Implementing and extending the Optimized Link State Routing Protocol

Andreas Tønnesen
UniK University Graduate Center
University of Oslo
andreto@olsr.org
Abstract

A MANET is a multi-hop ad-hoc wireless network where nodes can move arbitrary in the topology. The network has no given infrastructure and can be set up quickly in any environment. The Optimized Link State Routing (OLSR) protocol is a route management protocol for such mobile ad hoc networks. This study presents the work of implementing the OLSR routing protocol. The implementation is done in a modular fashion, allowing for the use of external plugins. Also, this study analyzes certain extensions to the protocol done in relation to the implementation, including Internet connectivity, security and auto-configuration. More technical implementation designs are also covered.
Preface

This thesis is written as a part of my master degree in Computer Science at the University of Oslo, Institute of Informatics. The thesis is written at UniK University Graduation Center.

I have always had a deep interest for how stuff works under the hood, and this especially applies to computer networks. Implementing, extending and experimenting with a MANET routing protocol has been the perfect assignment for me. It has been a lot of challenging, hard work, but at the same time it has been very rewarding and fun.

Through my work on this master, I have learned a lot about routing protocols and low-level network programming on the GNU/Linux platform. I have been allowed to work freely and to pursuit any new idea I have come up with. For giving me such freedom in my work, and for his guidance, I would like to thank Andreas Hafslund. I would also like to thank Paal Engelstad, Jon E. Andersson and Roar B. Rotvik who I have been working with on the extensions of OLSR. My thanks also goes to Prof. Øyvind Kure and Knut Øvsthus.

Also a special thanks to my wife Margareth, for her patience with me spending endless hours hacking computers.

Andreas Tønnesen,
Oslo, August 1st, 2004
## Contents

1 Introduction 2  
  1.1 Implementation work 2  
  1.2 Chapter overview 2  

2 Mobile ad-hoc networks 4  
  2.1 Ad-hoc networks 4  
  2.1.1 Wireless communication 4  
  2.1.2 Traditional IP routing 5  
  2.1.3 The MANET IETF working group 6  
  2.1.4 MANET and Mobile IP 6  
  2.2 MANET routing protocols 6  
  2.2.1 Reactive protocols - AODV 6  
  2.2.2 Proactive protocols - OLSR 8  
  2.2.3 Hybrids - ZRP 9  
  2.2.4 Overview 10  

3 OLSR - core functionality 11  
  3.1 Node addressing 11  
  3.2 Information repositories 11  
  3.2.1 Timeouts 12  
  3.3 Control traffic 12  
  3.3.1 Packet format 12  
  3.3.2 Message types 13  
  3.4 Multipoint Relaying 14  
  3.4.1 Multipoint relaying 14  
  3.4.2 Forwarding OLSR traffic 15  
  3.4.3 Link set optimization 16  
  3.5 Using multiple interfaces 17  
  3.6 Neighbor discovery 18  
  3.6.1 Link sensing 18  
  3.6.2 Neighbor detection 20  
  3.6.3 Two hop neighbor detection 20  
  3.6.4 MPR Selector detection 20  
  3.7 Link state declaration 21  
  3.7.1 Advertised Neighbor Sequence Number 21  
  3.7.2 Populating the topology set 22  
  3.8 Route Calculation 22  
  3.9 Overview 22
4 Auxiliary functionality

4.1 External access - HNA
4.1.1 HNA - message format
4.1.2 HNA - message processing
4.1.3 HNA - route calculation
4.2 Link-layer notifications
4.3 Link hysteresis
4.4 TC redundancy
4.5 MPR redundancy

5 Implementation background

5.1 Technical
5.2 History
5.2.1 olsr.org
5.3 Implementation overview
5.4 About the source-code
5.5 Portability
5.6 Tools used
5.7 Real life usage
5.7.1 The Wizards of Os experiment
5.7.2 Other usage
5.8 Related work
5.8.1 Other protocols

6 Implementing core functionality

6.1 General implementation issues
6.1.1 Output
6.1.2 Configuration file
6.1.3 Packets and OLSR definitions
6.1.4 Communicating with the kernel
6.1.5 Configuring network interfaces
6.1.6 Updating the kernel routing table
6.2 Control traffic
6.2.1 Broadcasting OLSR packets
6.2.2 IP address filtering
6.2.3 Directing outgoing traffic
6.3 IPv6
6.3.1 Olsrd IP data-type
6.3.2 Configuring interfaces using IPv6
6.3.3 IPv6 multicast
6.3.4 IPv6 sockets
6.3.5 Routing table update
6.4 The socket parser
6.5 The packet parser
6.6 Information repositories
6.6.1 Timers
6.6.2 List structures
6.6.3 Detecting changes
6.7 The scheduler
6.7.1 Data integrity - mutexes
6.8 Maintaining state
6.8.1 Sequence numbers
6.8.2 Duplicate set
6.8.3 Link sensing
Appendices

A  olsrd configuration file 117
B  olsrd manual page 121
C  olsr_protocol.h 124
D  The UniK - OLSR plugin library 132
E  Secure extension to the OLSR protocol 137
F  IP Address Autoconfiguration For Mobile Ad-Hoc Networks 142
G  Internet Connectivity for Multi-Homed Proactive Ad Hoc Networks 150
# List of Figures

2.1 A traditional base station scheme compared to an ah-hoc multi-hop network. ............................................. 5  
2.2 A scenario that can lead to the “counting to infinity” problem. ................................................................. 7  
2.3 A possible path for a route reply if A wishes to find a route to J. .............................................................. 8  
2.4 A ZRP scenario showing the zones of node A and node J using a r value of 2. Within the zones a pro-active routing protocol is used while a re-active protocol is used between zones. .................. 9  
2.5 The different components of the Zone Routing Protocol. ................................................................. 9  
3.1 The generic OLSR packet. ............................................................................................................................ 13  
3.2 Flooding a packet in a wireless multi-hop network. The arrows show all transmissions. ....................... 14  
3.3 Flooding a packet in a wireless multi-hop network from the center node using MPRs(black). The arrows show all transmissions. .................................................. 14  
3.4 Flooding a packet in a wireless multihop network. The arrows show the way information is passed, not all transmissions. ........................................ 15  
3.5 Flooding a packet in a wireless multihop network from the center node using MPRs(black). The arrows show the way information is passed, not all transmissions. ........................................ 15  
3.6 Node A has selected the black nodes as its MPRs. ..................................................................................... 16  
3.7 An OLSR routed network. The gray nodes are chosen as MPRs by one or more neighbor. .................. 17  
3.8 The OLSR MID message. ......................................................................................................................... 17  
3.9 A typical neighbor discovery session using HELLO messages. ............................................................ 18  
3.10 The OLSR HELLO message. .................................................................................................................. 19  
3.11 the 8 bit Link Code field. ........................................................................................................................ 19  
3.12 Nodes A and B runs OLSR on multiple interfaces. B uses the address of b1 as its main address. Nodes D and C runs on single interfaces(d1 and c1). ..................................................... 19  
3.13 the OLSR Topology Control message format. .................................................................................... 21  
3.14 OLSR information repositories relation overview. ........................................................................... 23  
4.1 A node in the MANET announces itself as a gateway to Internet for the nodes in the MANET. This is done by HNA messaging. ................................................................. 25  
4.2 The Host and Network Association message. ...................................................................................... 25  
4.3 A would add B as its Internet gateway. ................................................................................................. 26  
4.4 The link is set to symmetric at time A and set to asymmetric at time B. .............................................. 27  
4.5 The center node has chosen the black nodes as MPRs using a MPR_COVERAGE parameter of 1. .... 28  
4.6 The center node has chosen the black nodes as MPRs using a MPR_COVERAGE parameter of 2. Not all two hop neighbors are covered by two MPRs since they are only reachable through one one hop neighbor. ......................................................... 28  
5.1 olsrd main overview. ............................................................................................................................... 31  
6.1 The socket parser. ................................................................................................................................. 41  
6.2 The socket parser in its initial state passing all OLSR traffic to the packet parser. .................................. 41  
6.3 Overview of the packet-parser. ............................................................................................................. 42  
6.4 A linked list. It can be one-way linked only following the full lines or double-linked following the dashed lines as well. The last element may point to the first or some predefined value like NULL. ................................................................. 43
6.5 A example of a balanced binary-tree. A search will never exceed the depth of the tree in number of
seeks. In this example, a lookup will never result in more than 5 seeks. .............................. 44
6.6 Part of a B-tree. All data is actually stored in the leaf nodes. .............................................. 44
6.7 The olsrd generic data index. A static list of indexes that is used for hashing, where each entry is the
root element in a double-linked list. ................................................................. 45
6.8 Overview of the olsrd scheduler. When the scheduler polls it records a timestamp \( t_1 \). When all
operations for that poll is done, a second timestamp \( t_2 \) is recorded. The scheduler then sleeps for
\( \text{pollinterval} \backsailed (t_2 - t_1) \). ................................................................. 47
6.9 The parts of the code protected by the mutex. ................................................................. 48
6.10 The connection between the one- and two-hop neighbor sets. All lists are double linked with static
root elements. ................................................................................................. 50
6.11 A situation where all nodes will register redundant two hop neighbors. .............................. 50
6.12 A situation where B registers the addresses of A as two separate nodes before receiving a MID from
A if A uses interface1’s address as its main address. .............................................. 52
6.13 Node A is running olsrd on two network interfaces. Interface 1 is wireless while interface 2 is an
Ethernet interface. ......................................................................................... 54
7.1 A situation where node A should keep a stable Internet gateway by either using G1 or G2, and not
alter between them. ....................................................................................... 57
7.2 A IPv6 HNA packet containing one network announcement containing a netmask compared to one
containing the prefix x length. This prefix x approach could also be used on IPv4 HNA messages. .... 58
7.3 The IW spy registration based on an IP address. .............................................................. 61
8.1 Example of how a plugin intercepts an application and adds its own program flow. ............ 63
8.2 An example of how a plugin can enable the OLSR daemon to work as a relay for broadcasting. The
Local application and the plugin communicate using interprocess communication. ................ 64
8.3 A plugin can manipulate virtually every part of the olsr daemon. .................................... 64
8.4 The plugin initialization process. ................................................................................... 66
8.5 The message format used by the power-status plugin. This data is sent as the data part of a regular
OLSR message. ............................................................................................. 67
9.1 The main screen displays a list of known nodes and information about them. ..................... 70
9.2 The packet screen offers OLSR packet sniffing. ............................................................ 70
9.3 The route screen displays all active OLSR routes. ......................................................... 71
9.4 The settings screen displays information about the local nodes settings. ....................... 72
9.5 The internal design of the olsrd GUI client. .................................................................... 72
10.1 The test setup used for local usage testing. The links from A to B and C are altering (the link alters
between A->B and A->C every 45 seconds). Node G announces Internet connectivity. ............ 75
10.2 A topology setup used for OLSR overhead registration. ............................................. 79
10.3 Overhead of OLSR control traffic at the test network set up at WoS3. ......................... 80
11.1 The scenario used for the real life implementation of the attack described in section 11.2.1. Node B
has Internet connectivity and declares itself as a HNA gateway to the Internet. ................... 84
11.2 What node B in the scenario from figure 11.1 sees when requesting http://www.slashdot.org in its
local web browser. This is obviously a forged version of the real slashdot page. Note the hostname in
the address bar. ............................................................................................... 85
11.3 The basic signature message as used in the implementation. This is sent as the message body of an
olsr message. ................................................................................................. 85
11.4 The timestamp exchange challenge message. This is sent as the message body of an olsr message. .. 85
11.5 The timestamp exchange challenge-response message. This is sent as the message body of an olsr
message. ................................................................................................. 85
11.6 The timestamp exchange response-response message. This is sent as the message body of an olsr
message. ................................................................................................. 85
11.7 A replay attack on OLSR. The attacker records signed messages and plays them back on a later stage or to nodes that has no record of the sequence numbers of some of the recorded messages. 87

11.8 A illustration of the design of the secure plugin as related to olsrd. 89

12.1 The basic layout of the PAA solution. 92
12.2 The generic PAA packet. 93
12.3 The Forward Request message. 93
12.4 The Address Request message. 93
12.5 The Address Response message. 93
12.6 The Address Test message. 93
12.7 The Address Test Confirmation message. 93
12.8 An example of a conflictless PAA IP allocation session. 94
12.9 The flow of the IPv4 PAA-client. 96
12.10 The PAA-server operations. 97
12.11 The PAA-plugin. 98
12.12 The configured nodes A and B looses connectivity. A then configures a new node with address C. B also configures a new node with address C. A and B then reconnect, merging the networks. This leads to an address conflict amongst the already configured nodes C and C. 99

13.1 Node alters between having connection with Intermediate Node I(IN1) and Intermediate Node 2(IN2) causing Internet traffic to alter between being routed through Gateway1(GW1) and Gateway2(GW2). 101
13.2 Using gateways without NAT. The TCP connection stays up. This solution requires a globally routable address for the MANET host. 102
13.3 The TCP connection breaks just as A moves from IN1 to IN2 because the gateways use NET. 102
13.4 Using unidirectional UDP traffic from the MANET host to an Internet host through NAT gateways. 103
13.5 The TCP connection stays up due to explicit tunneling to one of the gateways. Throughput varies based on the route used. 103
13.6 A scenario where A has set up a tunnel to GW1 and routes all Internet destined traffic through this tunnel. The leftmost figure shows the traffic path when A has a bidirectional link to IN1. The rightmost figure shows the path of the Internet traffic when A only has a bidirectional link to IN2. 105
13.7 Setting up a tunnel to connect two external networks. 106
List of Tables

10.1 The systems used for CPU usage testing. *Node* refers to what node in the scenario depicted in figure 10.1 the system was used as.

10.2 CPU time used by different nodes. The rightmost column shows the percentage of total CPU time used by olsrd.

10.3 Memory used by one instance of olsrd with different topologies. All 2 hop neighbors are only reachable through one 1 hop neighbor. Some nodes are multi-homed.

10.4 Results of OLSR control traffic overhead tests 1-7.

10.5 Results of OLSR control traffic overhead in the scenario illustrated in figure 10.1. Note that close to all nodes in this scenario flood emit TC messages.
Chapter 1

Introduction

“In theory, there is no difference between theory and practice; In practice, there is.”
– Chuck Reid

Recent years mobile communication has increased in usage and popularity. Devices get smaller, batteries live longer and communication protocols get more robust and offer more throughput. Tasks earlier handled by wired communication can now be done using wireless devices and technology, thus giving users the advantage of mobility.

The vision of mobile ad-hoc networking is to support robust and efficient operation in mobile wireless networks, by incorporating routing functionality into mobile nodes. Such networks can have dynamic, sometimes rapidly-changing, multi-hop topologies which are likely composed of relatively bandwidth-constrained wireless links.

