0.00
10.0.0.11        10.0.0.3         0.000  0.000  0.00
10.0.0.150       10.0.0.11        0.000  0.000  0.00
10.0.0.150       10.0.0.3         0.000  0.000  0.00

--- 17:21:20.59 -------------------------------------------------- NAMETABLE

Destination IP addr   Destination Name
10.0.0.11             debiandesktop-eth0
10.0.0.150            debianserver-eth0
```

Standard output on debianserver (10.0.0.150):

```
       *** olsr.org - 0.4.10 (Nov 20 2007) ***

--- 17:22:09.33 ----------------------------------------------------- LINKS

IP address       hyst   LQ     lost   total  NLQ    ETX
10.0.0.3         0.992  0.000  0      0      0.000  0.00
10.0.0.11        0.992  0.000  0      0      0.000  0.00

--- 17:22:09.33 ------------------------------------------------- NEIGHBORS

IP address       LQ     NLQ    SYM   MPR   MPRS  will
10.0.0.3         0.000  0.000  YES   NO    NO    3
10.0.0.11        0.000  0.000  YES   NO    NO    3

--- 17:22:09.33 -------------------------------------------------- TOPOLOGY

Source IP addr   Dest IP addr     LQ     ILQ    ETX
10.0.0.3         10.0.0.150       0.000  0.000  0.00
10.0.0.3         10.0.0.11        0.000  0.000  0.00
10.0.0.11        10.0.0.150       0.000  0.000  0.00
10.0.0.11        10.0.0.3         0.000  0.000  0.00

--- 17:22:09.33 -------------------------------------------------- NAMETABLE

Destination IP addr   Destination Name
10.0.0.11             debiandesktop-eth0
10.0.0.3              debianlaptop-eth0
```

The time elapsed to update correct information in the name table was greater than the test in Section 4.3.2.1. It was updated at the same time when the topology table was updated. This is an expected result, since all names are supposed to be only exchanged by TC messages. We also see from the Topology table in standard output that all nodes got TC messages with all known neighbours, because the topology table consists of all possible paths on the network. This is a consequence of that `TcRedundancy` is `2`.

The test was successful since we can see that HELLO messages do not include names when `TcRedundancy` is `2`. In addition we can conclude that sending and receiving of the new type of TC messages work correctly.

4.3.2.3 Other results of the first test scenario

From the results in Section 4.3.2.1 and 4.3.2.2 we have also tested the `print_nametable()` function. This function writes name table information of the OLSR standard output. It prints out the information shown below:

```
--- 17:22:09.33 ------------------------------------------------- NAMETABLE

Destination IP addr   Destination Name
10.0.0.11             debiandesktop-eth0
10.0.0.3              debianlaptop-eth0
```

The last test with this test scenario is to show if `/etc/hosts` file is correctly updated. It is written by the `write_hostsfile()` function which is called by the name table scheduler timeout function. The `/etc/hosts` file for each node is presented below:

```
debiandesktop:/home/oyvind/masteroppgave# cat /etc/hosts
127.0.0.1       localhost       #localhost
10.0.0.11       debiandesktop   #myself
10.0.0.3        debianlaptop-eth0       #debianlaptop-eth0
10.0.0.150      debianserver-eth0       #debianserver-eth0


debianlaptop:/home/oyvind/masteroppgave# cat /etc/hosts
127.0.0.1       localhost       #localhost
10.0.0.3        debianlaptop    #myself
10.0.0.11       debiandesktop-eth0      #debiandesktop-eth0
10.0.0.150      debianserver-eth0       #debianserver-eth0



debianserver:/home/oyvind/masteroppgave# cat /etc/hosts
127.0.0.1       localhost       #localhost
10.0.0.150      debianserver    #myself
10.0.0.11       debiandesktop-eth0      #debiandesktop-eth0
10.0.0.3        debianlaptop-eth0       #debianlaptop-eth0
```

All names are stored, and applications on each of the hosts, will now be able to perform a DNS request to reach all destinations on the network. There is only one comment. The names stored on the first and the second line are only the OS hostname. The name convention as discussed in Section 3.4.1 is not used for names for the node itself, because we not need this in NEMAN.

4.3.3 Test scenario 2

In this test scenario and the following ones, we use the network emulator NEMAN as described in Section 4.3.1.2. Scenario files that have been created for NEMAN can be seen in Appendix A.

In this test we want to examine if the names that should be distributed through intermediate nodes. The `TcRedundancy` variable in `olsrd.conf` is set to `0` and the topology that is used in this test scenario is illustrated in Figure 4.8.

In this scenario, the names will be distributed as follows, if our implementation works as intended:

The names of node 1 and 3 will be distributed through HELLO messages. Node 2 will generate TC messages that consist of addresses and names of the nodes 1 and 2. In this way, the name of node 1 will be distributed so that node 3 receives it. The final expected result is that all nodes have bindings between names and corresponding IP addresses.

We use the GUI on NEMAN to start our version of OLSRd, and use the `check_neighbour.sh` script that was described in Section 4.2.6.

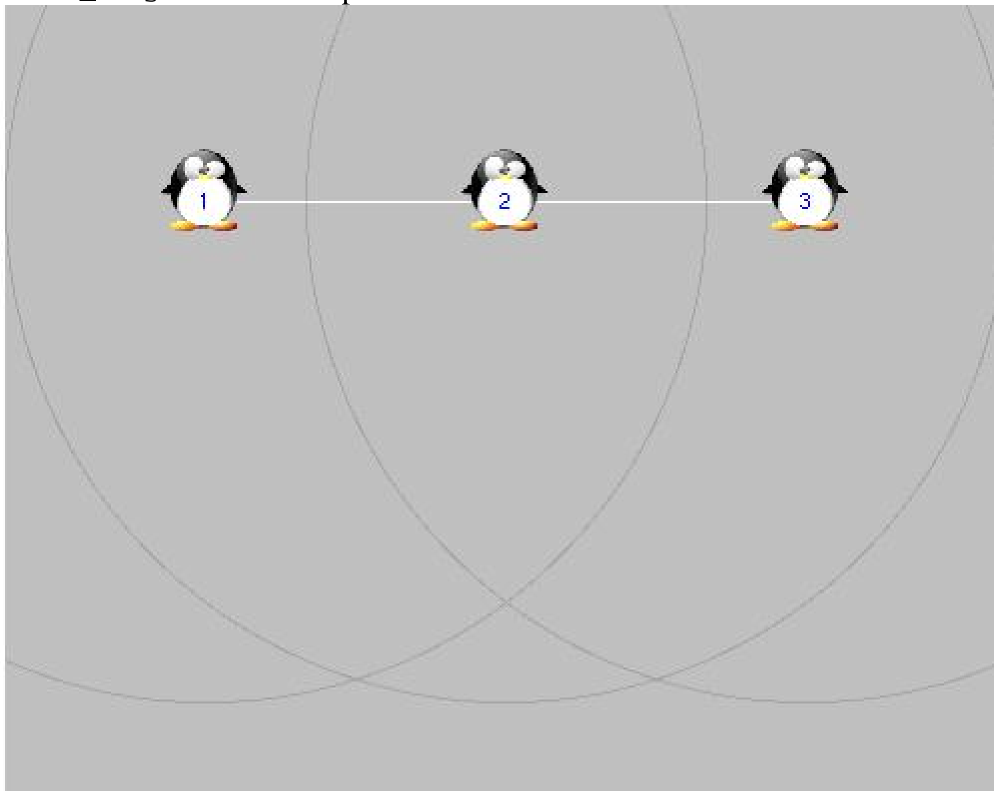

Figure 4.8: Topology used in the second test scenario. The topology shown here is a snapshot from the NEMAN GUI.

We get the following results of the different nodes:

Node 1 that is running on IP address 10.0.0.1 on interface tap1:

```
[root@dmms-lab119 sbin]# ./check-neighbours.sh tap1
1,10.0.0.2,10.0.0.2,1,tap1,dmms-lab119-tap2
1,10.0.0.3,10.0.0.2,2,tap1,dmms-lab119-tap3
```

Node 2 that is running on IP address 10.0.0.2 on interface tap2:

```
[root@dmms-lab119 sbin]# ./check-neighbours.sh tap2
1,10.0.0.1,10.0.0.1,1,tap2,dmms-lab119-tap1
1,10.0.0.3,10.0.0.3,1,tap2,dmms-lab119-tap3
```

Node 3 that is running on IP address 10.0.0.3 on interface tap3:

```
[root@dmms-lab119 sbin]# ./check-neighbours.sh tap3
1,10.0.0.1,10.0.0.2,2,tap3,dmms-lab119-tap1
1,10.0.0.2,10.0.0.2,1,tap3,dmms-lab119-tap2
```

The conclusion of these results is that the names are successfully distributed through both HELLO messages and TC messages. In addition we can conclude that building and parsing of both message types are successful in this simple topology. The name table has also been successfully updated on all nodes.

4.3.4 Test scenario 3

In this scenario we create a topology where one of the nodes has more neighbours than the others. The topology is illustrated in Figure 4.9. In that way, we can test if the implementation of the TC message body is done correctly when it comes to building it and parsing it. We also here test if names get distributed through intermediate nodes in a more complex topology.

The `TcRedundancy` variable in `olsrd.conf` is set to `0`. NEMAN is used to emulate all the nodes, and the `check_neighbours.sh` script is used to print out results.

We have a similar topology in this test scenario (Figure 4.9) as we saw in Section 4.3.3. The nodes are organised in a chain. The difference is that the most central node, 3, has four neighbours. If our implementation works, the HELLO messages transport the names of node 1 and node 5 respectively to node 2 and 4. This is also the case of node 6 and 7 that will transport their names through HELLO message to node 3. The rest of the nodes in this topology are MPR nodes. They will therefore announce their names through TC messages and include their MPR selectors so that all nodes get all the names.

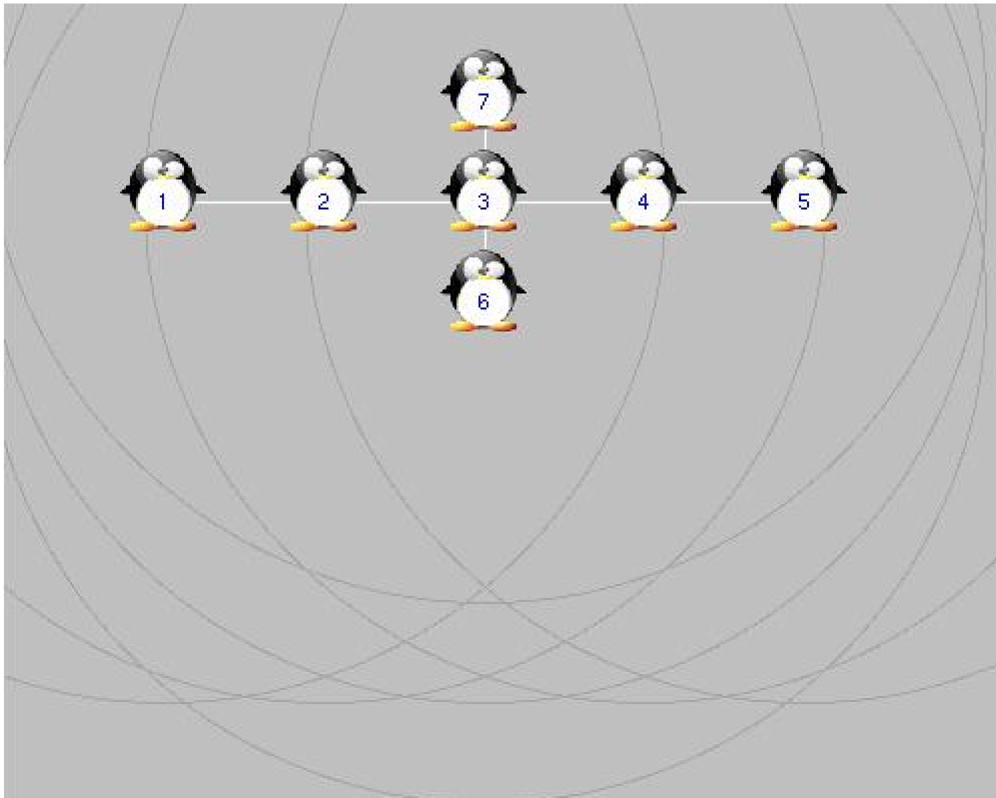

Figure 4.9: Topology used in the third test scenario. The topology shown here is a snapshot from the NEMAN GUI.

We show only some results from chosen nodes. The nodes we chose to show information from are: 1, 5, and 7. The results from the `check_neighbours.sh` script for these nodes are:

```
[root@dmms-lab119 sbin]# ./check-neighbours.sh tap1
1,10.0.0.2,10.0.0.2,1,tap1,dmms-lab119-tap2
1,10.0.0.3,10.0.0.2,2,tap1,dmms-lab119-tap3
1,10.0.0.4,10.0.0.2,3,tap1,dmms-lab119-tap4
1,10.0.0.5,10.0.0.2,4,tap1,dmms-lab119-tap5
1,10.0.0.6,10.0.0.2,3,tap1,dmms-lab119-tap6
1,10.0.0.7,10.0.0.2,3,tap1,dmms-lab119-tap7

[root@dmms-lab119 sbin]# ./check-neighbours.sh tap5
1,10.0.0.1,10.0.0.4,4,tap5,dmms-lab119-tap1
1,10.0.0.2,10.0.0.4,3,tap5,dmms-lab119-tap2
1,10.0.0.3,10.0.0.4,2,tap5,dmms-lab119-tap3
1,10.0.0.4,10.0.0.4,1,tap5,dmms-lab119-tap4
1,10.0.0.6,10.0.0.4,3,tap5,dmms-lab119-tap6
1,10.0.0.7,10.0.0.4,3,tap5,dmms-lab119-tap7

[root@dmms-lab119 sbin]# ./check-neighbours.sh tap7
1,10.0.0.1,10.0.0.3,3,tap7,dmms-lab119-tap1
1,10.0.0.2,10.0.0.3,2,tap7,dmms-lab119-tap2
1,10.0.0.3,10.0.0.3,1,tap7,dmms-lab119-tap3
1,10.0.0.4,10.0.0.3,2,tap7,dmms-lab119-tap4
1,10.0.0.5,10.0.0.3,3,tap7,dmms-lab119-tap5
1,10.0.0.6,10.0.0.3,2,tap7,dmms-lab119-tap6
```

All nodes got name from the other nodes. The implementation did not have any problems with this topology.

4.3.5 Test scenario 4

This test scenario is the last test that our implementation goes through. The topology consists of one node that is very central to the other nodes, and has to be chosen as a MPR node to reach one “outsider” node. The topology is illustrated in Figure 4.10, and its complexity has been increased. This scenario tests to see whether or not the names are distributed correctly through either HELLO messages or TC messages.

The `TcRedundancy` variable in `olsrd.conf` is set to `0`. NEMAN is used to emulate all the nodes, and the `check_neighbours.sh` script is used to print out results.

In the topology that is illustrated in Figure 4.10 we expect that only node 3 and 5 will be selected as MPR nodes. Further, if everything works as expected, node 3 will include the addresses and names of node 5 and 6, while node 5 will include addresses and names of nodes 1, 2, 3 and 4. Node 3 will therefore distribute the name of itself and node 6 through TC messages to the other nodes. Node 5 will distribute the names of itself and node 1, 2, 3, and 4.

In that way node 6 will get the names from all nodes on the MANET through TC messages from node 5. The name distribution between the MPR selectors of node 5 will go through HELLO messages. In that way, node 5 can include their names into the TC message.

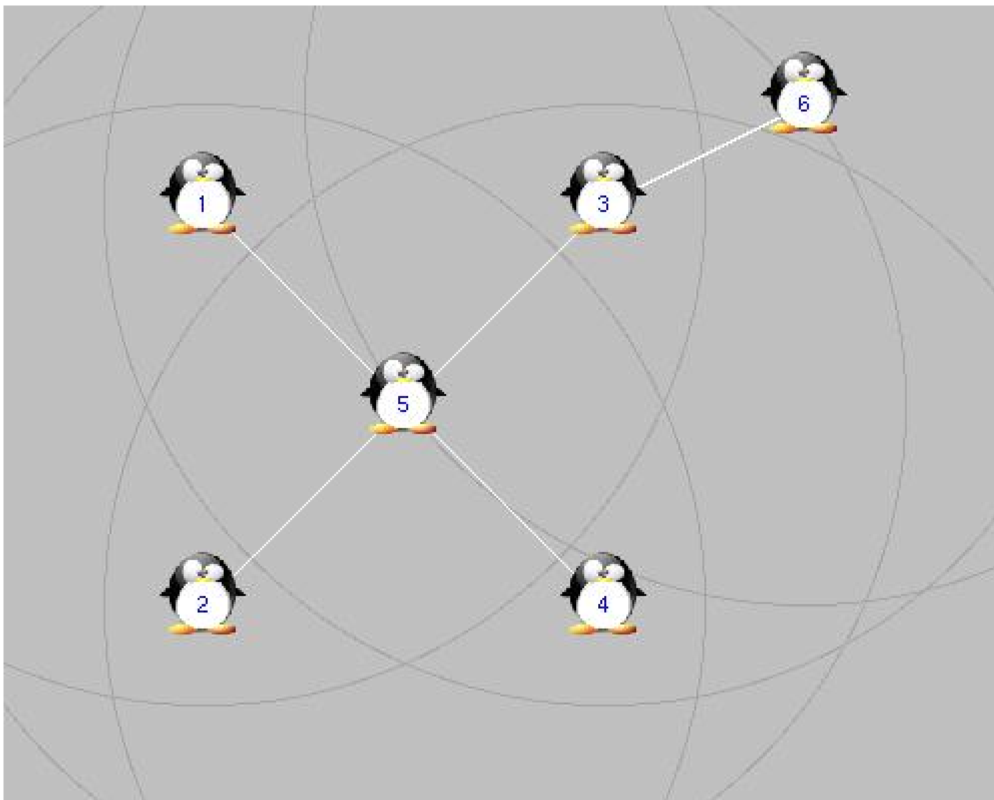

Figure 4.10: Topology used in the fourth test scenario. The topology shown here is a snapshot from the NEMAN GUI.

We present only results from some chosen nodes. These are node 1, 5 and 6.