1.1 Implementation work

The work this master thesis covers started out as a modification to an existing implementation of a routing protocol for mobile wireless networks. After a while the entire implementation had been rewritten and in many aspects redesigned. It was now implemented to comply to the RFC describing the protocol and to be as modular and extensible as possible.

The work on the master includes an entire implementation of the Optimized Link State Routing protocol. The implementation is very modular in design and is easy to extend through the use of plugins. Several extensions to the routing protocol are also part of the master. All these solutions are described in this thesis.

Due to space limitations, not every part of this thesis will include a full background presentation of the technical aspects of the material. It is assumed that the reader has some basic knowledge of things such as UDP/TCP IP networking and C programming.

1.2 Chapter overview

Mobile ad-hoc networks are introduced in chapter 2. This chapter also introduces the basics of wireless data-communication and other related technology. Three of the routing protocols proposed by the Internet Engineering Task Force (IETF) are also presented in this chapter.

OLSR operation is described in detail in chapters 3 and 4.

The process of implementing the OLSR protocol is described in chapters 5, 6 and 7. An interface to enable
the use of plugins is described in chapter 8 and a Graphical User Interface for the OLSR implementation is presented in chapter 9.

Chapters 11 to 13 presents extensions to the OLSR implementation. In chapter 11 a solution for securing OLSR is described. This solution provides integrity for OLSR control traffic. In chapter 12 an IP address auto-configuration protocol is presented. This protocol allows unconfigured hosts to connect to a MANET and receive an IP address to become a member of the routing domain. In chapter 13 problems and solutions to Internet connectivity in OLSR is discussed. Here, a solution including tunneling of traffic to Internet gateways is presented.

Finally, concluding remarks are made in chapter 14.
Chapter 2

Mobile ad-hoc networks

“Little by little, one travels far.”
- J.R.R. Tolkien

Much wireless technology is based upon the principle of direct point-to-point communication. Popular solutions like Group Standard for Mobile communications (GSM) and Wireless Local Area Network (WLAN) both use an approach where mobile nodes communicate directly with some centralized access point. These types of networks demand centralization for configuration and operation. Contrary to this model is the multi-hop approach. In multi-hop scenarios, nodes can communicate by utilizing other nodes as relays for traffic if the endpoint is out of direct communication range.

Mobile ad-hoc networks, MANET[25], uses the multi-hop model. These are networks that can be set up randomly and on-demand. They should be self-configuring and all nodes can be mobile resulting in a possibly dynamic network topology.

2.1 Ad-hoc networks

Centralized networks, such as GSM, cannot be used in all situations. Significant examples of such scenarios include establishing survivable, efficient, dynamic communication for rescue operations, disaster relief efforts and military networks. Such network scenarios cannot rely on centralized and organized connectivity, they can be conceived as applications of MANETs. The set of applications for MANETs is diverse, ranging from small, static networks that are constrained by power sources, to large-scale, mobile, highly dynamic networks.

To enable multi-hop communication in a distributed manner, all nodes should be able to act as routers for each other (see Figure 2.1). Routes are set up and maintained by a routing protocol. MANET routing protocol design is a complex issue considering the possible rapidly changing topology of such networks.

For route maintenance one has two main approaches in MANETs, reactive and proactive. Reactive routing protocols set up traffic routes on-demand, whilst proactive protocols attempts to dynamically maintain a full understanding of the topology.

2.1.1 Wireless communication

Ad-hoc networks are not restricted to any special hardware. But today such networks are most likely to consist of nodes utilizing so-called WLAN interfaces. These are wireless interfaces operating according to IEEE specifications 802.11a[2], 802.11b[3] or 802.11g[4]. Throughout this document it is assumed that ad-hoc networks consists of links made up by either WLAN or Ethernet[42] interfaces.
IEEE 802.11[12] does not support multi-hop communication by itself. Two modes are defined for communication using WLAN devices:

- Infrastructure mode: The wireless network consists of at least one access point and a set of wireless nodes. This configuration is called a Basic Service Set (BSS). An Extended Service Set (ESS) is a set of two or more BSSs (multiple cells).
- Ad hoc mode: This is a peer-to-peer mode. This configuration is called Independent Basic Service Set (IBSS), and is useful for establishing a network where nodes must be able to communicate directly and without any centralized access point.

The Ad-hoc mode is obviously the mode to use when setting up a MANET, but it lacks one basic requirement: multi-hop. Traffic is only transmitted to neighbors within radio range when using the ad-hoc mode, therefore there is a need for MANET routing protocols to set up and maintain traffic paths.

### 2.1.2 Traditional IP routing

Routing is the primary function of IP. IP datagrams are processed and forwarded by routers which relay traffic through paths set up by various routing protocols. Routing in today's fixed networks is based on network aggregation combined with best matching. TCP/IP hosts use a routing table to maintain knowledge about other IP networks and IP hosts. Networks are identified by using an IP address and a subnet mask, and routes to single hosts are rarely set up. When a packet is to be forwarded, the routing table is consulted and the packet is transmitted on the interface registered with a route containing the best match for the destination. If no network matches are found, a default route is used if one exists.

When configuring a network interface with an IP address, a route to the network the address is a member of is usually registered on the interface automatically. This route is not set up with a gateway (the next
hop along the path to the host) since hosts with addresses within this network are assumed to be reachable directly from this interface. This shows that the traditional IP routing maintains an idea of all hosts within the same subnet being on the same link. This means that all hosts in a subnet are available on a single one-hop network segment, typically via routers or switches. When working on wireless multi-hop networks this is not the case. One needs to redefine the idea of nodes being available “on the link”. In MANETs nodes routes traffic by retransmitting packets on the interface it arrived. This approach breaks with the wired “on-link” way of thinking.

MANET requires a different mindset when it comes to routing. Aggregation is not used in MANETs. all routing is host based. This means that for all destinations within the MANET, a sender has a specific route. In a wired network this is not necessary due to the fact that all nodes in the local network are considered available on the link.

2.1.3 The MANET IETF working group

The Internet Engineering Task Force (IETF) has set down a working group for MANET routing [38]. The purpose of this working group is “to standardize IP routing protocol functionality suitable for wireless routing application within both static and dynamic topologies. The fundamental design issues are that the wireless link interfaces have some unique routing interface characteristics and that node topologies within a wireless routing region may experience increased dynamics, due to motion or other factors.” [38].

A wide diversity of protocols have been proposed, but as of this writing, only three protocols are accepted as experimental Request For Comments (RFC), namely Ad hoc On-Demand Distance Vector (AODV) [54], Optimized Link State Routing (OLSR) [23], and Topology Dissemination Based on Reverse-Path Forwarding (TBRPF) [56]. The Dynamic Source Routing Protocol (DSR) [27] is expected to be accepted as a RFC in a near future.

2.1.4 MANET and Mobile IP

In the Internet community, Mobile IP (MIP) [53] is often mentioned when it comes to routing support for mobile hosts. This is technology to support nomadic host roaming, where a roaming host may be connected through various means to the Internet other than its well known fixed-address domain space. The host may be directly physically connected to the fixed network on a foreign subnet, or be connected via a wireless link, dial-up line, etc. Supporting this form of host mobility requires address management, protocol interoperability enhancements and the like, but core network functions such as hop-by-hop routing still presently rely upon pre-existing routing protocols operating within the fixed network. In contrast, the goal of mobile ad hoc networking is to extend mobility into the realm of autonomous, mobile, wireless domains, where a set of nodes, which may be combined routers and hosts, themselves form the network routing infrastructure in an ad hoc fashion.

2.2 MANET routing protocols

As mentioned earlier, three proposed protocols have been accepted as experimental RFCs by the IETF. Two of these are presented here. They are both based on well known algorithms from Internet routing. AODV uses the principals from Distance Vector routing (used in RIP [67]) and OLSR uses principals from Link State routing (used in OSPF [66]). A third approach, which combines the strengths of proactive and reactive schemes is also presented. This is called a hybrid protocol.

2.2.1 Reactive protocols - AODV

Reactive protocols seek to set up routes on-demand. If a node wants to initiate communication with a node to which it has no route, the routing protocol will try to establish such a route.
The Ad-Hoc On-Demand Distance Vector routing protocol is described in RFC 3561[54]. The philosophy in AODV, like all reactive protocols, is that topology information is only transmitted by nodes on-demand. When a node wishes to transmit traffic to a host to which it has no route, it will generate a route request (RREQ) message that will be flooded in a limited way to other nodes. This causes control traffic overhead to be dynamic and it will result in an initial delay when initiating such communication. A route is considered found when the RREQ message reaches either the destination itself, or an intermediate node with a valid route entry for the destination. For as long as a route exists between two endpoints, AODV remains passive. When the route becomes invalid or lost, AODV will again issue a request.

AODV avoids the “counting to infinity” problem from the classical distance vector algorithm by using sequence numbers for every route. The counting to infinity problem is the situation where nodes update each other in a loop. Consider nodes A, B, C and D making up a MANET as illustrated in figure 2.2. A is not updated on the fact that its route to D via C is broken. This means that A has a registered route, with a metric of 2, to B. C has registered that the link to D is down, so once node B is updated on the link breakage between C and D, it will calculate the shortest path to D to be via A using a metric of 3. C receives information that B can reach D in 3 hops and updates its metric to 4 hops. A then registers an update in hop-count for its route to D via C and updates the metric to 5. And so they continue to increment the metric in a loop. The way this is avoided in AODV, for the example described, is by A noticing that its route to D is old based on a sequence number. B will then discard the route and C will be the node with the most recent routing information by which A will update its routing table.

AODV defines three types of control messages for route maintenance:

**RREQ** - A route request message is transmitted by a node requiring a route to a node.

As an optimization AODV uses an expanding ring technique when flooding these messages. Every RREQ carries a time to live (TTL) value that states for how many hops this message should be forwarded. This value is set to a predefined value at the first transmission and increased at retransmissions. Retransmissions occur if no replies are received.

Data packets waiting to be transmitted (i.e. the packets that initiated the RREQ) should be buffered locally and transmitted by a FIFO principal when a route is set.

**RREP** - A route reply message is unicasted back to the originator of a RREQ if the receiver is either the node using the requested address, or it has a valid route to the requested address. The reason one can unicast the message back, is that every route forwarding a RREQ caches a route back to the originator.

**RERR** - Nodes monitor the link status of next hops in active routes. When a link breakage in an active route is detected, a RERR message is used to notify other nodes of the loss of the link. In order to enable this reporting mechanism, each node keeps a “precursor list”, containing the IP address for each its neighbors that are likely to use it as a next hop towards each destination.

Figure 2.3 illustrates an AODV route lookup session. Node A wishes to initiate traffic to node J for which it has no route. A broadcasts a RREQ which is flooded to all nodes in the network. When this request is
forwarded to J from H, J generates a RREP. This RREP is then unicast back to A using the cached entries in nodes H, G and D.

### 2.2.2 Proactive protocols - OLSR

A proactive approach to MANET routing seeks to maintain a constantly updated topology understanding. The whole network should, in theory, be known to all nodes. This results in a constant overhead of routing traffic, but no initial delay in communication.

The Optimized Link State routing (OLSR) is described in RFC3626 [23]. It is a table-driven pro-active protocol. As the name suggests, it uses the link-state scheme in an optimized manner to diffuse topology information. In a classic link-state algorithm, link-state information is flooded throughout the network. OLSR uses this approach as well, but since the protocol runs in wireless multi-hop scenarios the message flooding in OLSR is optimized to preserve bandwidth. The optimization is based on a technique called MultiPoint Relaying. The MPR technique is studied further in chapter 3.

Being a table-driven protocol, OLSR operation mainly consists of updating and maintaining information in a variety of tables. The data in these tables is based on received control traffic, and control traffic is generated based on information retrieved from these tables. The route calculation itself is also driven by the tables.

OLSR defines three basic types of control messages all of which will be studied in detail in chapter 3:

- **HELLO** - **HELLO** messages are transmitted to all neighbors. These messages are used for neighbor sensing and MPR calculation.

- **TC** - **Topology Control** messages are the link state signaling done by OLSR. This messaging is optimized in several ways using MPRs.

- **MID** - **Multiple Interface Declaration** messages are transmitted by nodes running OLSR on more than one interface. These messages lists all IP addresses used by a node.

OLSR is further studied in chapters 3 and 4.
2.2.3 Hybrids - ZRP

Hybrid protocols seek to combine the proactive and reactive approaches. An example of such a protocol is the Zone Routing Protocol (ZRP)\cite{39}. ZRP divides the topology into zones and seek to utilize different routing protocols within and between the zones based on the weaknesses and strengths of these protocols. ZRP is totally modular, meaning that any routing protocol can be used within and between zones. The size of the zones is defined by a parameter $r$ describing the radius in hops. Figure 2.4 illustrates a ZRP scenario with $r$ set to 1. Intra-zone routing is done by a proactive protocol since these protocols keep an up to date view of the zone topology, which results in no initial delay when communicating with nodes within the zone. Inter-zone routing is done by a reactive protocol. This eliminates the need for nodes to keep a proactive fresh state of the entire network.

ZRP defines a technique called the Bordercast Resolution Protocol (BRP) to control traffic between zones. If a node has no route to a destination provided by the proactive inter-zone routing, BRP is used to spread the reactive route request. Figure 2.5 illustrates the different components of ZRP.
2.2.4 Overview

The three routing protocols AODV, OLSR and ZRP have been introduced in this section. AODV and OLSR are both accepted as experimental RFCs by the IETF and they are probably the two most popular MANET routing protocols at the current time. Much research and work is being done using these protocols. The hybrid ZRP protocol has not gained that much popularity. The protocol is actually more of a framework than a routing protocol, and it relies on well defined and robust routing protocols to be utilized in and between the zones. The latest ZRP Internet draft expired January 2003, but work is still said to be done by the authors and others. The need for solutions like ZRP might arise when the basic protocols are well tested and their limitations have been proven.
Chapter 3

OLSR - core functionality

“Many attempts to communicate are nullified by saying too much.”
– Robert Greenleaf

The Optimized Link State Routing Protocol (OLSR) is developed for mobile ad hoc networks. The protocol is documented in the experimental Request For Comment (RFC) 3626. OLSR is table-driven and pro-active and utilizes an optimization called Multipoint Relaying for control traffic flooding.

RFC3626 modularizes OLSR into core functionality, which is always required for the protocol to operate, and a set of auxiliary functions. This chapter presents the core functionality of OLSR. The core functionality specifies, in its own right, a protocol able to provide routing in a stand-alone MANET. Each auxiliary function provides additional functionality, which may be applicable in specific scenarios, e.g., in case a node is providing connectivity between the MANET and another routing domain. All auxiliary functions are compatible, to the extent where any auxiliary function may be implemented with the core. Furthermore, the protocol is said to allow heterogeneous nodes, i.e., nodes which implement different subsets of the auxiliary functions, to coexist in the network. As we shall later, this is not the case for all auxiliary functions.

It is important to understand that OLSR does not route traffic. It is not in any way responsible for the actual process of routing traffic. OLSR could rather be described as a route maintenance protocol in that it is responsible for maintaining the routing table used for routing packages, but such protocols are usually referred to as routing protocols.

3.1 Node addressing

OLSR uses an IP address as the unique identifier of nodes in the network. As OLSR is designed to be able to operate on nodes using multiple communication interfaces, every node must choose one IP address that is set to be its main address.

OLSR can be used both with IP version 4 (IPv4) [44] and version 6 (IPv6) [28]. In an OLSR context the differences between IPv4 and IPv6 is the size of the IP addresses transmitted in control messages, the minimum size of messages and the address to use as destination for control traffic.

3.2 Information repositories

As a derivate of the classical link-state algorithm, OLSR maintains state by keeping a variety of databases of information. These information repositories are updated upon processing received control messages and the information stored is used when generating such messages. Here follows a brief look at the different information repositories used in core OLSR.
Multiple Interface Association Information Base
This dataset contains information about nodes using more than one communication interface. All interface addresses of such nodes are stored here.

Link Set
This repository is maintained to calculate the state of links to neighbors. This is the only database that operates on non-main-addresses as it works on specific interface-to-interface links.

Neighbor Set
All registered one-hop neighbors are recorded here. The data is dynamically updated based on information in the link set. Both symmetric and asymmetric neighbors are registered.

2-hop Neighbor Set
All nodes, not including the local node, that can be reached via an one-hop neighbor is registered here. Notice that the two hop neighbor set can contain nodes registered in the neighbor set as well.

MPR Set
All MPRs selected by the local node is registered in this repository. The MPR concept is explained in section 3.4.

MPR Selector Set
All neighbors that have selected this node as a MPR are recorded in this repository.

Topology Information Base
This repository contains information of all link-state information received from nodes in the OLSR routing domain.

Duplicate set
This database contains information about recently processed and forwarded messages.

3.2.1 Timeouts
Most information kept in these repositories are registered with a timeout. This is a value indicating for how long the registered information is to be considered valid. This value is set according to a validity time fetched from the message from which the data was last updated. The use of such a distributed validity time allows for individual message emission intervals for all nodes in the network. All database entries are removed when no longer valid according to the registered timeout. Such entries are said to be timed out.

3.3 Control traffic
All OLSR control traffic is to be transmitted over UDP on port 698. This port is assigned to OLSR by the Internet Assigned Numbers Authority (IANA). The RFC states that this traffic is to be broadcasted when using IPv4, but no broadcast address is specified. When using IPv6 broadcast addresses does not exist, so even though it is not specified in the RFC, it is implicit understood that one must use a multicast address in this case.

3.3.1 Packet format
All OLSR traffic is sent in OLSR packets. These packets consist of a OLSR packet header and a body as displayed in fig 3.1.
The fields in the OLSR packet header are:
3.3.2 Message types

The core functionality of OLSR defines three message types, which will all be described in detail later. All core functionality of OLSR is based on processing and generation of these messages.

However, the OLSR protocol packet format allows for a wide variety of custom packets to be transmitted and flooded to the needs of the designer. OLSR will forward unknown packet types according to the default
3.4 Multipoint Relaying

OLSR uses flooding of packets to diffuse topology information throughout the network. Flooding, in its simplest form, means that all nodes retransmit received packets. To avoid loops, a sequence number is usually carried in such packets. This sequence number is registered by receiving nodes to assure that a packet is only retransmitted once. If a node receives a packet with a sequence number lower or equal to the last registered retransmitted packet from the sender, the packet is not retransmitted. On wired networks, other optimizations are usually added such as no retransmission on the interface on which a packet arrived. On a wireless multi-hop network however, it is essential that nodes retransmit packets on the same interface that it arrived, since this is the very nature of wireless multi-hop networks. This again causes every re-transmitter to actually receive a duplicate packet from every symmetric neighbor that re-transmits the packet. A wireless flooding scenario is depicted in Figure 3.2. One can see that every retransmission leads to a reception of the same packet. The originator of the flood could be any node in the figure.

The number of retransmissions using traditional flooding is $n - 1$ where $n$ is the number of nodes in the network. In our case (Figure 3.2) it will be 24. This flooding technique can clearly benefit from some sort of optimization.

3.4.1 Multipoint Relaying

The concept of multipoint relaying is to reduce the number of duplicate retransmissions while forwarding a broadcast packet. This technique restricts the set of nodes retransmitting a packet from all nodes, to a subset of all nodes. The size of this subset depends on the topology of the network.

This is achieved by selecting neighbors as Multipoint relays (MPRs). Every node calculates its own set of MPRs as a subset of its symmetric neighbor nodes chosen so that all 2-hop neighbors can be reached through a MPR. This means that for every node $n$ in the network that can be reached from the local node by at minimum two symmetric hops, there must exist a MPR $m$ so that $n$ has a symmetric link to $m$ and $m$ is a symmetric neighbor of the local node. In the scenario illustrated in Figure 3.6, node A selects the black nodes as MPRs. This way all two hop nodes can be reached through a MPR. Node B will not retransmit.
traffic from a that is to be flooded.

OLSR lets nodes announce their own willingness to act as MPRs for neighbors. 8 levels of willingness are defined from the lowest \texttt{WILL\_NEVER(0)}, which indicates that this node must never be chosen as a MPR, to the highest \texttt{WILL\_ALWAYS(7)}, which indicates that this node should always be chosen as a MPR. The willingness is spread through HELLO messages and this information must be considered when calculating MPRs.

Finding the optimal MPR set has been proved to be a NP-complete problem in [62]. RFC 3626 proposes a rather simple heuristic for MPR calculation. The MPR scheme has been further studied and analyzed in amongst others, [52], [15] and [51]. In this thesis the MPR technique is not further analyzed.

### 3.4.2 Forwarding OLSR traffic

Relaying of messages is what makes flooding in MANETs possible. OLSR specifies a default forwarding algorithm that uses the MPR information to flood packets. One is however free to make one's own rules for custom forwarding of custom messages. But all messages received that carries a type not known by the local node, must be forwarded according to the default forwarding algorithm. The algorithm can be outlined as:

1. If the link on which the message arrived is not considered symmetric, the message is silently discarded. To check the link status the link 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 a MPR, then the message is forwarded. If not, the message is discarded. To check this the MPR selector set is queried.

5. If the message is 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 fact that all received unknown message types are forwarded using this approach makes flooding of special message-types possible even if these message-types are only known to a subset of the nodes.

Figures 3.4 and 3.5 shows the paths information is passed when being spread, first using regular flooding, then using MPR flooding. The number of retransmissions in a MPR scenario highly depends on the network topology and the MPR calculation algorithm. Using the same topology as in fig 3.2, a possible MPR calculation could lead to the black nodes in fig 3.3 being chosen as MPRs by the center node. As one can see, if the center node is to flood a message throughout the network, 4 retransmissions are done using MPR as opposed to 24 using traditional flooding.

The duplicate set

To be able to check if a message has already been retransmitted, a cache of recently processed and forwarded messages is maintained. The data stored is the minimum needed to identify the message. This means that the actual message content is not stored, but rather just originator address, message-type and sequence number. This data is cached for a constant time of 30 seconds suggested in the RFC. Every received message that is processed by the local node is registered in the duplicate set. If the message is forwarded, the duplicate-entry representing this message is updated accordingly, registering on what interfaces the message has been forwarded. Based on querying the duplicate set, a node can then keep track of already processed messages and already forwarded messages on a per interface basis.

Forward jitter

To avoid radio collisions due to synchronized forwarding, a jitter is introduced to the message forwarding. This is a random small time interval for which the message is to be cached in the node before forwarding it. When using forwarding-jitter, piggybacking of messages will often occur since multiple messages that are to be forwarded might arrive within the buffer period. When this happens, messages are stacked within the same OLSR packet.

3.4.3 Link set optimization

Due to the nature of the MPR selection, only nodes which are chosen as MPRs by one or more neighbors, needs to declare their link-state. In fact, these nodes need only declare the MPR selectors in the link-state messages. When this information is flooded to all nodes in the MANET, all nodes will have enough
information to calculate shortest path routes to all hosts. The default OLSR setting is that a node only floods link-state messages if it is chosen as MPR by at least one neighbor, and it only announces its MPR selectors in these messages. In a topology as illustrated in figure 3.7 only the nodes selected as MPRs (gray nodes) by one or more neighbors will transmit link-state messages. One can easily see that this information, in addition to some neighbor-sensing scheme, will be sufficient to create a full understanding of the topology.

3.5 Using multiple interfaces

Nodes participating in an OLSR routing domain can be multi-homed. That means that they can run OLSR on multiple communication interfaces using multiple identifiers. Multiple interface declaration (MID) messages are used to diffuse information about multi-homed nodes. A MID message is essentially just a list of addresses used by interfaces on which a node runs OLSR. The format of the MID message is displayed in figure 3.8. The data is sent as the message part of an OLSR-message included in an OLSR packet as seen in fig 3.1.

Upon receiving a MID message, a node updates its 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 found in the originator field of the OLSR-message header. When adding a route to a node, OLSR will add routes to all addresses of other interfaces on which the remote node runs OLSR, using the same path.
Figure 3.9: A typical neighbor discovery session using HELLO messages.

All nodes running OLSR on more than one interface are generating MID messages on regular intervals. MID messages are to be flooded throughout the network using the default forwarding algorithm.

### 3.6 Neighbor discovery

Obviously, OLSR needs some mechanism to detect neighbors and the state of the communication lines to them. HELLO messages are emitted on a regular interval for this purpose. A very simplified version of a neighbor discovery session using HELLO messages, is displayed in figure 3.9. A first sends an empty HELLO message. B receives this message and registers A as an asymmetric neighbor due to the fact that B can not find its own address in the HELLO message. B then sends a HELLO declaring A as an asymmetric neighbor. When A receives this message it finds its own address in it and therefore sets B as a symmetric neighbor. This time A includes B in the HELLO it sends, and B registeres A as a symmetric neighbor upon reception of the HELLO message.

But HELLO messages serves other purposes as well. They are generated and transmitted to all one-hop neighbors to achieve link-sensing, neighbor-sensing, two-hop neighbor-sensing and MPR selector sensing.

In HELLO messages nodes transmit information about all known links and neighbors. The types of the neighbors are also declared. This includes declaring what MPRs the node has selected. Registered links and neighbors are grouped by the link and neighbor type to optimize byte usage. It is very important to note that HELLO messages are generated on a per interface basis. This is because HELLO messages are used for link sensing, which requires the use of possible non-main-addresses.

The format of the HELLO message can be seen in fig 3.10. This message is included as the body part of an OLSR-message in an OLSR packet as seen in fig 3.1. The 8 byte link-code contains both info about the link to the neighbor and the type of the neighbor. The link type describes the state of the link and the neighbor type describes the state of the neighbor including MPR information. Note that a link can be set as asymmetric while the neighbor is still set as symmetric, if multiple links to the neighbor exists. The 8 bit link code data is ordered as displayed in figure 3.11.

#### 3.6.1 Link sensing

To keep up-to-date information on what links exist between a node and its neighbors, the link set is maintained. In HELLO messages a node emits all information about the links to neighbors from the interface on which the HELLO is transmitted. When declaring links, the IP addresses of the actual interfaces making up the link are used. When declaring the neighbor state of neighbors not reachable on the interface on which the HELLO is transmitted, the main address of the neighbor node is used.

In a scenario like the one depicted in figure 3.12, A would send the following information in its HELLO message on interface a1:

- Its current link and neighbor state for d1.
Figure 3.10: The OLSR HELLO message.

Figure 3.11: the 8 bit Link Code field.

Figure 3.12: Nodes A and B runs OLSR on multiple interfaces. B uses the address of b1 as its main address. Nodes D and C runs on single interfaces(d1 and c1).

- Its current link and neighbor state for c1.
- Its current neighbor state for the main address of node B which is b1.

When building a HELLO to be transmitted on a2, node A will include the following information:

- Its current neighbor state for d1.
- Its current neighbor state for c1.
Its current link and neighbor state for b2.

Upon receiving a HELLO from a neighbor, a node checks to see if the HELLO message contains the IP address of the interface the message was received. The link set is then updated as follows:

- If no link entry exists for the tuple (originating IP, IP of received interface) then such an entry is created. The originating IP is fetched from the IP header of the received packet. Whenever a link entry is created a corresponding neighbor entry is created as well if no such entry exists.
- An asymmetric timer is then updated according to the validity time received. This timer decides for how long the link entry is to be considered asymmetric if the symmetric timer times out.
- If the address of the receiving interface is located in the received HELLO message, the symmetric timer is updated and the status of the link is updated if necessary. The status of the neighbor entry according to this link entry is also updated if necessary.
- Finally the actual holding time for this entry is set to be the maximum of the asymmetric timer and the symmetric timer.

3.6.2 Neighbor detection

Neighbor detection populates the 1-hop neighbor repository and only uses the main addresses of nodes. As seen in the previous section, the neighbor entries are closely related to the link entries. Whenever a link entry is created, the neighbor table is queried for a corresponding neighbor entry. Note that this neighbor entry must be registered on the main address of the node. If no such entry can be located, then a new neighbor entry is created. This means that while a node can have several link-entries describing different links to the same neighbor, only one neighbor entry exists per neighbor.

The status of the neighbor entries is also updated according to changes in the link-set. A neighbor is said to be a symmetric neighbor if there exists at least one link-entry in the link set connecting a local interface to one of the neighbors interfaces where the symmetric timer is not timed out. When a link-entry is deleted, the corresponding neighbor entry is also removed if no other link entries exist for this neighbor.

3.6.3 Two hop neighbor detection

A node also maintains a repository of all nodes reachable via symmetric neighbors. This is the two hop neighbor set. This database is used for MPR calculation.

Upon receiving a HELLO message from a symmetric neighbor, all reported symmetric neighbors, not including addresses belonging to the local node, are added or updated in the two hop neighbor set. Entries in the two hop neighbor set are all based on main addresses, so for all received entries in the HELLO message the MID set is queried for the main address. Note that the two hop neighbors also may contain neighbors reachable by one hop.

3.6.4 MPR Selector detection

The MPR flooding scheme is based on the requirement that nodes have registered what neighbors have chosen them as a MPR. Nodes mark their selected MPR neighbors in HELLO messages by setting the Neighbor Type to be MPR_NEIGH.