```
[root@dmms-lab119 sbin]# ./check-neighbours.sh tap1
1,10.0.0.2,10.0.0.5,2,tap1,dmms-lab119-tap2
1,10.0.0.3,10.0.0.5,2,tap1,dmms-lab119-tap3
```

```
1,10.0.0.4,10.0.0.5,2,tap1,dmms-lab119-tap4
1,10.0.0.5,10.0.0.5,1,tap1,dmms-lab119-tap5
1,10.0.0.6,10.0.0.5,3,tap1,dmms-lab119-tap6

[root@dmms-lab119 sbin]# ./check-neighbours.sh tap5
1,10.0.0.1,10.0.0.1,1,tap5,dmms-lab119-tap1
1,10.0.0.2,10.0.0.2,1,tap5,dmms-lab119-tap2
1,10.0.0.3,10.0.0.3,1,tap5,dmms-lab119-tap3
1,10.0.0.4,10.0.0.4,1,tap5,dmms-lab119-tap4
1,10.0.0.6,10.0.0.3,2,tap5,dmms-lab119-tap6

[root@dmms-lab119 sbin]# ./check-neighbours.sh tap6
1,10.0.0.1,10.0.0.3,3,tap6,dmms-lab119-tap1
1,10.0.0.2,10.0.0.3,3,tap6,dmms-lab119-tap2
1,10.0.0.3,10.0.0.3,1,tap6,dmms-lab119-tap3
1,10.0.0.4,10.0.0.3,3,tap6,dmms-lab119-tap4
1,10.0.0.5,10.0.0.3,2,tap6,dmms-lab119-tap5
```

The results presented here, show that all nodes got names from the other nodes through HELLO messages and TC messages. Our implementation also passed this test.

4.3.6 Conclusion – testing functionality

Our implementation passed all the tests presented in this section. Based on the results we now have a name service that will work independently of the topology structure. HELLO messages and TC messages work in harmony to update the name table.

4.4 Summary

This chapter first analysed which elements we needed to change by modifying them or extending them. Then we discussed how we implemented our design. We needed to implement modifications and extensions for generation of HELLO messages and TC messages. A consequence of this was that we also needed to do modifications when OLSRd parses these packets. The name table was created to store names. A timeout function deletes old entries and writes updated information into the `hosts` file. The last section of this chapter described how we tested our implementation. We declared four test scenarios that should prove to us that our implementation works. In order to test the solution on a real MANET, we used the MANET emulator NEMAN. All the tests passed, which proves that our solution works independently of the topology structure.

5 Evaluation

This chapter evaluates our implementation of the name extension in OLSRd. First, we declare our measurement metrics, and what input parameters that we use to measure them. Results of the experiments are presented and analysed.

5.1 Test methods

This section describes how we design different experiments to evaluate our implementation. For a design of different experiments we first have to define the metrics and which input parameters we should use to measure the metrics. From the parameters we can create individual experiments that measure the metrics in different ways. The metrics should be able to measure if we fulfil our claims that are declared in Section 1.1.4.

Our main goal with these tests is to measure the total amount of overhead that our implementation generates on the network in a given time interval, and compare the overhead against other variants of the OLSR protocol. In our context overhead means the load in bytes on the network that corresponds to the payload of the OLSRd control messages.

In OLSRd all control messages are stored in OLSRd packets. Hence, we also want to capture the number of packets that are generated through the given time interval. This is because we want to justify the result with respect to the overhead. The overhead is proportional to the number of packets captured. We need to see results with both of these two metrics in order to be able to conclude correctly.

The reason why overhead is an interesting metric, is that we declared as one of our claims (see Section 1.1.4) that our solution should be efficient. Our solution should be efficient, since bindings between names and IP addresses are created at the same time when a node is discovered. The question is how the overhead is affected by this.

Summarized, the metrics that we want to measure are:

- *Overhead* in a given time interval of the experiment.
- The *number of packets* that are captured during the time interval of the experiment.

The experimental design is designed with the experiment parameters shown in Figure 5.1.

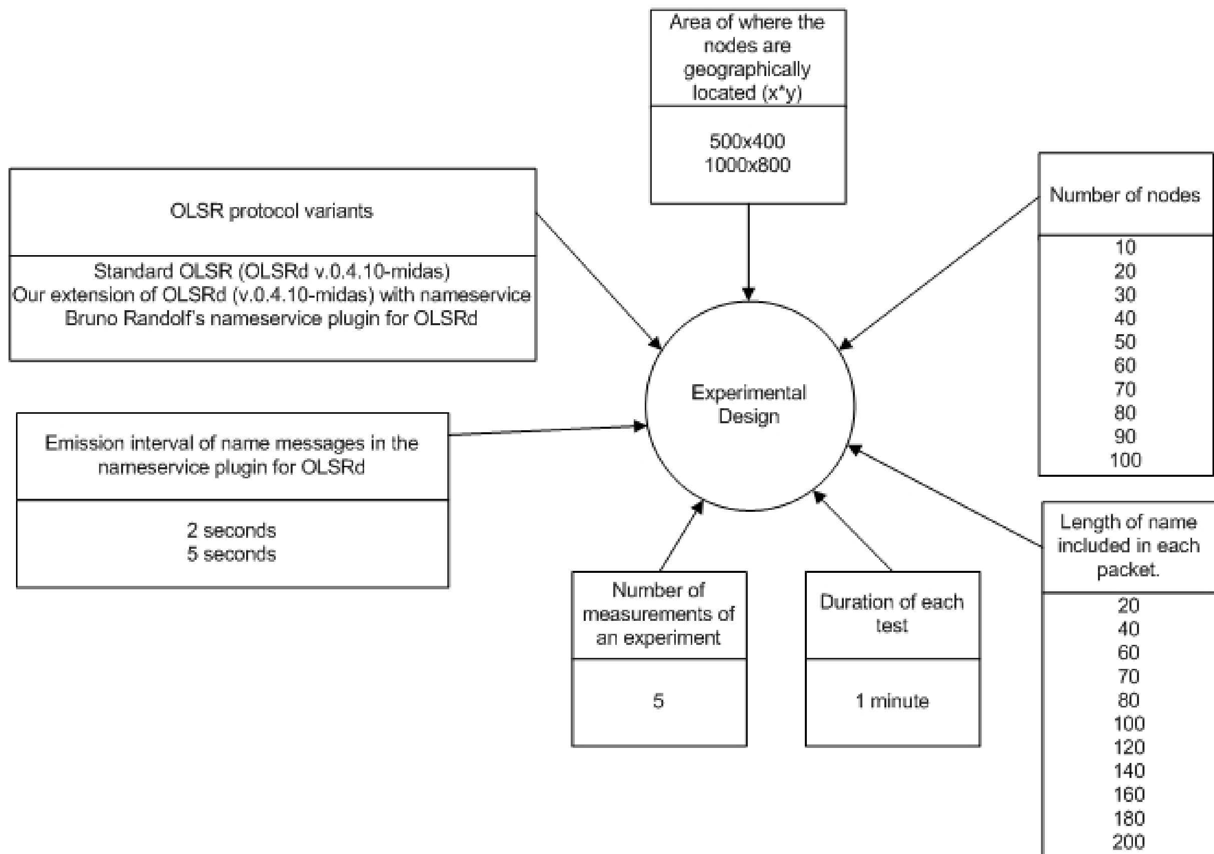

Figure 5.1: The experimental design with all input parameters and their values.

Each parameter has a different purpose to measure different parts of our implementation. The number of nodes is chosen to be between 10 and 100. With this parameter we can measure the behaviour of our implementation with different node densities.

We also choose to use two different areas of 500x400 and 1000x800, to see if there is any difference when the density of nodes decreases because of nodes are located in a greater space.

These parameters are used to compare the standard version of OLSRd, together with our implementation of the name extension in OLSRd and the name service plug-in that already exists with OLSRd.

In order to be able to compare our solution with the plug-in, we must modify the plug-in value of the emission interval of the name messages. The standard value is that the name messages are emitted every 2nd minute. In OLSRd the HELLO messages are sent every 2nd second, while the TC messages are broadcasted every 5th second. Hence, we change the value of the emission interval of name messages in the plug-in to be two seconds and five seconds.

The last part we want to measure is how the overhead metric increases with respect to the length of the hostname of a node. This is done by increasing the number of characters in the hostname for each experiment.

Each measurement has duration of one minute. During one minute we capture a number of packets that are able to explain the behaviour of OLSRd and give us a reliable result. All experiments are measured five times to get a more reliable average result. An average value

of the five measurements is enough in order to get a value that should not be affected by a great variance in one of the tests.

We have now defined what we want to measure, but we also need some tools in order to perform them:

- NEMAN – is used to emulate all nodes with the different versions of OLSRd.
- `tcpdump` – is used to capture the traffic from the monitor interface of NEMAN.
- Self made scripts – for measurements and extraction of the results, we create some scripts for that purpose.

NEMAN is used in these experiments and is briefly explained in Section 4.3.1.2. It has a monitor interface called `tap0`, where all traffic from all nodes are sent to. This traffic can then be captured with a tool like `tcpdump` and stored to a file with this command:

```
tcpdump –i tap0 port 698 –w result
```

All packets that are captured with this command are stored into a file called `result`. It also specifies which interface it should listen to, and a filter that specifies what traffic should be captured. In our case, the filter is `port 698`, because that all OLSRd traffic is transported through source and destination port `698`. In order to do this the `tcpdump` command above are included into a C-program, which controls that it runs five times with duration of 1 minute.

When the whole experiment is finished we can read the result file with `tcpdump` and extract the information that we are interested in. In our experiments we are, as mentioned, interested in the total amount of overhead during the experiment. Next, we have to summarize all the packet lengths. Each line in the result file corresponds to a packet, and holds all the information about it. A line from the result file might be presented as follows, with the `tcpdump -n -r result` command:

```
17:15:00.752225 IP 10.0.0.2.698 > 10.0.255.255.698: UDP, length: 77
```

There exist various versions of `tcpdump`. It depends on the version of `tcpdump` how a packet is presented, but in all our cases a packet is presented as illustrated above. We see that the length of the packet is located at the end of the line. We can then use the following command to extract all the lengths and summarize them:

```
tcpdump -n -r result | awk '{sum += $8} END {print sum}'
```

This command uses the `–n` option in order to not convert IP addresses into names. Next the result file is read with the `–r` option. Each line will then be piped to the `awk` program, which extracts the word number 8 in the line. This word is the last word in the line and is the packet length. When all packets are read by `tcpdump` and piped to `awk`, the sum of all packet lengths is printed to standard output.

A similar command can also get the number of packets in the result file:

```
tcpdump -n -r result | awk '{sum += 1} END {print sum}'
```

Each time a packet is read, the sum-variable is increased by one. When `tcpdump` is finished with reading all packets, the sum holds the number of packets read.

The information printed out from the two last `tcpdump` commands is stored in files for each measurement. A script gathers all information about an experiment into another file. From this file we can plot the results into a graph.

As mentioned we use NEMAN for emulation of virtual nodes on a MANET. The computer that runs the emulated nodes has two Intel Pentium Xeon 2.80 GHz Processors and 256 MB memory. The operating system is Linux with a 2.6.16 kernel.

When we run NEMAN with 100 virtual nodes, it consumes much of the computer's CPU capacity. Hence, it is necessary to have a computer with high CPU resources to avoid bottleneck behaviour which will affect our results.

5.2 Measured overhead in a 500x400 area

The first results that are presented are from the tests performed where nodes exist in a 500x400 area. There are two diagrams presented to present the average number of packets during the experiment and the total amount of overhead.

We first discuss our expectations of the results. The average number of captured packets should be more or less equal between the standard version of OLSRd and our name extension of it. This because that we have not extended OLSRd in such a way that there should be generated more messages during a time interval (which is one minute used in these tests). We have only extended the payload of each packet. Nevertheless, there can be small differences in the number of captured packets because we can not guarantee when packets are generated during a time interval. The number of packets captured from the plug-in should be higher than our name extension of OLSRd and the standard OLSRd. The reason is that with the plug-in OLSRd will generate more messages than standard OLSRd. Another aspect is that it might be less number of packets that we expect from OLSRd with plug-in, since an OLSRd packet can include more than one message, which is illustrated in Figure 2.6. The last expectation of the results is that the number of packets captured is proportional with the number of nodes.

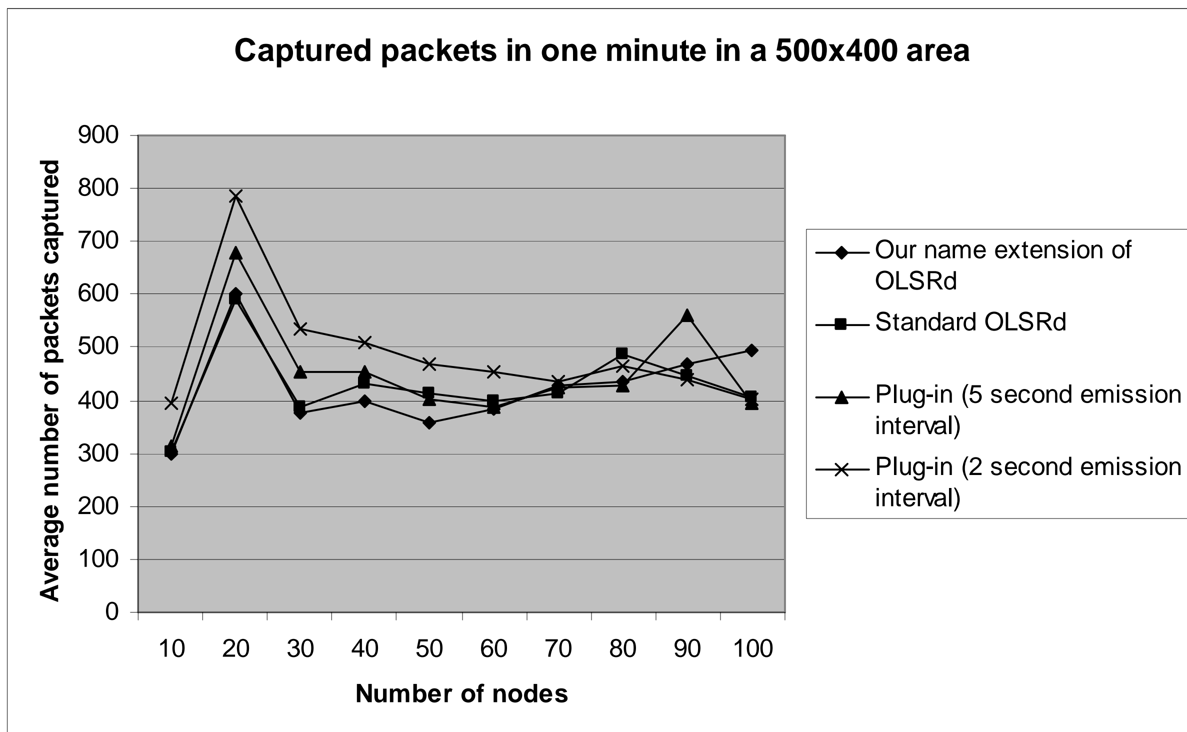

Figure 5.2: This diagram shows us how many packets that were captured during the experiment when nodes were stored in an area of 500x400.

Figure 5.2 presents the results of average packets captured in an area of 500x400. As expected our name extension and standard OLSRd have almost equal number packets, but we can locate some differences. Our OLSRd version with the name extension has less number of packets captured when number of nodes is between 30 and 50. In addition this is also the case when number of nodes is 80.

This difference from our expectations is difficult to explain from theory. There should be captured more packets with our solution rather than less. Our extensions of the control messages should generate more packets. The HELLO messages and the TC messages consume more space in the packet with respect to number of bytes in each message. Hence, the average number of messages in each packet should be less than in standard OLSRd. A consequence of this is that a higher number of packets must be generated in our name extension of OLSRd in order to transfer the equal number of messages as standard OLSRd. The aberration is though not that big. An explanation can be arbitrary factors such as hardware limitations or how NEMAN performs in a given scenario. We also have to look at the total amount of overhead before we conclude.

When we look at the two lines that are the results from the plug-in in Figure 5.2, the number of packets captured is higher or equal with our name extension. The exceptions are when number of nodes is 90 and 100. In these scenarios our solution generates a higher number of captured packets than the other versions of OLSRd. It is expected that our solution generates a higher number of packets than standard OLSRd, but not the plug-in versions. The reason can be an arbitrary factor as mentioned in the last paragraph.

Earlier, we expected that number of packets should be proportional to the number of nodes. This is not the case as we see in Figure 5.2. The number of packets captured increases between 10 and 20 nodes. Next, in all of the OLSRd versions the value decreases a lot when

the number nodes are 30. It flattens out and varies a bit up and down as we move right on the X-axis. The explanation of this might be the behaviour of OLSRd when the density of the nodes increases. If the density is high, which it is when there are 100 nodes in a 500x400 area, almost all nodes can hear each other. The result of this is that almost everyone becomes neighbours, and generation of TC messages will be at a minimum. This can be the explanation of the variance of number of packets captured.

When it comes to the total amount of overhead in each scenario, we expect that it coincides with number of packets captured. In addition we expect that our name extension in OLSRd has higher overhead in each scenario than standard OLSRd. The OLSRd with the name service plug-in is expected to be more or less equal to our solution. The results are presented in Figure 5.3.

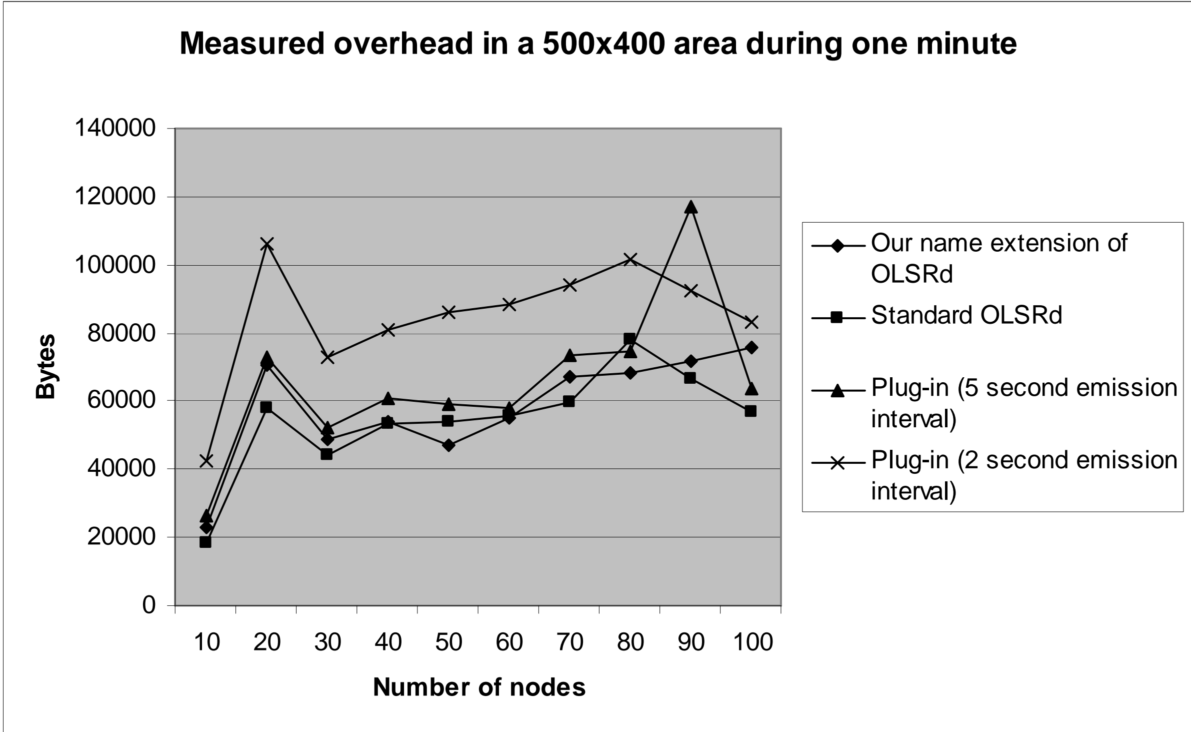

Figure 5.3: The measured overhead where nodes exist in a 500x400 area.

Figure 5.2 shows a diagram where the average packet size in bytes (which is the overhead), is the Y-axis while number of nodes is the elements on the X-axis The diagram in Figure 5.2 is in coherence with Figure 5.3 and they must be read together when we analyse them. If the number of packets is high, we get a higher payload and vice versa. There are some differences from the expected results but if we compare the overhead with the number of packets captured in Figure 5.2, it gives us a more reasonable understanding. As mentioned if the number of packets is high, we get a higher payload and vice versa.

The diagram in Figure 5.3 shows the measured overhead that corresponds to the packets captured in Figure 5.2. The results are as expected when the number of nodes is 10 and 20. Our solution and the plug-in with an emission interval of five seconds have almost equal values. We remark that the plug-in generates much more overhead than our solution when the emission interval is 2 seconds. This is also a behaviour that is expected.

When number of nodes is 30, the value of the overhead heavily decreases at all versions of the OLSRd protocol. This is a consequence of the behaviour of MPR mechanism. When number of nodes are 20, the nodes are organised such that the MPR nodes are many compared to the number of nodes. When the number of nodes is 30, the MPR calculation algorithm selects a better MPR set to reach all two hop neighbours. Hence, the packets are forwarded many times when the number of nodes is 20. This is the behaviour of OLSR, and proves that this routing protocol is more suitable for dense networks.