Upon receiving a HELLO message, a node checks the announced neighbors in the message for entries matching one of the addresses used by the local node. If an entry has a matching address and the neighbor 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.
3.7 Link state declaration

Link state routing protocols are based on nodes flooding the network with information about their local links. In protocols like ISIS[65] this information is mostly links to subnets, since these protocols are highly based on aggregation of networks. OLSR uses host based flat routing, so the link state emitted describes links to neighbor nodes. This is done using Topology Control(TC) messages. The format of a TC message is shown in figure 3.13.

![Figure 3.13: the OLSR Topology Control message format.](image)

TC messages are flooded using the MPR optimization. This is done on a regular interval, but TC messages are also generated immediately when changes are detected in the MPR selector set. In OLSR the flooding process itself is optimized by the usage of MPRs, but as explained in section 3.4.3, the MPR technique introduces two link-state declaration optimizations as well. As we will see in the Auxiliary functionality chapter, OLSR nodes can also be tuned to send more than just its MPR selector set. One should notice that more robust routing could be achieved by announcing more than the MPR selector set.

The MPR functionality introduces two optimizations to TC messaging:

Size optimization

The size of TC messages is reduced due to the fact that a node may only declare its MPR selectors in TC messages. The factor of this reduction is related to how dense the network topology is. In a topology as shown in figure 3.3 the TC message size of the center node would be reduced to half the size of a “classical” TC message(not including headers). When using IPv6, a simple example like this reduces a net-wide broadcast message with 64 bytes.

Sender optimization

Nodes that has no links to declare usually does not transmit TC messages. The exception here is nodes that just lost their MPR selectors. These nodes are to generate empty TC messages for a given interval to update the nodes in the MANET.

But except from this special case, if only declaring MPR selectors in TC messages, only nodes selected as MPRs will generate TC messages. Such a reduction in actual transmitted messages greatly reduces the overall overhead of control traffic.

3.7.1 Advertised Neighbor Sequence Number

The Advertised Neighbor Sequence Number(ANSN) is a sequence number associated with a nodes advertised neighbor set. This number is however, not increased on every TC generation. The ANSN represents the “freshness” of the information contained in the message. This means that whenever a node detects a
change in its advertised neighbor set the ANSN is increased. Keep in mind that the advertised neighbor set in a node can, as described later, vary from only the MPR selectors to the entire symmetric neighborhood.

### 3.7.2 Populating the topology set

Upon receiving a TC message, the TC repository is updated as follows:

- If no entry is registered in the TC repository 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 TC repository 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 TC repository on the address of the originator with an ANSN equal to the received ANSN, then the validity time of the entry is updated.

### 3.8 Route Calculation

The proposed heuristic for route calculation in RFC3626 is a relatively trivial shortest-path algorithm. It can be outlined as:

1. Add all one hop neighbors registered as symmetric to the routing table with a hop-count of 1.
2. For each symmetric one-hop neighbor, add all two hop neighbors registered on that neighbor that has:
   - Not already been added to the routing table.
   - A symmetric link to the neighbor.
   These entries are added with a hop-count of two and next-hop as the current neighbor.
3. Then, for every added node N in the routing table with hop-count $n = 2$ add all entries from the TC set where:
   - the originator in the TC entry == N
   - the destination has not already been added to the routing table
   New entries are added with a hop-count of $n + 1$ and next-hop as the next-hop registered on Ns routing entry.
4. Increase $n$ with one and do step 3 over until there are no entries in the routing-table with hop-count $== n + 1$
5. For all entries E in the routing table the MID set is queried for address aliases. If such aliases exist an entry is added to the routing table with hop-count set to Es hop-count, and next-hop set to Es next-hop for every alias address.

### 3.9 Overview

We have seen that OLSR functionality can be divided into three main modules: Neighbor sensing, multi-point relaying and link-state flooding. We have also seen that most control traffic is generated based on the set of repositories maintained by OLSR. These datasets are also updated dynamically based on received control messages.
Figure 3.14 displays an overview of the information repositories in OLSR and their relations to message processing, message generation and route calculation. Received HELLO messages trigger updates in the link set which again triggers updates in the neighbor set, which then again triggers recalculation of the MPR set. The 2 hop neighbor set is also updated based on received HELLO messages again triggering a recalculation of the MPR set. Finally the MPR selector set is updated according to information received in HELLO messages. Received TC messages triggers updates in the topology set while the MID set is updated upon receiving MID messages. All received messages will also be registered in the duplicate set if not already registered.

When generating HELLO messages, the link set, neighbor set and MPR set is queried. When generating TC messages, the MPR selector set is queried. When forwarding control traffic, the MPR selector set and the duplicate set is used.

Finally, route calculation is based on information retrieved from the neighbor set, the 2 hop neighbor set, the TC set and the MID set.

Figure 3.14: OLSR information repositories relation overview.
Chapter 4

Auxiliary functionality

“It is good speaking that improves good silence.”
– Dutch proverb

As stated earlier, RFC3626 divides the functionality of OLSR into two sections: core and auxiliary. The core specification is always required for the protocol to operate, while the auxiliary specification provides additional functionality which may be applicable in specific scenarios. In this chapter the auxiliary functionality of OLSR is examined.

4.1 External access - HNA

A MANET can exist isolated, independent of other computer networks. However, the option to connect to other networks should be offered on some level since this is very likely to be a requirement in many situations. Internet access in MANETs has been documented in [20] and [57].

OLSR offers this kind of external connectivity at the routing protocol level. A host can announce itself as a gateway to specific networks using Host and Network Association (HNA) messages. Fig 4.1 illustrates a typical HNA scenario. A node has an Ethernet link on which it has Internet access. This node wishes to offer Internet connectivity to the other nodes in the MANET. This is done by sending HNA messages.

Due to OLSR's default forwarding algorithm, all nodes does not need to support the HNA functionality for HNA messages to be flooded throughout the MANET. But all nodes must support HNA processing and route calculation for the actual HNA routing to work. If a node routes Internet traffic to an intermediate neighbor based on HNA information, the intermediate neighbor must also have set up an Internet route for the traffic to be routed. Therefore, in the general case, the neighbor must support HNA functioning.

4.1.1 HNA - message format

A HNA message is basically just a list of network addresses and netmasks. If a node is to announce itself as a gateway to the 193.156.97.0/24 network, then the node would send the network address 193.156.97.0 and the netmask 255.255.255.0.

4.1.2 HNA - message processing

All HNA data received is registered in the host and network association set. This information repository is kept “fresh” in the same manner as the repositories mentioned in chapter 3. This means that entries are deleted as soon as the validity time expires.
Figure 4.1: A node in the MANET announces itself as a gateway to Internet for the nodes in the MANET. This is done by HNA messaging.

Bits: 1 2 3 4 5 0 1 76 8 9 0 1 76 8 9 0 1 2 3 4 5 76 8 9 0 1 2 3 4 5 2 3

Network address

Netmask

Figure 4.2: The Host and Network Association message.

Entries are registered with the following fields:

- **Gateway** - The main address of the node announcing itself as gateway. This is the originator address of the HNA message.
- **Network** - The network the gateway can route traffic to.
- **Netmask** - The netmask of the network address describing the prefix length.
- **Vtime** - The timestamp specifying for how long this entry is valid.

When a HNA message is received, the HNA data set is updated either by updating an existing entry, or if no entry exists for the gateway address, creating a new entry.

### 4.1.3 HNA - route calculation

HNA routes are recalculated on all changes in the topology or the HNA set. If there exist multiple gateways to the same network, the gateway closest (in hop-count) is chosen.

HNA routing is done hop-by-hop. This means that HNA routes are added with the next-hop neighbor on the route to the gateway as the actual gateway to the HNA announced network. In a scenario like the one illustrated in figure 4.3, A would add the following entry to its routing table:

```
destination: 0.0.0.0/0
gateway: B
metric: 3
```

This means that the actual gateway who announced the external connectivity is not added to the routing table.
4.2 Link-layer notifications

OLSR is designed to work independent of what communication hardware it runs on top of. This means that all link detection and maintenance is done on a relatively high layer of the OSI\textsuperscript{1} model.

One could however imagine taking other variables, gathered from the lower layers, into consideration if available. This could be information from the link-layer which describes the quality of links. RFC3626 allows for usage of this kind of information in link sensing. This information can be used to detect link breakage at an earlier stage than the other OLSR mechanisms and to add robustness to link-sensing by setting a threshold for accepting links.

4.3 Link hysteresis

Webster's dictionary defines \textit{hysteresis} as:

\texttt{hys-ter-e-sis}

A retardation of an effect when the forces acting upon a body are changed (as if from viscosity or internal friction); especially: a lagging in the values of resulting magnetization in a magnetic material (as iron) due to a changing magnetizing force.

Link-hysteresis means that one, based on some function, “slow down” link-sensing. This is to make links robust against bursty loss or transient connectivity between nodes. This means that we are interested in making sure a newly registered link is not just a node passing by at high speed or a node that alternates between residing just outside and just inside radio range. In other words, hysteresis provides a more robust link-sensing at the cost of more delay before establishing links.

The strategy suggested in RFC3626 is based upon two functions, one \textit{stability} rule and one \textit{instability} rule, in addition to two link-quality thresholds. Hysteresis requires one to maintain a link-quality value for every link. This value will trig the update of link status when it crosses one of the defined thresholds, and this provides the actual retardation effect. When using hysteresis, the status of links are only changed under two conditions:

- A link is set to be symmetric if it is currently set to asymmetric and the link-quality of the link is bigger than than the upper threshold.
- A link is set to asymmetric if it is currently set to symmetric and the link-quality of the link is smaller than than the lower threshold.

\textsuperscript{1}The OSI model is a widely used reference model showing seven layers that define the different stages that data must go through to travel from one device to another over a network.
Figure 4.4 illustrates these two cases. The delay in link sensing is caused by the difference between the upper and lower threshold and the function used to calculate the quality.

The functions suggested in RFC3626 are based on a scaling factor. They are defined as:

**The stability rule**

\[ L_{\text{link}\_\text{quality}} = (1 - \text{HYST}\_\text{SCALING}) \times L_{\text{link}\_\text{quality}} + \text{HYST}\_\text{SCALING} \]

**The instability rule**

\[ L_{\text{link}\_\text{quality}} = (1 - \text{HYST}\_\text{SCALING}) \times L_{\text{link}\_\text{quality}} \]

The stability rule should be applied on a registered link every time an OLSR-package is received at that link. The instability rule should be applied to a registered link every time a packet is lost. Packet loss is detected by tracking OLSR-packet sequence numbers and by registering the `htime` field from a neighbors HELLO packets and make sure HELLO packets arrive within this interval. The following are suggested values for hysteresis calculation:

- `HYST\_THRESHOLD\_HIGH` = 0.8
- `HYST\_THRESHOLD\_LOW` = 0.3
- `HYST\_SCALING` = 0.5

One can observe that if starting with a link quality of 0 it takes three stability rule appliances to reach the upper threshold if using these values. However, going from 1.0 to the lower threshold only takes two appliances of the instability rule.

Hysteresis is a highly tune-able technique which will perform very different given other parameters.

### 4.4 TC redundancy

The two TC optimizations explained in section 3.7, optimizes the message size of TC messages and the set of nodes generating such messages. But the optimizations makes the topology understanding less robust. One may have multiple paths between nodes without knowing it.
To enable a more robust understanding of the topology, nodes can be set to announce more than just their MPR selector set in TC messages. A parameter TC_REDUNDANCY is introduced. TC messages are built based on this parameter. The following actions are taken based on the value of TC_REDUNDANCY:

- **0** - The advertised link set of the node is limited to the MPR selectors.
- **1** - The advertised link set of the node is the union of its MPR set and its MPR selector set.
- **2** - The advertised link set of the node is the full symmetric neighbor set.

If all nodes in a MANET has TC_REDUNDANCY set to 2, then all symmetric links in that network will be announced to all nodes. This means that all nodes, not just the nodes selected as MPRs, will generate TC messages.

A node has no knowledge of other nodes TC_REDUNDANCY settings, and it does not need to. Nodes without TC redundancy implemented can coexist with TC redundancy enabled nodes.

### 4.5 MPR redundancy

The core functionality section of RFC 3626 states that MPRs should be chosen so that all 2 hop nodes are covered by at least one MPR. This selection scheme will result in highly optimized flooding. But once again, bandwidth optimization can be sacrificed for robustness. One could decide that 2 hop neighbors should be covered by more than one MPR if possible. To do this, a parameter MPR_COVERAGE is introduced. This parameter specifies how many MPRs the MPR calculation should attempt to set for a 2 hop neighbor.

Core-MPR calculation states that every two hop neighbor must be covered by at least one MPR. One can not transform this rule to be *every two hop neighbor must be covered by at least n MPRs* while working with MPR redundancy. This is due to the fact that two hop neighbors might not be reachable through more than one symmetric neighbor. MPR redundancy becomes the attempt to get two hop neighbors covered by up to n MPRs.

Figures 4.5 and 4.6 illustrates the usage of the MPR_COVERAGE parameter. As seen in figure 4.6 incrementing MPR_COVERAGE leads to less optimized retransmission.
Chapter 5

Implementation background

“Knowing is not enough, we must apply; willing is not enough, we must do.”
- Bruce Lee

Simulations of ad hoc routing protocols can aid in the basic design and testing of a protocol, but certain assumptions and simplifications will be made in simulations that are not valid in a real-world scenario. Hence, it is important to implement and test routing protocols in different real-world scenarios. Work described in [45] and [26] confirms the fact that there are differences between simulations and real life usage.

One of the main goals of this master thesis project was to implement an OLSR daemon based on RFC3626 that was easy to extend and could be used for both testing as well as real-life usage. This has been done and the implementation, the UniK OLSR daemon, is referred to as olsrd in this document. The remaining chapters of this thesis is focused on this implementation and extensions made to it.

5.1 Technical

Olsrd is only implemented for the GNU/Linux platform at the current time. Some rather tricky issues prevents it from compiling on other Unix clones. These issues are discussed in detail in chapter 6.

The implementation is done in C and should compile on all modern(kernel >2.0) GNU/Linux systems. It supports loading of dynamically loadable libraries (plugins) at runtime and a GUI front-end is available. Several extensions to olsrd has been implemented as plugins, some of which are reviewed in later chapters.

5.2 History

Work on the olsrd implementation was started spring 2003. At first the plan was to add and experiment with MID functionality in the existing draft3[22] compatible OLSR implementation by INRIA[5]. This was completed by summer 2003. This means that much olsrd code originally was based on the INRIA implementation. But since then, close to all code has been rewritten or heavily modified. Olsrd is therefore considered an independent OLSR implementation and not just an extension to the INRIA implementation. If one compares the OLSR draft3 to RFC3626 one realizes the extent of the differences that exist between what can be considered an initial draft and the final protocol specification.