The trend of the diagram in Figure 5.3 when the number of nodes is 30 through the end shows more or less expected results. Our solution generates more overhead in one minute compared to standard OLSRd. In addition we see that the plug-in generates a higher payload than our solution in all scenarios, except from the scenario when number of nodes is 100. We also note that the plug-in with 2 second emission interval of name messages always generates more overhead than our name extension of OLSRd.

5.3 Measured overhead in a 1000x800 area

This section discusses the results from the tests where nodes exist in a 1000x800 area. There are two diagrams that respectively present the average number of packets during the experiment and the total amount of overhead.

Expected results from number of packets captured during one minute are values that are equal or higher when the nodes were in a 500x400 area which we discussed in Section 5.2. If the density decreases we can expect more MPR nodes that are needed to reach all two hop neighbours. The consequence is a higher value of retransmissions through intermediate nodes.

Further, we expect that our name extension of OLSRd and standard OLSRd, behave almost similarly. The difference is that each message is having a higher payload. Hence, we can expect more packets with our name extension of OLSRd than standard OLSRd.

Both of the OLSRd plug-in versions are expected to have higher number of packets captured than standard OLSRd and our extension of it. The name service plug-in uses name messages that are own control messages in OLSRd. Hence, since each host generates more control messages they have to create more packets in order to be able to emit them.

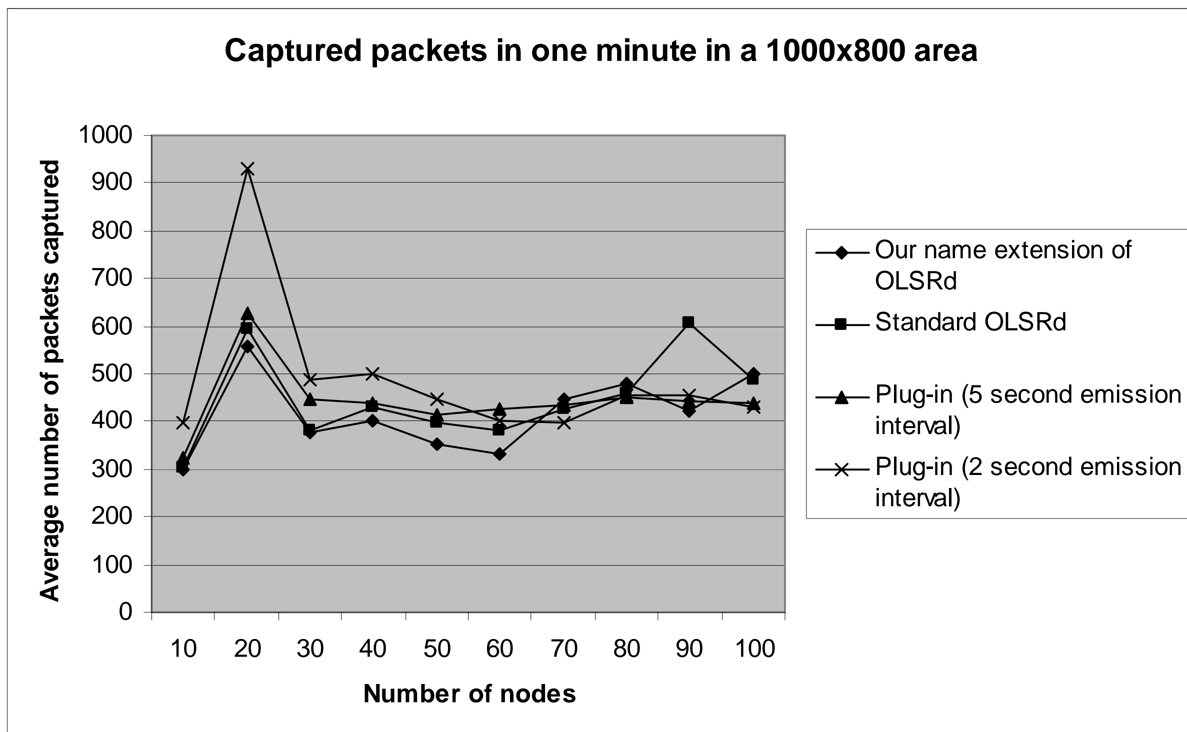

Figure 5.4: This diagram shows us how many packets that were captured during the experiment when nodes were stored in a area of 1000x800.

The diagram in Figure 5.4 presents the number of packets captured through all the scenarios when the area is 1000x800. Expected results can be seen when number of nodes are 10 and 20. Then the same behaviour can be seen when number of nodes is 30 as we saw in Figure 5.2. The number of packets decreases with a high rate, and can be explained with the same arguments as we discussed in Section 5.2: When the number of nodes is 20, the MPR nodes are selected in a way that leads to a higher number of forwardings. The number of MPR nodes is less when the number of nodes is 30. This proves that OLSRd is suited to dense networks.

When we look at the diagram in Figure 5.4, the trend is that the number of packets, for each of the OLSRd variants, varies up and down when the number of nodes is between 30 and 100. In addition we see that the number of packets is almost equal for each of the protocol variants. The differences are as expected: Our name extension and standard OLSRd has almost equal number of packets captured, while the plug-in versions have small higher values. An exception is when the number of nodes is 90. In this scenario, the standard OLSRd version has a higher value than the others. This is a result that could be explained from arbitrary factors, such as hardware load and how NEMAN acts at the time when that experiment was measured. This must be taken into consideration when we look at the measured overhead.

If we compare the number of packets captured in the area of 1000x800 (Figure 5.4) with 500x400 in Figure 5.2, we can see that the number of packets is more or less equal for each scenario. The greatest differences are when the number of nodes is 10 and 20. In the other scenarios, the packets are a bit higher when the area is 1000x800 than they are in the area of 500x400.

The measured overhead is from this, expected to be higher in the 1000x800 area, than with an area of 500x400. We further expect that our solution has higher values of overhead, than the standard version of OLSRd. The plug-in version, when the emission interval is five seconds

are expected to be nearly equal with our solution. We also expect that the overhead is higher with the plug-in version when the emission interval is two seconds.

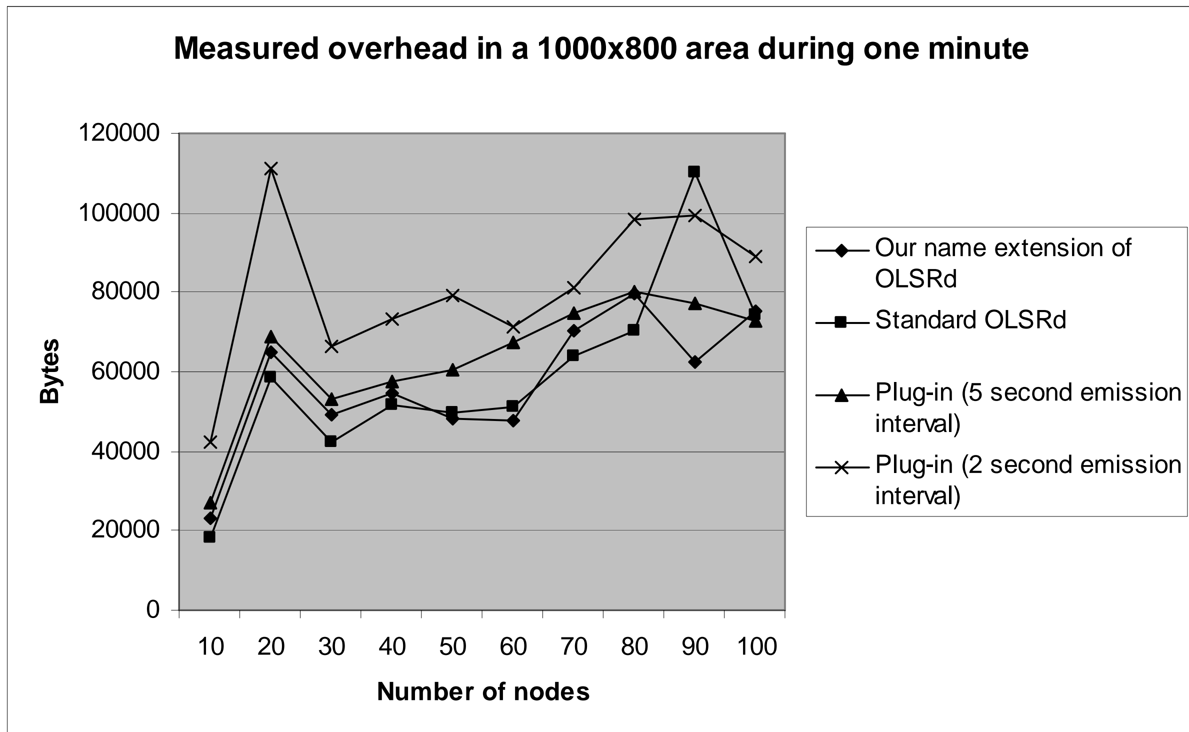

Figure 5.5: The measured overhead where nodes exist in a 1000x800 area.

The diagram in Figure 5.5 shows the overhead in bytes at the Y-axis, while the number of nodes is the elements on the X-axis. Results in this diagram look like our expectations: The minimum value of the overhead almost always belongs to the standard OLSRd version. The exceptions are when the number of nodes is 50 and 60. In addition we have an exception when the number of nodes is 90. The result in this scenario must specially be read together with the diagram in Figure 5.4. As we see there, the number of packets is higher than any of the other protocol variants, and is also affecting the result of the overhead.

If we look more at the diagram in Figure 5.5, we can see that the overhead is almost equal between our name extension and the plug-in version with five second emission interval. This is an expected result.

Some unexpected values can be seen when the number of nodes is 50, 60 and 90. In these scenarios, our implementation of a name extension in OLSRd seems to generate less overhead than standard OLSRd. The only explanation of this behaviour is arbitrary environmental factors, such as hardware limitations.

The plug-in version with 2 second emission interval is the OLSRd variant that always generates the highest value of overhead (Except from when the number of nodes is 90). This is an expected behaviour.

5.4 Measured overhead when hostname in OLSRd control message increases

We discussed in Section 5.1 an input parameter of increasing the number of characters in the hostname to measure overhead. The expected result of this test is that the overhead increases proportional with the number of characters that are in the hostname.

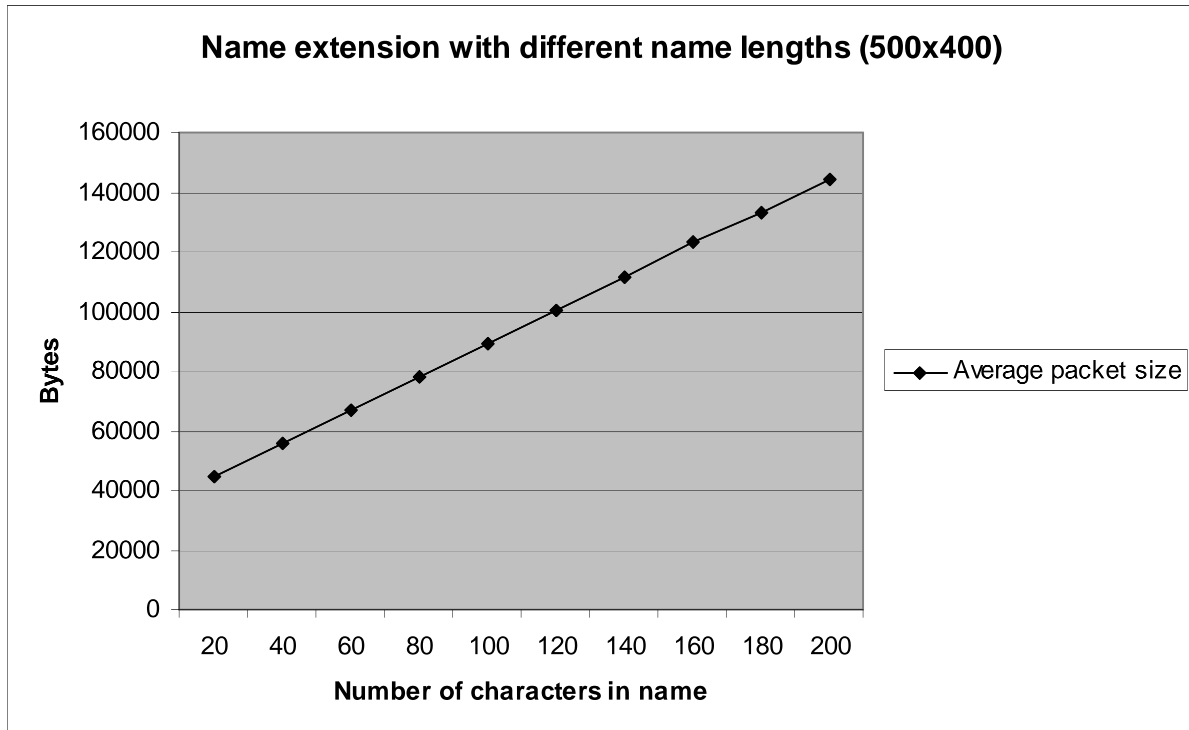

Figure 5.6: This diagram shows how the total amount of overhead increases with number of characters in the hostname.

We can see from Figure 5.6 that the graph is linear. This result coincides with our expectation. If we increase the hostnames on every host in a network with the same number of characters, the result can be described by a mathematical formula. In a given scenario where the number of nodes, N, is constant, the overhead, O, can be expressed as: $O = c * N + P$, where c is the number of characters in the hostname, and P is the payload from other information that flows on the network. An assumption of this formula is that each host has the same number characters in its hostname.

5.5 Summary

This chapter has evaluated our implementation with two metrics: overhead and corresponding number of packets generated. An experiment design was made in order to measure our metrics in different ways. We divided the results into the two geographical area sizes that were defines as one of the input parameters in our design of the experiments. Results shown were in some experiments expected while unexpected in others. The most unexpected result was that the overhead was not proportional with the number of nodes. The reason is the behaviour of MPR in OLSR.

We saw some unexpected results that are difficult to explain from theory. For instance, it is difficult to explain why our solution in some scenarios generates less number of packets and less overhead in bytes than standard OLSRd. This behaviour can be explained from arbitrary factors such as hardware limitations and how NEMAN acts.

Otherwise, the trend in our results is that our name extension in OLSRd generated a bit more overhead than standard OLSRd. Compared to the plug-in versions did our name extension of OLSRd generate equal or less amount of overhead. We also measured the overhead when we increased the hostname on each node in a given scenario. It was proven that the overhead is proportional (linear) to the number of characters.

When we look at our claims we declared in Section 3.1, we declared that we wanted to keep a balance between efficiency at a maximum and overhead at a minimum. This analysis over the problem has proven that this claim has been fulfilled, since our solution has equal or better values compared to the plug-in version.

6 Conclusion

This chapter summarizes this report. It discusses if we have fulfilled our claims that we defined in Section 1.1.4. Further work on the subject that this thesis reflects are discussed at the end of this chapter.

6.1 Summary of the report

In the introduction of this thesis we were introduced to problems that we meet when we try to use the *Domain Name System* (DNS) in MANETs. Because of unstable topology, it can not work, because DNS is bound to a client-server relationship.

In order to serve MANETs a reliable name resolution service, we need to redesign the basic ideas around DNS that are designed for the hierarchal Internet. We defined a set of claims that our solution must meet.

A literature study was done in order to find related work on this study. Solutions that other scientists have done were found, but none of them fulfilled all of our claims. The literature study told us that the MIDAS working group has written a design document, where the OLSR protocol makes the base of a name resolution system. The idea was based on extensions of the existing control messages and data sets in OLSR.

This was our base for a design on a name resolution system. The design was implemented and tested to see if the functionality worked as intended.

Our implementation was evaluated on two metrics: Overhead and number of corresponding captured packets. It was also compared to other protocols variants to see if the overhead metric had values that our solution could accept. The result was that our name extension of OLSRd did it equal or better than the existing OLSRd name service plug-in, based on the overhead metric.

6.2 Claims versus our solution

In this section we discuss whether we have fulfilled our claims that were defined in Section 1.1.4. We list each claim and argue about the achievements with our solution of name resolution system:

Distributed service:
Our name resolution service must be a distributed service, since we design the service for MANETs. The problem with the existing DNS is that it is not a distributed service. It is bound to a hierarchal client-server relationship that can not work in MANETs because of its dynamic topology. Since our name resolution system is based on a routing protocol that is designed for OLSRd, we can say it is a distributed service. The OLSR protocol provides distribution of IP address so that every node can maintain a routing table to all possible destinations on the MANET. We have extended OLSR to also distribute names, so it can be reached to all nodes

on a MANET with its corresponding IP address. Hence, we can conclude that our name resolution service is a distributed service.

Reduce overhead:
Extending an existing protocol is not a way of reducing overhead. There is always a question of what we want to pay in order to get desired information. Our solution is based on an existing version of OLSRd. We extend some of its control message to serve node a way of name resolution. An alternative way of doing this has been implemented as a plug-in. The name service plug-in floods name messages through the MPR flooding scheme with OLSRd. Default, the value of the emission interval of name messages is 120 seconds. This is a disadvantage because of two reasons: First, it is not an effective solution. Second, if we have a MANET with high mobility the names may be invalid in a long time. In our name resolution system we try to meet up with these disadvantages, where the main goal is to get a more efficient solution. The efficiency aspect is one of the declared claims that are discussed next. From the results in the evaluation chapter in this report, our name resolution system performed equal or better than the plug-in. In order to justify the overhead value the emission interval of plug-in name messages was respectively set to two and five seconds. One of the goals with our design in Section 3.1 was to keep a balance of overhead at a minimum and efficiency at a maximum. Though, when trying reaching the efficiency at a maximum, the results from our evaluation can conclude that we have reduced the overhead at a minimum value.

Efficiency:
The hostnames that are distributed with our name resolution system are transported with OLSR HELLO and TC messages. Based on this a name entry is simultaneously created in the name table when a new node on the network is discovered by OLSRd and there is a route to it. This fulfils the efficiency claim.

Application layer transparency:
We claim that existing software should be able to work on top of our name resolution service without any modifications. This is solved by maintaining the hosts file in the operating system. This `hosts` file exist both on Linux/Unix and Windows platforms. When an application want to resolve a name, it ask the operating system which calls the `gethostbyname()` function. `gethostbyname()` first tries to find a match by parsing the `hosts` file, before any DNS servers are queried. The application is not affected by the maintenance of the hosts file hence we have achieved application layer transparency.

Independency:
Our solution is built on the OLSR routing protocol. This routing protocol acts independently of any other technology. Hence, this claim is fulfilled.

6.3 Further work

This section investigates future work that can be done in order to get a better name resolution service based on OLSRd, than the scope of this thesis was to solve.

Investigating unexpected results
As we discussed in Section 5.5, some of the values in our experiments were unexpected with respect to theory. The first step in further work on a name resolution, based on what is described in thesis, is to try to go more in depth in these results in order to explain this behaviour of OLSRd.