In October 2003 RFC3626 was released and now full RFC compliance became the goal of the project. In November 2003 UniK olsrd version 0.2.0 was made public available through a website. But full RFC core compliance was not reached until release 0.3.8 in January 2004. Not much later 0.4.0 was released. It covered all auxiliary functionality as well, except link-layer notifications.
As the implementation has matured, several extensions to it have been made. These extensions were initially implemented into the main olsrd code. This was clearly not a very modular solution, and the codebase of olsrd became bloated with special-purpose code. Therefore a plugin interface was defined and implemented, allowing the use of external plugins by the daemon. This way extensions can be made without altering the codebase of olsrd.

5.2.1 olsr.org

In January 2004 the Internet domain-name olsr.org was registered, and the official olsrd webpage was now located at http://www.olsr.org. This webpage is currently visited by about 50 individual IP addresses on a daily basis. As of this writing, it has had over 11500 daily unique visitors since it was first set up. Close to 2000 downloads of the implementation has been done, and people have contributed with binary packages for different Linux distributions like Debian, RedHat and Familiar (for hand-held devices). A package for the LinkSys WRT54G wireless router, running Linux, has also been made.

Two mailing-lists have been set up at olsr.org. Olsr-users is a list for bug-reporting and troubleshooting, while olsr-announce is set up for announcement of new version and other changes. The users list has currently close to 50 subscribers and the mail archives can be browsed at the web side.

5.3 Implementation overview

Implementing relatively large projects in C can often be quite messy, but when the project is done by one person, consistent coding-styles and design principals can be achieved.

Throughout the code some main design principals are followed:

- **Modularity** - All code that can be seen as a generic mechanism used by other entities should be implemented as modular as possible. This, in many cases, means that entities using these functions should dynamically register themselves with the function. An example of this is the scheduling functionality. Entities that wishes to generate messages on a regular interval should register a message generation function with the scheduler to be called regularly on a given interval.

  This kind of modularity becomes very important when dealing with plugins as described in chapter 8. Plugins are code that is loaded at runtime to extend a program.

- **Consistent data-structures** - In a table-driven protocol most of the computing done will be processing on tables. This means that most of the code will be related to lists of different kinds. Regardless of what kind of data structuring scheme is being used, this should be consistent. This means that as far as reasonable, all tables should have the same structure.

- **IP transparency** - The daemon should be able to operate on either IPv4 or IPv6 addresses. This means that the compiled code should be able to run in either IPv4 or IPv6 mode. So all functions and data-types treating IP addresses must be implemented to handle both 32-bit IPv4 addresses and 128-bit IPv6 addresses. This should be as transparent as possible, meaning that as much as possible (and reasonable) of the code should work on both IPv4 and IPv6 addresses. In cases where separate functions are made for IPv4 and IPv6 operation, a wrapper function should be implemented.

- **Readable code** - The code should be easy to read for an outsider. This is probably not always accomplished as a coder often tends to have an incredibly subjective view of his/her own code.

- **Platform independent code** - Platform dependent code should be separated from the rest of the code in a modular fashion, and it should be made available through a well defined interface. This way, implementing those functions for other platforms should be as easy as possible.

The OLSR daemon implementation is illustrated in figure 5.1. The different entities are explained briefly here. They will be reviewed in detail in later sections.
- **Socket parser** - The socket parser checks for incoming traffic using a `select(2)` loop. It then calls the function associated with the sockets that has incoming data. Sockets and their corresponding functions are registered at run-time.

- **Packet parser** - The packet parser receives all incoming OLSR traffic. Even though the parser has several responsibilities, it basically has three options when it comes to treating a message:
  - Discard the packet. This is done if the packet is found to be invalid.
  - Process the packet according to given instructions. To do this, the parser has to be capable of treating this message-type.
  - Forward the packet according to the default forwarding algorithm. This is done if the packet is valid but the parser has no knowledge of this message-type.

The entity to which a received message is sent, is responsible for updating the information repositories needed if the information is considered fresh enough and has not already been processed.

Parse functions for messages are registered with the packet parser at run-time.

- **Information repositories** - The tables are the heart of a table driven routing protocol. Here fresh information is kept and all calculations of routes and packets are done based on these repositories.

In these tables the information needed to at minimum describe the current state of the network, and this nodes immediate links, is kept. The various packet parsing functions both update these tables and relies on information in these tables to be able to process the messages. The forwarding functionality in particular, relies on the duplicate table which is a cache of all recent processed and/or forwarded packets.

All these tables are regularly timed out. This means that entries no longer considered valid are removed. Whenever these databases are updated in a way that changes the understanding of the network topology, the routes are recalculated.

- **Scheduler** - The event scheduler runs different events at different intervals. To transmit a message at a given interval, one can register a packet generation function with the scheduler. Timing out of tables is also triggered by the scheduler. To maintain an information repository that is timed out on a regular basis, one can register a timeout function with the scheduler.

The scheduler runs in a thread of its own and shared memory is protected using a mutex so that the packet creation entity does not have to consider synchronization and memory-protection.
5.4 About the source-code

Due to the fact that the source code of olsrd and the extensions is of somewhere between 25 and 30 thousand lines, it is not included in the appendix. If the you wish to study the code mentioned in these sections, which is highly recommended, you should download it from http://www.olsr.org. The source is bzipped and tarred into a file called uolsrd-x.x.x.tar.bz2 (where x’es are replaced with version numbers). Assuming you have the tar and bunzip2 tools available on your system you can untar and unzip this file doing:

tar jxvf uolsrd-x.x.x.tar.bz2

The olsrd source-tree consists of the following directories(relative to the unik-olsrd-x.x.x directory):

- **bin** - The olsrd binary ends up here after linking.
- **src** - All platform independent code.
- **src/linux** - All Linux-only code.
- **lib** - Various plugin code. Set up with src and bin directories as well.
- **front-end** - GUI front-end source. Set up with src and bin directories as well.
- **files** - Default configuration file and manual page.

The olsrd code is licensed under the General Public License(GPL)[11]. This license states that anyone is free to use and modify the code, but any modification that is publicly released must include full source code and be licensed under the GPL.

5.5 Portability

In a perfect world the implementation would be 100% POSIX compliant, but this of course, is not the case. Certain network based requirements cannot be met in at platform independent way, as well as issues like link-layer notifications which is highly driver-dependent. This is discussed in detail in chapter 6

A FreeBSD port has been planned for some time, but at this point there is no FreeBSD code available.

5.6 Tools used

Lots of software tools are needed for developing projects like olsrd. Fortunately all these tools are made freely available through the GNU project[35]. The tools used for development are the GNU compiler collection(gcc), the GNU emacs editor, the GNU make utility, the GNU debugger(gdb), the GNU profiler(gprof), valgrind and memproof.

5.7 Real life usage

Small, embedded systems are a natural platform for MANET routing. An implementation of a MANET routing protocol should be able to run on these kind of systems. Olsrd has been known to run on several different hardware platforms including i386, ARM, MIPS and PPC based systems. ARM and MIPS processors are widely used in embedded systems. As olsrd is implemented in pure C, it has very few dependencies. The main dependencies apart from a standard C library, is the POSIX thread library libpthread. The pthreads library is however very widespread and exists for most platforms. Olsrd also has a small footprint, and does not require much resources as we will see in chapter 10.

Even though olsrd is developed using the GNU C library(glibc) it should link to other standard libraries as well. It is known to link to uclibc[18] which is a small C library used on many embedded systems.
5.7.1 The *Wizards of Os* experiment

The *Wizards of Os*[10](WoS) is an annual conference held in Berlin. The main topics for WoS are free software, free content, free science, free networks and free hardware. Quoting from their web-page: “*The wos is showcasing freedom that works*”.

At WoS3, held June 10th to 12th 2004, an experimental mesh network was set up. This network was routed by OLSR using the UniK OLSR implementation. This might be one of the biggest real life usage tests of OLSR ever done, with a network of up to 35 nodes, some mobile, some fixed. OLSR was presented as a mean of setting up free wireless community networks at a workshop called *free networks* where a presentation of the olsrd project was given by the author.

5.7.2 Other usage

Olsrd is known to be used in many different scenarios and in a wide variety of projects by different institutions, both commercial and research-based. A few known examples are: Olsrd is used in a project where an *Internet service provider* (ISP) is setting up wireless access in apartment buildings in USA. Another company is using olsrd in a project to create *top set boxes* for hotels using wireless communication in Italy. Yet another company in the U.K. is working on integrating olsrd into a small “mesh ready” USB wireless device. And Thales Communications AS is using olsrd and most of the solutions presented in the remaining chapters in various pilot projects in Norway. Olsrd is also widely used in free network experiments where non-profit organizations work to set up wireless access in areas where there is very poor or no wired access available.

5.8 Related work

Multiple OLSR implementations exist, but as of this writing, only three of the freely available implementations claim to be RFC3626 compliant: NRL olsrd[8], QOLSR[9] and UniK olsrd.

NRL olsrd is developed at the Naval Research Laboratory of the U.S. army. It is implemented in C++ and said to be RFC3626 compliant. Tests show that it is interoperable with the UniK implementation. Nrlolsrd is available for many platforms(Unix, OS X, Windows NT), however, it does not support multiple interfaces. As seen later, the multiple interface support is actually what prevents UniK olsrd form being ported to other platforms.

QOLSR is a C++ implementation of RFC3626 aimed at Quality of Service research for MANETS. This implementation is part of an ongoing project at Laboratoire de Recherche en Informatique in France. This implementation has not been tested with UniK olsrd.

5.8.1 Other protocols

AODV-UU[1] is an implementation of AODV done at the Department of Information Technology at Uppsala University in Sweden. The implementation is interesting as it runs both on real life GNU/Linux systems and in the NS2 network simulator and it is consists of user-space code only(as opposed to other AODV implementations).

Kernel AODV[6] is another interesting AODV implementation. It is created by NIST which is an agency of the US Department of Commerce’s Technology Administration. Kernel AODV is implemented as a Linux kernel module. This way all AODV operations are done in kernel space.
Chapter 6

Implementing core functionality

“I have stopped reading Stephen King novels. Now I just read C code instead.”

– Richard O’Keefe

In this chapter we will take an in-deep look at the different approaches taken when implementing the core functionality of OLSR. The UniK olsr daemon is referred to as olsrd, while the protocol specification, RFC3626, is referred to using the all uppercase acronym OLSR.

In all technical aspects of this chapter it is assumed that one is working on a GNU/Linux based system with UDP/TCP IP support.

6.1 General implementation issues

6.1.1 Output

Olsrd can be set to direct various degrees of output describing current operation and status to the standard output (stdout). There are 9 debug levels that can be used where 1 yields a minimal output while 9 will output lots of information. Olsrd also logs start, stop and errors to the system log using the glibc syslog(3) interface. If olsrd is set to run at debug-level 0, it will detach itself from the parent process and run in the background as most daemons do, syslogging will still be done. The debug level is set either at the command line using the -d switch, or in the configuration file.

The interface for stdout output from olsrd is the function:

```c
int olsr_printf(int, char *, ...);
```

This function is basically just a wrapper for the printf(2) standard library call. But the function takes a debug-level integer as the first parameter. This value indicates what minimum debug-level is required for the output to be displayed.

6.1.2 Configuration file

At startup, olsrd tries to read various configuration parameters from a configuration file. Parameters given at the command line overrides the settings from the configuration file. The location of this file is by default set to /etc/olsrd.conf but a different file can be used using the -f command line switch.

The configuration file is a set of directives mapping keywords to values. Values such as interfaces to use, IP version to use, plugins to load, various emission intervals and so on, can be set in this file. An example of a
configuration file can be found in appendix A while appendix B contains the Linux manual page for olsrd. In the manual page all command line options are documented.

6.1.3 Packets and OLSR definitions

All data-types and constants that are directly related to the OLSR protocol, and some globally used definitions and data-types, are defined in the header-file `olsr_protocol.h`. Here all packet-formats can be found, these are the actual data-structures used when transmitting data. Olsrd sometimes uses other internal structures to represent packets while processing or generating them. This header file is included in appendix C and the reader is encouraged to study this file.

6.1.4 Communicating with the kernel

One of the first things olsrd does after setting up various variables based on either a configuration file, command line arguments or just default values (often a mix of these), is to validate and configure communication interface information. This requires fetching information from the kernel. The task of communicating with kernel mechanisms can seem daunting to someone that have never worked on such a low level on a Linux system before. Luckily the GNU/Linux system is made up of open-source components, therefore one is free to study all source code. Studying kernel and library header-files and reading source-code of applications that include the operations one wishes to implement, is a smart thing to do when facing such new ground. A look at the source of the standard Unix network interface configuration tool `ifconfig`, is highly recommended when getting into network interface configuration.

To be able to communicate with drivers running in kernel-space, two approaches are used. The first is the `ioctl(2)` system call. The second is the `proc` pseudo file-system.

`ioctl(2)`

The definition of the `ioctl` function is:

```c
int ioctl(int d, int request, ...);
```

The Linux manual page of `ioctl` states:

The `ioctl` function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with `ioctl` requests. The argument `d` must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It’s traditionally char *argp (from the days before void * was valid C), and will be so named for this discussion.

An `ioctl` request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an `ioctl` request are located in the file `<sys/ioctl.h>`.

Usually, on success zero is returned. A few `ioctl` s use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and `errno` is set appropriately.

In other words, `ioctl(2)` is a multi-purpose system call. Based on the request parameter, lots of information can be gathered from or sent to kernel space. Most hardware devices can be manipulated and information can be fetched from them by this system-call. Data to be received or set, is put in some predefined data-structure depending on what request is issued.
The proc pseudo file-system

The /proc directory on GNU/Linux systems provides a file-system like interface to the kernel. This allows applications and users to fetch information from and set values in the kernel using normal file-system I/O operation.

The proc file system is sometimes referred to as a process information pseudo-file system. It does not contain “real” files but rather runtime system information (e.g. system memory, devices mounted, hardware configuration, etc). For this reason it can be regarded as a control and information center for the kernel. In fact, quite a lot of system utilities are simply calls to files in this directory. For example, the command \texttt{lsmod}, which lists the modules loaded by the kernel, is basically the same as \texttt{`cat /proc/modules’} while \texttt{lspci}, which lists devices connected to the PCI bus of the system, is the same as \texttt{`cat /proc/pci’}. By altering files located in this directory you can change kernel parameters while the system is running.

The \texttt{proc(5)} manual page describes most entries in detail.

6.1.5 Configuring network interfaces

Olsrd needs to fetch information about the network interfaces it is set up to use. This mainly includes fetching the unicast and broadcast IP addresses used by the network interface. To fetch such information various ioctl calls are made. They are documented in the \texttt{netdevice(7)} manual page. If using IPv6, some considerations have to be taken, this is discussed later. Most interface configuration in olsrd is done in the \texttt{src/interface.c} file. The ioctl commands are defined in various header-files, the most interesting commands regarding network interfaces, can be located in \texttt{sys/ioctl.h}.

Network interfaces are only set up for use with olsrd if certain criteria is met. The interface has to have the flagset, it has to be configured with an IP address and broadcast address and it cannot be a special device such as the loopback interface.

Detection of interface type

Olsrd will try to register what interfaces are WLAN interfaces to be able to use different HELLO intervals on wireless and wired interfaces. Since mobility is assumed to be low on wired links, one can reduce control-traffic overhead by using a larger HELLO interval on wired interfaces. These intervals can be set in the configuration file.

To detect if an interface is to be registered as wireless, a request for WLAN information ioctl, \texttt{SIOCGRNAME}, is issued to the driver. If it returns information other than \texttt{NULL}, the interface is set to be wireless. This, however, does not work as desired when using hardware such as Ethernet-to-WLAN bridges.

When olsrd selects the interface to use in a route, interfaces will be prioritized based on a metric set in the configuration process. As of now this metric is set to 1 if the interface is detected to be a WLAN interface, and 0 if not. This means that if a node has two symmetric links to a neighbor node, one over a WLAN interface and one over an Ethernet interface, then the Ethernet interface will be chosen when setting up the route.

6.1.6 Updating the kernel routing table

Another task that requires communicating with the kernel is adding and removing IP routes. Again the ioctl system-call is what we need, and again a look at the source code of standard Unix tools can be of benefit.

\footnote{All standard header-files are relative to /usr/include}
In this case, taking a look at the source of the `route` command can be good advice. The source code for the actual kernel route updates can be found in `src/linux/kernel_routes.c`.

The header-file `bits/sockios.h` declares the ioctl commands needed to manipulate kernel routes, while the `net/route.h` header-file conditions the data-structures needed to pass the route information to the kernel.

6.2 Control traffic

6.2.1 Broadcasting OLSR packets

When using IPv4, OLSR control traffic is to be broadcasted on UDP port 698. However, the RFC does not specify what broadcast address is to be used. The natural choice is to use the broadcast address an interface is preconfigured with when transmitting on that interface.

Broadcast addresses are usually made up of the interface address bitwise OR'ed with the inverse of the netmask. If an interface is configured with the address 192.168.44.23 with netmask 255.255.255.0 then the broadcast address would be 192.168.44.23 OR 0.0.0.255 = 192.168.44.255.

The UDP/IP implementation in GNU/Linux will only pass received broadcasted messages up the stack if the message is destined either for the broadcast address that the receiving interface is configured with, or in the special case of using the broadcast address 255.255.255.255. If using the broadcast address fetched from the interface, OLSR traffic will only be received by other nodes using communication interfaces configured with the same broadcast-address, usually meaning that the interface is configured with an IP address in the same subnet as the sender. However, in the 255.255.255.255 case, all nodes receiving the data will always pass the message up the networking stack, if, as described in the next section, address-spoofing filtering is disabled on the receiving device. This means that all IPv4 addresses can be used by nodes in such a MANET without any consideration for subnetting.

Since ad-hoc networks often should be “open” solutions with no special infrastructure, using 255.255.255.255 as destination for control-traffic can be seen as a good solution in many scenarios. Actually, the AODV RFC explicit says that all control traffic is to be broadcasted 255.255.255.255, so this is not an unknown approach.

In olsrd, one can specify the broadcast address to use in the configuration file. By default, the broadcast address used is the one an interface is set up with.

6.2.2 IP address filtering

IP spoofing filtering is much used for ingress/egress filtering[37]. This is to prevent IP packets originating from addresses to which the receiving interface has no route, from being processed. This way it is harder for nodes to spoof an originator address.

This filtering is enabled by default in the Linux kernel by the initialization scripts in most standard Linux distributions, and it must be disabled in olsrd for several reasons. As mentioned in the previous section, one must disable this filtering to be able to use the 255.255.255.255 control-traffic broadcast approach. Doing this means that all packets broadcasted to 255.255.255.255 will be passed up to the application layer of all nodes listening on that UDP port no matter the sender address. But the IP spoofing filter mechanism has turned out to cause more problems than just the 255.255.255.255 broadcast issue. When two nodes have multiple symmetric links between them, the filter will stop incoming broadcasts on all other interfaces but the one on which a host route to the neighbor node is added. When using IPv4, olsrd automatically disables address spoof filtering for all interfaces on which it runs.

Address spoof filtering can be disabled per interface in Linux by writing “0” to the proper proc entry. For the `eth0` interface that will be `/proc/sys/net/ipv4/conf/eth0/rp_filter`. 

37
### 6.2.3 Directing outgoing traffic

As explained in section 3.6.1 one has to be able to transmit individual HELLO packets on every network interface due to link-sensing. A problematic scenario would be if a node was to run olsrd on two interfaces that are configured with IP addresses in the same subnet range. Transmitting to a broadcast-address in this subnet using one UDP socket\(^2\) would result in traffic only being sent on the interface that has the first entry for this subnet in the routing-table. Say a node wishes to run OLSR on the interfaces `eth0` and `eth1`. The interfaces are configured as follows:

```
eth0:  IP:  192.168.1.1
      NETMASK:  255.255.255.0
      BROADCAST:  192.168.1.255
eth1:  IP:  192.168.1.2
      NETMASK:  255.255.255.0
      BROADCAST:  192.168.1.255
```

The routing table contains the following entries:

<table>
<thead>
<tr>
<th>DESTINATION</th>
<th>GATEWAY</th>
<th>NETMASK</th>
<th>INTERFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.0.0</td>
<td>0.0.0.0</td>
<td>255.255.255.0</td>
<td>eth0</td>
</tr>
<tr>
<td>192.168.0.0</td>
<td>0.0.0.0</td>
<td>255.255.255.0</td>
<td>eth1</td>
</tr>
</tbody>
</table>

If the host sends a UDP packet to `192.168.1.255`, it will now be transmitted through `eth0` only.

Another problem would be a situation where one uses the `255.255.255.255` broadcast address or as we will later see, when IPv6 multicast is to be used. In these cases, all messages will always be transmitted on all interfaces. Clearly the ability to direct traffic explicitly to interfaces is needed. To be able to gain such control of traffic, the sockets used for transmission has to be mapped to physical interfaces. One socket is created for every interface that participates in the OLSR routing domain. This socket is bound to the interface using the `S0_BINDToDevice setsockopt(2)` call. In Linux one can bind several sockets to the same UDP port if they are bound to different interfaces using this setsockopt call. Unfortunately, this is a Linux specific way to do this, and this issue is one of the biggest problems if one is to port olsrd to other platforms. There is no standard POSIX way to bind sockets to devices.

### 6.3 IPv6

Olsrd supports both IP version 4 and 6. The main goal when IPv6 support was added, was to make the code transparent to what IP version it was running. The IP version to use is set at runtime, and the olsrd binary supports both addressing schemes, no recompiling is required.

#### 6.3.1 Olsrd IP data-type

The most obvious difference when working on IPv6 contra IPv4, from the developers point of view, is the address size. While IPv4 uses 32-bit addresses IPv6 addresses are made up of 128 bits. To make this difference transparent the following data-type is defined in `olsr_protocol.h`:

```
union olsr_ip_addr
{
    olsr_u32_t v4;
    struct in6_addr v6;
};
```

\(^2\) A socket is the Unix abstraction of a network communications endpoint. The analogy is to a wire (the network data connection) being plugged into a socket.
This union is used whenever dealing with IP addresses in olsrd. To be able to work easily with `olsr_ip_address`
data-types two macros are defined in `olsr_protocol.h`:

```c
COPY_IP(to, from)
```
Copies an IP address as pointed to by `from` to the memory area pointed to by `to`.

```c
COMP_IP(ip1, ip2)
```
Compares the address pointed to by `ip1` with the address pointed to by `ip2`.

This is very convenient when working on this transparent IP address data-type.

### Converting IP addresses to strings

When debugging, it is highly desirable to be able to print IP addresses in the common IPv4 and IPv6
“human readable” formats. Olsrd provides some functions for conversion of the `olsr_ip_addr` data-type
to a NULL terminated char array. The most used of these functions is defined in `olsr.h` as:

```c
char *
olsr_ip_to_string(union olsr_ip_addr *);
```

Since this function uses the `olsr_ip_addr` union, it is IP version transparent. This function uses a static
allocated buffer, so it is not reentrant. That means that it should not be used twice in the same function call.

### 6.3.2 Configuring interfaces using IPv6

All interfaces used in normal IP routing must be set up with a unicast address to receive unicasted traffic.
In IPv6 there are several classes of unicast addresses. The different classes are:

- **Link-Local addresses** - These are used during auto-configuration and when no routers are present.
  Link Local addresses are valid only on the link of an interface. This means that traffic destined for
  such addresses is never routed.

- **Site Local addresses** - These are similar to the private IPv4 addresses, with the added advantage
  that everyone who uses this capability is given the ability to use the given 16 bits for a maximum of
  65536 subnets. Also, because you can assign more than one ip to an interface, you can communicate
  on your private LAN and the Internet simultaneously without any special routing or NAT translations.

- **Global unicast addresses** - These are public (Internet) IP addresses. All subfields in global addresses
  are variable-length, non-self-encoding.

Since interfaces often are configured with multiple IPv6 addresses from both the site local and global
address space, there is the question of what address olsrd should use for the interface. The address-type-to
use can be set in the configuration file using the `IP6ADDRTYPE` statement. By default it is set to use site
local addresses since a MANET often is seen as a private network. However, if Internet connectivity is
desired one needs to use global addresses or one must use some address translation scheme in the Internet
gateways. Optionally, a node could send all its site-local and global addresses in MID messages to ensure
that all of its addresses are route-able.

In olsrd, configuration of interfaces is done different when using IPv6 from IPv4. While IPv4 configuration
uses ioctl calls, IPv6 operation reads all info from the proc file `/proc/net/if_inet6`. This file lists all configured
IPv6 addresses and the interfaces they are configured to. This way a network interface can have multiple
entries in the file, one for each configured IPv6 address. Each entry includes information about the address
type as well. When fetching interface information, this proc file is parsed and the first located address of
the correct type for the specific network interface is used. If no address of the desired class is located for a
network interface, then that network interface will not be set up for use by olsrd.
6.3.3 IPv6 multicast

Since IPv6 has no broadcast mechanism, a multicast address is used by the sockets transmitting OLSR packets. Using multicast in its most basic form in IPv6 is a matter of setting a socket to join a multicast sender group and to join a multicast receive group. This is done using the setsockopt(2) call and the IPV6_ADD_MEMBERSHIP and IPV6_JOIN_GROUP flags.

The default multicast-address used is ff05::15 for interfaces using a site-local address and ff0e::1 for interfaces using a global address, if nothing else is specified by the user. The multicast addresses to use can be set in the configuration file.

6.3.4 IPv6 sockets

The IPv6 socket API for GNU/Linux aims to be mostly compatible with the IPv4 API. In fact, if coding in an “IPv6 aware” style, most programs should be IP version independent. The main socket related system calls, such as socket(2), bind(2), listen(2) and accept(2), are all IP version independent. However, the sockaddr_in6 structure used to represent and IPv6 address is bigger than the generic sockaddr used for IPv4 addresses. One should therefore use the struct sockaddr_storage for IP address storage if writing IP version independent code.

6.3.5 Routing table update

Updating kernel IP routes is done quite similar for IPv6 as for IPv4. The ioctl’s used are the same, but IPv6 sockets are used for the calls and different parameter data-types are used. The data-structures used are defined in include/linux/ipv6_route.h. IPv6 equivalents of all route adding and removal functions are implemented.

6.4 The socket parser

As seen in figure 5.1, incoming OLSR data is first handled by the socket parser. This entity is responsible for listening for data on a given set of sockets. Sockets and their corresponding parse functions, are registered with the socket parser at runtime. The socket parser uses the familiar select(2) system call to detect when data is available on any socket in a given set of sockets. The socket parser functionality is illustrated in figure 6.1. This modular design allows for multiple entities to listen for multiple types of data without the need for a program flow, often a thread, of their own. So whenever an entity wishes to listen for data on a socket, it calls the function prototyped in socket_parser.h as:

```c
void
add_olsr_socket(int, void(*)(int))
```

Here, the first argument is the socket(file-descriptor) to check for data, and the second argument is the function to call when data is available on the socket. A function is available to remove a registered socket as well:

```c
int
remove_olsr_socket(int, void(*)(int))
```

Sockets can be registered and removed at any time. This makes the socket parser suitable to handle both server- and client-side type connections. If some entity wishes to act as a server for a TCP socket, it would first register its connection listening socket with the socket-parser with a function that handles new connections. Whenever a new connection is initiated, the function will register the new socket with the socket parser. If the connection is lost, the entity removes the socket from the socket parser set.
6.5 The packet parser

Olsrd registers all control-traffic sockets with the socket parser at startup. These sockets are registered with a packet parser function to be called whenever data is available. This way the packet parser receives all broadcasted traffic received on UPD port 698 on either interface. As explained earlier, one socket is maintained per communication interface running olsrd. So upon startup the situation will be as illustrated in figure 6.2 given that only OLSR sockets are registered with the socket-parser.

The packet parser receives OLSR packets on the form illustrated in figure 3.1. It checks if the reported size in the OLSR header matches the received amount of data. If so, it parses the packet into messages. These messages are passed on to registered message-parsing functions. If the received size does not match the size read from the OLSR header, the packet is silently discarded.

This design is similar to the socket parser. Message-parser functions can be registered and removed for all message types dynamically. One can also register functions with a packet type of PROHISP (defined in parser.h), such functions will receive all incoming messages. A function, defined in parser.h, is used
to add a message-parser function:

```c
void
olsr_parser_add_function(void (*)(union olsr_message *,
                          struct interface *,
                          union olsr_ip_addr *),
                       int, int);
```

The first argument is the function to call upon receiving a message of the given type. This function is passed the message, information of on what interface the message arrived and the IP address of the neighbor that transmitted it. The next argument is the packet type, and the last is a boolean value stating whether or not the registered function is forwarding the message. If this value is set to 0, the default forwarding algorithm is applied to the messages of the registered type by the packet parser.

The packet parser delivers all OLSR messages contained in a packet to the functions registered to parse this message-type. All messages are also delivered to message parser entries registered with the PROMISCUOUS type. If none of these functions are registered to forward the message, then the packet parser forwards the message according to the default forwarding algorithm.

### 6.6 Information repositories

Table driven routing protocols maintain a soft-state. This means that information is dynamically updated and removed based on changes in the network topology and timeouts. All information is kept in tables, and these must be designed in some clever way, as they are traversed and searched almost continuously.