Compress hostname:
First, there are mechanisms that can reduce overhead even more than the MPR flooding scheme serves us. Storing hostnames into OLSRd control messages makes them much bigger than they were before. The header size of a HELLO message is 16 bytes. If we assume that a host has a hostname of 16 characters, the HELLO message becomes twice as large than it was before the extension. TC messages grow even more, since the new TC messages also add the neighbour's name together with the IP address. A mechanism that can reduce the payload is to use a compressing algorithm of hostname in order to make the names shorter in the control messages. Such algorithms can be a Huffman algorithm or similar, that is able to compress text strings.

Name resolution with multiple interfaces:
Because of the test environment we were not able to investigate the possibilities of using multiple interfaces in OLSRd. In order to make our name resolution service supporting nodes with more than one interface, MID messages must also be extended like HELLO messages and TC messages.

Compatibility with other versions of OLSRd:
Problems are met, if we connect nodes that use our name extension OLSRd together with nodes that use the standard version of OLSRd. Our name resolution is not compatible with other versions of OLSRd. I.e. if a standard version of the HELLO messages is received with our extended OLSRd, the result is unpredictable. An assumption in Section 3.1 says that compatibility between our solution and the standard version or older versions of OLSR can not be guaranteed. Since building and parsing the new control messages have to be done differently with respect to the OLSRd version, it might be an impossible task to make them work together. An alternative is that if it does not understand the structure of the message, then it should be silently discarded.

Cooperation with occasionally discovered DNS servers:
An interesting approach is to look at the possibilities of making our name resulting service in OLSRd working with surrounding DNS servers that a node might discover occasionally. If a node suddenly discovers that it is connected to the Internet, it should be able to use the DNS to resolve names. In addition, it could be interesting to see if it is possible to upload the information that this node knows about the MANET to a DNS server. These aspects will create a hybrid technology of our name resolution system based on OLSRd and DNS.

Appendix A

This report is attached with a CD containing source code, different tools used and a copy of the report. The CD has the following directories:

- **olsrd-0.4.10 source code** – The source code of the original OLSR daemon. The C source files are located under `src`, while the nameservice plug-in is located under `lib/nameservice/`
- **olsrd-0.4.10-with_naming** – The source code of the OLSR daemon with our name extension. The C source files are located under `src`, while the *Nameservice* plug-in is located under `lib/nameservice/`
- **NEMAN_scenarios** – The scenario files used under testing the functionality (sub-directory `functionallity_test_scenarios`) and evaluation measurements (sub-directory `eval_scenarios`).
- **test_results** – The test results we got from the experiments in Chapter 5. They are divided into sub-directories with these corresponding tests:
  - Standard OLSRd – Sub-directory: `without_name`
  - Our extension of OLSRd – Sub-directory: `with_name`
  - The plug-in with 5 second emission interval – Sub-directory: `plug-in`
  - The plug-in with 2 second emission interval – Sub-directory: `plug-in2`
- **tools** – Different tools that are programs or scripts that was necessary in order to be able to perform testing and measurements:
  - `check-neighbours.sh` – The script that uses the shared library `libsocktap.so` to convert the ports on the virtual interfaces.
  - `libsocktap.so` – the binary version of the shared library.
  - `find.sh` – a script to find result directories (this script is modified in the `name_length` directory). This script is also located under the sub-directories of each experiment.
  - `extract_results.sh` – a script to extract all results from `tcpdump` files of an experiment (this script is modified in the `name_length` directory). This script is also located under the sub-directories of each experiment. The result is a text file for each scenario where the result is stored.
  - `get_all_results.sh` – a script to gather all result files that `extract_results.sh` created from an experiment with all its scenarios. The experiments are divided into the areas of 500x400 and 1000x800. The result is a text file where all results are stored. This script is also located under the sub-directories (500x400 and 1000x800) of each experiment. (This script is modified in the `name_length` directory.)
  - `measure.c` – a C program that is used for performing automatic tests with `tcpdump`
- **report** – A copy of the report in PDF (.pdf) format and Word (.doc) format.

Bibliography

[1] P. Mockapetris, "Domain Names - Concepts and Facilities (RFC 1034)," 1987.

[2] Laura Marie Feeney, "A Taxonomy for Routing Protocols in Mobile Ad Hoc Networks," *SICS Technical Report T99/07*, 1999.

[3] Paal Engelstad, Do Van Thanh, and Geir Egeland, "Name Resolution in On-Demand MANET and over External IP Networks," presented at Communications, 2003. ICC '03. IEEE International Conference on, 2003.

[4] Charles E. Perkins, *AD HOC Networking*: Addison-Wesley, 2001.

[5] Mario Gerla, Taek Jin Kwon, and Guangyu Pei, "On Demand Routing in Large Ad Hoc Wireless Networks with Passive Clustering," presented at Wireless Communications and Networking Conference, 2000. WCNC, 2000.

[6] Kaixin Xu and Mario Gerla, "A Heterogeneous Routing Protocol Based On A New Stable Clustering Scheme," presented at MILCOM 2002, 2002.

[7] Xiaoyan Hong, Jun Liu, and Randy Smith, "Distributed Naming System for Mobile Ad-Hoc Networks," presented at the 2005 International Conference on Wireless Networks (ICWN-05), 2005.

[8] MIDAS, "http://www.ist-midas.org/," 2007. Last visited 30.10.2007.

[9] Joe Gorman, "The MIDAS Project: Interworking and Data Sharing," *Interworking 2006 Santiago, Chile 15-17 January 2007*, 2007.

[10] The MIDAS working group, "D2.1 Annex to MIDAS Project Deliverable: Design of Oppurtunistic Connectivity Service," 2006.

[11] Thomas Heide Clausen and Philippe Jacquet, "Optimized Link State Routing (RFC 3626)," 2003.

[12] Andreas Tønnesen, "http://www.olsr.org," 2007. Last visited 10.12.07.

[13] Andreas Tønnesen, "Implementing and extending the Optimized Link State Routing Protocol," in *UniK University Graduate Center*, vol. Master thesis: University of Oslo, 2004.

[14] Matija Pužar and Thomas Plagemann, "NEMAN: A Network Emulator for Mobile Ad-Hoc Networks," 2005.

[15] Internet Systems Consortium Inc., "http://www.isc.org/index.pl?/sw/bind/index.php," 2007. Last visited 29.10.07.