#### 6.6.1 Timers

Entries in the tables in olsrd are kept fresh by setting and maintaining a timestamp on creation and updates. This timestamp is the sum of current time and the validity time this information is registered with. If the current timestamp is higher than the registered timestamp, the entry is said to be timed out. To check for timed out entries, tables are traversed and all timestamps are compared to current time. This operation is done regularly on a small time interval to be sure no entries are kept for much longer than their validity time. Most timeout functions are called at every scheduler poll.
Olsrd provides functions for retrieving timestamps and comparing them with current time. These functions are declared in `olsr.h`.

### 6.6.2 List structures

As mentioned, lots of searches and traversals are done on the tables in olsrd. By default the scheduler calls functions that traverse all tables to remove timed out entries, at every poll (defaults to every 0.1 seconds). In addition close to all message parsing and generation is based upon searching these tables. It is pretty obvious that one will benefit from using data-structures that minimizes the search time and eases the process of adding and deleting entries. We will take a look at the most commonly used data structures for such operations.

**Linked lists**

Linked lists are the most trivial approach to structuring dynamically sized data sets. A linked list is just a list of entries where each entry keeps a reference to the next entry in line. The last entry could either reference the first entry in the list, making the list more of a circle, or some predefined value (typically `NULL` in a C environment).

This approach can be optimized for deletion by using double linked-lists. This means that all entries maintain a pointer to their previous and next entries. This way one needs no knowledge of other list elements than the actual element to remove when deleting elements. On top of this, a linked list can be sorted to make search time smaller. One can sort the list by some value in the entries data. This can optimize search time by not having to traverse the entire list to verify that an element does not exist. One can also search backwards or forwards based on the value of the element one searches for.

![A linked list. It can be one-way linked only following the full lines or double-linked following the dashed lines as well. The last element may point to the first or some predefined value like NULL.](image)

**Binary trees**

The binary tree is a search optimized version of linked lists. The linked list entries are expanded to include left and right children, indicating values less than or greater than the parent element. Filling in the structure with elements yields a tree like structure, with one root node and an unlimited number of branches of data, as illustrated in figure 6.5.

The functions for inserting and searching such a data-structure are rather trivial, and the payoff is that a search never extends the depth of the tree. But removal of nodes is a rather complex issue since the possible subtrees spanning from the node to be removed has to be rearranged. Even for a relatively small tree like the one illustrated in figure 6.5 the removal of the root node(20) will trigger a rather expensive rearrange operation. And if one is to keep the tree balanced, meaning that no leaf nodes are more than 1 level apart, insertion becomes more of a complex issue as well.
B-trees

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key itself but greater than the preceding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node. Note that the actual data is stored only in leaf nodes, the tree structure is only used for searches. Figure 6.6 illustrates a typical b-tree.

B-trees are index data structures designed for fast lookup, insert and delete mainly designed for really huge indexes that are too big to fit in main memory and therefore reside on mass storage like harddisks. This approach is complex and would be an overkill for the requirements of olsrd.

Hashing

Hashing is the transformation of a certain source value, often a string of characters, into a usually shorter fixed-length value or key, that represents the original data. In a data-structuring context, hashing is used to index and retrieve items in databases and hash-lists, where entries are indexed by the hash. Many programming environments offers such hashed lists as a part of the basic data/object types. Hashes makes lookup faster in the case where the hashing function requires less resources than a lookup in a structure like a binary-tree. The easiest approach to hashing, using array standards from the C language, is to create a hash function that produces an integer between 1 and $n - 1$. Then one can allocate an array of $n$ elements of a certain data-type. New elements are then inserted into the `hash(data)` index of the array.

A hashing function is used to generate hashes, often referred to as keys. These hashes should be “relatively unique”, meaning there is a rather low probability of generating the same hash from two different sources. But most important, the function must always produce the same hash for the same input value. The possibility of equal hashes generated based on different input values is highly dependent on the design of the
hash function. Functions working on hashed lists therefore have to be able to handle index collisions.

**The approach taken in olsrd**

OLSR is not designed to route huge networks. So the tables in OLSR will hold a relatively small amount of entries. A MANET consisting of more than 500 nodes is not very likely to be routed as one OLSR domain, so as the tables in olsrd usually registers at maximum one entry per node, some sort of hashing scheme based on the identifier of nodes will pay off here.

Olsrd uses a hash based on the main IP address of a node to index the node in a statically allocated array. Every element in this array is the root element in a double-linked list. The structure is illustrated in fig. 6.7. The elements in the lists are not ordered and the size if the hash-array is set by the constant HASHSIZE defined in *src/hashing.h*. This size must be as big as the largest possible hash derived from an IP address using the hash function *olsr_hashing* also defined in *src/hashing.h*.

The hashing function is currently very trivial. In IP addresses, the lower bits are expected to be the most unique part. This is definitely the case if, for example, most IP addresses in a dataset are part of the same subnet. Therefore, when creating hashes from IP addresses, it is natural to utilize the lower bits of the address. The hash generated for an IP address in olsrd is simply the lower 5 bits of the address. This gives a hash list-size of 32. This value can easily be updated to be more suitable for larger amounts of data.

Using statically allocated root elements makes traversing and inserting elements easier, as one never has to check for empty lists. No data is ever stored in the root elements, they are only used as references to the start, and the end, of the list. To lookup an element, given an IP address, the following steps must me taken:

1. Find the hash of the IP address:
   \[
   \text{hash} = \text{olsr\_hashing}(\text{IP})
   \]
2. Traverse the list from the root element at index = hash in the table:
3. Check for a match:
   
   ```
   if (element->IP == IP)
   ```

To insert an element the following steps are required:

1. Set the new elements next pointer to be the root elements next element:
   ```
   new_element->next = root.next
   ```

2. Set the root elements next elements previous pointer to be the new element:
   ```
   root.next->prev = new_element
   ```

3. Set the root elements next pointer to be the new element:
   ```
   root.next = new_element
   ```

4. Set the new elements previous pointer to be the root element:
   ```
   new_element->prev = &root
   ```

These four steps are available through the macro `QUEUE_ELEMENT` defined in `src/olsr.h`.

Removing an element is very easy and requires no knowledge about anything but the element to be removed:

1. Set the previous elements next pointer to be this elements next pointer:
   ```
   entry_to_delete->prev->next = entry_to_delete->next
   ```

2. Set the next elements previous pointer to be this elements previous pointer:
   ```
   entry_to_delete->next->prev = entry_to_delete->prev
   ```

3. The entry is now dequeued and can be deleted:
   ```
   delete(entry_to_delete)
   ```

The removal operation is available through the `DEQUEUE_ELEMENT` macro defined in `src/olsr.h`.

As seen in these examples, the use of statically allocated root elements prevents the situation where a list goes empty. Due to this, one never has to check for empty pointers or pointers to oneself when removing elements.

### 6.6.3 Detecting changes

Whenever changes are being made to entries that causes updates of the topology or neighborhood understanding, the function responsible for these changes must also signal that such changes have been made. This is done by setting one of the the global variables `changes_topology` or `changes_neighborhood` to `UP`. These variables are declared in the `olsr.h` header-file. Setting `changes_topology` causes all routes, including MPR calculation and HNA routes as described later, to be recalculated. Setting `changes_neighborhood` causes MPR selection to be recalculated in addition to all routes and HNA routes. Recalculation of only the HNA routes as explained in section 7.1.3, can be triggered by the global variable `changes_hna`.

These variables are processed at every scheduler poll in the function `olsr_process_changes` defined in `olsr.h`. If routes needs to be recalculated immediately, a function can call the `olsr_process_changes` function directly. This is necessary in some situations, like when changes have occurred in the neighborhood. If these changes lead to changes in registered MPR selectors, the MPR selector set should be updated.
prior to processing the next message. And since the message processing functionality holds the mutex used by the scheduler, no scheduler poll will occur, recalculating the MPR set, until all messages in the current packet has been processed. Therefore the function calls `olsr_process_changes` directly, to ensure that possibly remaining messages are forwarded using updated MPR selectors.

### 6.7 The scheduler

The scheduler in olsrd runs registered tasks at given intervals. By default the olsrd scheduler polls every 0.1 second. This interval might not be fine-grained enough if running olsr with small emission/hold intervals, therefore it can be set in the configuration file. When using the default emission intervals it should be more than sufficient.

As with the socket- and packet-parser, the scheduler is designed in a modular fashion. One can register and remove scheduled functions at runtime. Entities that wishes to have some function, typically a packet generation function, ran at a constant interval can register this function with the scheduler using this function, defined in `scheduler.h`:

```c
int olsr_register_scheduler_event(void (*)(), float, float, olsr_u8_t *);
```

The first parameter is a pointer to the function to be ran, the second is the interval(in seconds) on which the function is to be ran, the third is the initial time to register as passed since the last call of this function and the fourth is a trigger by which the function can be forced to be called by the scheduler at the next poll. To register a function that is called on every scheduler poll the following function is used:

```c
int olsr_register_timeout_function(void (*)());
```

This function is named `olsr_register_timeout_function` due to the fact that the functions registered to be ran at every scheduler poll is most likely to be functions that times out information in repositories.

At a scheduler poll, all timeouts of all registered functions are checked and updated. The functions are executed if required. For functions registered with a trigger, this trigger is checked. If set to 1, the function will be executed regardless of the timer recorded for it. The timer for that function will then be reset. All timeout-functions are then ran, and the `olsr_process_changes` function is executed to update any changes. After running all necessary functions the scheduler sleeps for \((\text{poll interval} - \text{processing time})\) seconds where \text{processing time} is the time elapsed for the above processing.
6.7.1 Data integrity - mutexes

The scheduler runs in a thread of its own using the `pthreads` library. This means that it has access to all functions and data in olsrd, while still maintaining its own program flow. This again leads to the problem of multiple resources accessing the same data “simultaneously”, known as a race condition. This could occur in olsrd in a situation where a packet generation process initiated by the scheduler, builds a packet based on the link-set. At the same time the link-sensing functionality called by the packet-parser running in a separate thread, is writing to the link-set. This type of situations must be avoided, and to do this the common practice is to utilize some sort of locking of code regions. This means that a thread is only allowed to access certain parts of a program, called a critical section, if it can take hold of a certain resource. This resource is the lock. While one thread holds the lock no other threads can take it. This is also referred to as mutual exclusion. Therefore the lock resource is often referred to as a mutex.

Olsrd utilizes the pthread mutex API to make sure the functions called by the scheduler and the packet-parser never runs simultaneously. The socket parser has to acquire the lock to be able to call the appropriate packet-parser function when receiving traffic. The scheduler has to acquire the lock to be able to execute any of the registered functions. The design is illustrated in figure 6.9.

6.8 Maintaining state

Now that the main building blocks of olsrd are covered, it is time to see how the actual OLSR functionality is implemented using these blocks.

6.8.1 Sequence numbers

Sequence numbers are used to determine the freshness of messages. At startup, the initial sequence numbers to use in message generation are initialized with random values.

The packet format used in OLSR limits the sequence numbers to a 16 bit value. This leads to the occurrence of wrap-around\(^3\) of sequence numbers. To prevent any problems concerning wrap-around, the RFC proposes the use of the following statement to decide if the 16 bit value \(S_1\) represents more recent information than \(S_2\). MAXVALUE is the largest possible value of the sequence number.

\[
((S_1 < S_2) \land (S_1 - S_2 \leq \text{MAXVALUE}/2)) \lor ((S_2 > S_1) \land (S_2 - S_1 > \text{MAXVALUE}/2))
\]

\(^3\)When a number stored in a variable is increased from the max value allowed by the size of the data-type it will wrap around and start over at 0
Thus, when comparing two messages, it is possible, even in the presence of wrap-around, to determine which message contains the most recent information. This check is available through the macro `SEQNO_GREATER_THAN(s1, s2)` defined in `src/olsr_protocol.h`.

### 6.8.2 Duplicate set

As the same message can be received several times, due to different nodes forwarding it, one needs some system to avoid duplicate processing of data. This is what the duplicate set is used for as described in section 3.4.2. The duplicate set can be queried through this function, implemented in `src/duplicate_set.c`:

```c
int olsr_check_dup_table_proc(union olsr_ip_addr *, 
                           olsr_u16_t);
```

This function also updates the duplicate set if no match is found, so that the message queried for is registered as processed. Note that no standard olsrd functionality calls this function directly. The `olsr_forward_message` function, presented in section 6.10, takes care of both querying the duplicate table and queuing the message for forwarding if necessary.

### 6.8.3 Link sensing

All link sensing code reside in the `link_set.c` file. Link sensing combined with HELLO message generation is some of the more complex code in olsrd, but the algorithms are pretty well described in RFC3626. The link sensing functionality is part of the HELLO parsing functionality and controls neighbor detection. The link set is maintained by the HELLO message parsing function `olsr_process_received_hello`. The entire interface to updating the link set is the function declared in `link_set.h` as:

```c
struct link_entry *
update_link_entry(union olsr_ip_addr *,
                 union olsr_ip_addr *,
                 struct hello_message *,
                 struct interface *);
```

This function updates the link status of the link on which a HELLO message was received, and if necessary creates the entry. The link set is kept fresh by a timeout-function that is registered with the scheduler to be ran at every poll. This function deletes timed out entries. To query the link set for the status of a link, the following function is available:

```c
struct link_entry *
lookup_link_entry(union olsr_ip_addr *, union olsr_ip_addr *);
```

It takes the local and the remote IP addresses as arguments to define the link in question. When working with the link set, it is worth noticing that several links can exist between the same nodes.

### 6.8.4 One- and two-hop neighbor sensing

The neighbor set is maintained by the link set. The link set functionality creates, updates and deletes neighbor entries. This code resides in the `src/neighbor_table.c` file. The neighbor set has a strong connection to the two-hop neighbor set, and the data structures from the two sets maintain pointers to each other as illustrated in figure 6.10. These connections are kept to assist in the MPR calculation.
Two hop neighbors are registered in the two-hop neighbor set based on HELLO messages. It is important to note that a symmetric one-hop neighbor might be registered as a two hop neighbor as well. In a situation like depicted in figure 6.11, A would declare B and C as its neighbors in its HELLO messages. B would then add C to its two hop neighbor set even though it has registered C as a symmetric one-hop neighbor. This redundancy makes a possible transition from an one-hop neighbor to a two-hop neighbor smoother when a symmetric link is lost.

6.8.5 MPR registration and calculation

The MPR selector set is populated based on information received in HELLO messages. A node declares its MPR set in these messages. This way a node will know what neighbors have chosen it as a MPR. The MPR selector set is queried when forwarding messages using the default forwarding algorithm, and timed out by a timeout function registered with the scheduler.

As opposed to the MPR selector set, the MPR set contains all selected MPRs. Actually, the MPR set is not a data repository arranged in the way most other datasets in olsrd are. If a node is selected as a MPR, the member ia_mpr in the neighbor entry representing the node, is set to 1. This way the MPR selection does not need to maintain a database of its own. Recalculation of MPRs is triggered by changes in the one- or two-hop neighborhood.
Calculating MPRs

The MPR computation process is described in section 8.3.1 of RFC3626. It is one of the more complex algorithms in olsrd. The implementation of the algorithm is located in src/mpr.c. The structure illustrated in figure 6.10 is used in this process. All one-hop neighbors are linked to the two-hop neighbors they can reach. This way when selecting a neighbor as MPR, all corresponding two-hop neighbors can easily be updated to reflect this.

The algorithm removes all previously selected MPRs and recalculates the whole MPR set. This is done by traversing the neighbor set based on the registered willingness of neighbors, starting with a willingness of 7 and decreasing down to 1. Nodes with a willingness of 0(WILL_NEVER) are never selected as MPRs while nodes announcing a willingness of 7(WILL_ALWAYS) will always be selected as MPRs.

Setting willingness

In olsrd willingness is set based on the power-status of the node. This information is extracted from the pseudofile /proc/apm which is the user-space interface to the the Advanced Power Management offered by the kernel. If no such file is present, willingness will be set to WILL_DEFAULT(3). The user can also set a fixed value for willingness in the configuration file.

The willingness is based on a trivial calculation. Beneath is the snippet of code that calculates willingness. This code is implemented in the function olsr_calculate_willingness in src/olsr.c:

```c
/* If AC powered */
if (ainfo.ac_line_status)
    return 6;
/* If battery powered */
*    * juice > 78% will: 3
*    * 78% > juice > 26% will: 2
*    * 26% > juice will: 1
*/
return (ainfo.battery_percentage / 26);
```

The algorithm removes all previously selected MPRs and recalculates the whole MPR set. This is done by traversing the neighbor set based on the registered willingness of neighbors, starting with a willingness of 7 and decreasing down to 1. Nodes with a willingness of 0(WILL_NEVER) are never selected as MPRs while nodes announcing a willingness of 7(WILL_ALWAYS) will always be selected as MPRs.

Setting willingness

In olsrd willingness is set based on the power-status of the node. This information is extracted from the pseudofile /proc/apm which is the user-space interface to the the Advanced Power Management offered by the kernel. If no such file is present, willingness will be set to WILL_DEFAULT(3). The user can also set a fixed value for willingness in the configuration file.

The willingness is based on a trivial calculation. Beneath is the snippet of code that calculates willingness. This code is implemented in the function olsr_calculate_willingness in src/olsr.c:

```c
/* If AC powered */
if (ainfo.ac_line_status)
    return 6;
/* If battery powered */
    *    * juice > 78% will: 3
    *    * 78% > juice > 26% will: 2
    *    * 26% > juice will: 1
*/
return (ainfo.battery_percentage / 26);
```

The calculation is based on two factors. First, if the node is connected to an AC power-source it is regarded to have unlimited access to power, therefore it will announce a willingness of 6 which is the maximal “normal” willingness level. If the node is not AC powered, it is considered battery powered. In that case the willingness is set based on the percentage of power left on in battery source as shown in the code above.

6.8.6 Topology set

The topology set consists of tuples describing symmetric links between nodes. Links are registered on the form (last_addr, dest_addr) which describes a link from the node with address last_addr to the node with address dest_addr. These entries are maintained and created based on received TC messages and the data-set is timed out in the same manner as all other tables. The topology set is queried when calculating routes, this is the actual link-set in OLSR.

6.8.7 MID set

The MID set is populated and maintained based on received MID messages. However, the implementation of the MID parsing contains an extension to the the functionality described in the RFC.

As MID messages are considered to be static, one can save control-traffic overhead by using a high emission interval on these. Since all nodes use their main address as originator in OLSR messages, a temporary lack
Figure 6.12: A situation where B registers the addresses of A as two separate nodes before receiving a MID from A if A uses interface1's address as its main address.

of MID information in new nodes joining the MANET should not cause any problems. Before receiving a MID message from a multi-interfaced host, one can still communicate with the host through its main address.

Due to link sensing, neighbors announce the interface address of neighbors when announcing links in HELLO messages. This might not be the main address of a node. In the case illustrated in figure 6.12, node A uses the address of its interface1 as its main address. It will therefore set this address as the originator address in all messages. Node B will however announce a link to the address of A's interface2. This will cause node C to register node A as two separate nodes, one symmetric neighbor based on the main address of node A and one two-hop neighbor via B to the address of A's interface2. Upon receiving a MID message, the duplicate two-hop neighbor will reside in the two hop table until timed out. This is not a desired functionality. In olsrd the two-hop neighbor set is checked for possible duplicates when first registering a MID entry. This means for every alias address in the MID message, the two-hop table is checked for entries registered on this address. Such an entry is deleted immediately.

6.8.8 Route calculation

Route calculation should be seen as a separate entity from the actual routing table update. Olsrd keeps its own internal understanding of the OLSR routes added to the kernel, and all kernel update code is kept separate. The kernel update code is somewhat OS dependent, and keeping all OS dependent code separated from the OLSR core makes porting easier.

When recalculating routes, one has to keep an understanding of the current routes added. This is to be able to let the interception of the new and current sets of routes stay in the kernel routing table. The way this process is implemented in olsrd can be outlined as the following steps:

- The current routes is the set \textit{current routes}.
- Calculate new routes and add them to the \textit{new routes} set.
- Remove the routes \( \text{current routes} - (\text{new routes} \cap \text{current routes}) \).
- Add the routes \( \text{new routes} - (\text{new routes} \cap \text{current routes}) \).
- Set \textit{current routes} = \textit{new routes} set.
The actual process of calculating the routes is explained in section 3.8. All route calculation is done in the files src/routing_table.c and src/process_routes.c, while the Linux kernel update code can be found in src/linux/kernel_routes.c.

### 6.9 Declaring state

The cornerstone in a link-state routing protocol is the declaration of the local nodes link-state. In addition to a neighbor-sensing mechanism, nodes build their understanding of the topology based upon a set of link tuples. These tuples describe links between nodes. This information is passed between nodes in OLSR control traffic. In olsrd, entities register message generation functions with the scheduler as shown in section 6.7. All OLSR-message generation functions are located in the files src/generate_msg.c and build_message.c.

#### 6.9.1 Generating HELLO

As an example of HELLO generation, consider the scenario depicted in figure 6.13. This example is very similar to the one in section 3.6.1, but since this functionality is rather important we will go over it again. Node A has selected node B as a MPR, and C, D, and E are symmetric neighbors of A. A HELLO message generated by node A for interface 1 would contain:

```
HELLO MESSAGE HEADER
TYPE: MPR
   B = interface address

HELLO MESSAGE HEADER
TYPE: SYMMETRIC
   C = interface address
   D = main address
```

Note that since the HELLO is to be sent on interface 1, B and C’s main addresses are not used. Instead the addresses of the interfaces A has registered in its link set are used. However, the main address of D is used since A has no link to D via interface 1. To create this message the link set is queried for all links registered on the interface the HELLO is to be sent on. The Link set is also queried for all links on other interfaces, and the corresponding neighbor entries are used to retrieve the main addresses for these neighbors. Finally all entries are grouped by link and neighbor status.

A HELLO message generated by node A on interface 2 would contain:

```
HELLO MESSAGE HEADER
TYPE: MPR
   B = main address

HELLO MESSAGE HEADER
TYPE: SYMMETRIC
   C = main address
   D = interface address
```

In olsrd, one has the option to use different intervals on HELLO messages based on whether or not the interface is wireless. This is because an interface connected to a wired network is much less likely to experience a high degree of mobility on the link. It is considered an optimization to reduce overhead. The intervals of wired and wireless interfaces are set separately in the configuration file. Due to this possibility A can send HELLO messages on an interval of 2 seconds on the wireless interface while using an interval of 10 seconds on the Ethernet interface.
6.9.2 Generating TC

The TC message is the actual link state announcement in OLSR. Within core OLSR functionality terms, a TC consists of all MPR selectors of the local node. This information is kept in the MPR selector set. In addition, an ANSN sequence number is maintained describing the freshness of the announced MPR selector set as explained in section 3.7.1. If a node has no MPR selectors, it needs not send TC messages. However, nodes do transmit empty TC messages for a given interval after the MPR selector set goes empty. The timer controlling the emission of empty TC messages is declared in mpr_selector_set.h as send_empty_tc. If this timestamp is higher than the current timestamp, TC messages are generated even if no MPR selectors exist. The TC message generation can also be triggered by changes in the MPR selector set. When such a change is detected, the trigger registered with the TC generation function in the scheduler is set to 1, thus causing the scheduler to execute the TC generation function at next poll regardless of the registered interval.

Here is a snippet of code from the olsr_time_out_mprs_set function in the file src/mpr_selector_set.c where a MPR selector entry that times out is removed:

```c
if(olsr_timed_out(&mprs->MS_time))
{
    /* Dequeue */
    mprs_to_delete = mprs;
    mprs = mprs->next;

    DEQUEUE_ELEMENT(mprs_to_delete);

    mprs_count--;

    /* Delete entry */
    free(mprs_to_delete);
    changes = UP;
}
```

The changes variable is the trigger to create a TC message immediately. By setting changes = UP a TC message is created and transmitted at the next scheduler poll.
6.9.3 Generating MID

In olsrd, MID messages are created based on the interfaces configured at startup. A list of entries describing interfaces on which olsrd is running is available through the global pointer ifnet. This is a pointer to a struct interface, which is declared in src/interface.h. By traversing this linked-list, one can gain information about all interfaces running, olsrd and this is the way MID messages are generated. As of now, all interfaces must be added at startup, either in the configuration file or at the command line. Olsrd will then use those of these interfaces that are properly configured. This means that the MID message data is actually static, but it is generated every time a MID message is to be sent. This is due to the possibility of dynamic adding and removal of interfaces in the future.

MID messages could also be used to declare all IPv6 addresses of network interfaces on which olsrd runs. This is however not implemented at the current time. For now only one IPv6 address is used per interface.

6.10 Forwarding traffic

As explained in section 3.4.2, OLSR uses a default forwarding algorithm to retransmit traffic using the MPR scheme. This algorithm ensures that messages are only forwarded on interfaces on which one has a link to at least one MPR selector. The duplicate table is used to prevent that the same message is forwarded or processed, twice. The default forwarding algorithm is available as the function:

```c
int olsr_forward_message(union olsr_message *, /* The message */
    union olsr_ip_addr *, /* Originator address */
    olsr_u16_t, /* Sequence number */
    struct interface *, /* Receiving interface */
    union olsr_ip_addr */ /* Last hop sender */
```

This function is declared in src/olsr.h and implemented in src/olsr.c. The function both updates the duplicate table and forwards the message, if it is to be forwarded, according to the default forwarding rules.

6.11 Future work

In olsrd, all core OLSR functionality has been implemented and well tested for small networks. The future work on this functionality will be optimization and bug fixing. But new features, concerning this code, are planned to be added in future versions. This includes dynamic addition and removal of interfaces to use and declaration of multiple IPv6 addresses belonging to the same interface, in MID messages.
Chapter 7

Implementing auxiliary functionality

“Act and you shall have dinner. Think and you shall be dinner.”
– Klingon Proverb

As explained earlier, RFC3626 divides OLSR functionality into one core section that must be covered by all implementations, and one optional auxiliary section consisting of extensions to the core functionality. All auxiliary functionality is covered by the UniK olsr daemon implementation. This chapter explains in detail the implementation of this functionality.

7.1 HNA

As explained in section 4.1, Host and Network Association messages allow hosts to announce themselves as gateways to external networks. Full HNA support is implemented in olsrd.

7.1.1 Generating HNA

In the configuration file the user specifies the networks that the node should announce itself as a gateway to. These networks are specified by network address and netmask for IPv4, or network address and prefix length for IPv6. If a host is to announce itself as a gateway to the Internet in an IPv4 routed network, the following entry must be added to the configuration file:

```
HNA 0.0.0.0 0.0.0.0
```

For IPv6, Internet gateways would set the following in the configuration file:

```
HNA6 :: 0
```

These local entries are kept in a local HNA table which can be updated dynamically at runtime. This way HNA entries to announce can be updated by some extension like explained in section 8.3.2.

A HNA message generation function is registered with the scheduler at startup. If there exist any entry in the local HNA set, HNA messages are generated and transmitted at the given interval. This interval is set to be rather large as default since HNA information is not assumed to be very dynamic. The emission interval can be set by the user in the configuration file. The local HNA set is implemented in src/local_hna_set.c.
7.1.2 HNA set

A HNA parser function is registered with the packet parser at startup. This function maintains the HNA set. All received HNA information that is valid is kept in this data-set. This means that multiple HNA entries can exist for the same network if multiple nodes announce themselves as gateways to the same network. If this happens, only the gateway closest, in terms of hop-count, is to be added to the routing table. All HNA entries are however, to be registered and processed. Most of this code is kept in the src/hna_set.c file.

7.1.3 HNA route calculation

All changes in the network topology must be considered as a possible change in the routes to HNA gateways. Therefore the HNA routes are recalculated rather often in a mobile context. As seen in section 6.6.3 all variables that signal a topology change in olsrd triggers a recalculation of HNA routes. HNA recalculation can also be triggered by setting the global variable changes_hna to UP.

HNA route calculation is a matter of calculating the closest gateway to all networks for which HNA entries exist. This set of routes is then compared to the current HNA routes, and differences are updated in the current set and in the kernel routing table much like the approach described in section 6.8.8.

Although not explicitly specified in RFC3626, HNA routes should be kept as stable as possible. This means that in a situation as displayed in figure 7.1, where multiple gateways are registered for the same network and all the gateways are registered with the same hop-count, route calculation should not “dangle” between these gateways. One gateway should be chosen, and this should be used as long as the given situation exists. In olsrd this is the default behavior due to the hashing of entries in the HNA set and the route calculations traversal of this set. The stability of routes over time, based on other factors than hop-count, is however not taken into consideration when calculating HNA gateways.

HNA route calculation is implemented in src/routing_table.c and src/process_routes.c.
7.1.4 HNA and IPv6

As HNA messages are made up of network addresses and netmasks, it would seem natural to replace the netmask used in IPv4 HNA messages (figure 4.2) with the network prefix length when using IPv6. As displayed in figure 7.2, this would only require 8 bits compared to the 128 bits required to construct an IPv6 netmask. At first this was the way IPv6 HNA messages was implemented in olsrd, but after some discussion on the MANET mailing-list, it was decided to use the netmask approach for IPv6 as well to keep RFC compliance.

7.2 Link hysteresis

OLSR link hysteresis is based on a variable maintained for all links describing their current link quality. If the link is not set to symmetric and the link quality value passes some upper threshold, then the link is marked as symmetric. If a link is marked as symmetric, and its quality value sinks below some lower threshold, the link is set to asymmetric. Most link hysteresis code can be found in src/hysteresis.c.

The link-hysteresis implemented in olsrd is based on the suggested exponentially smoothed moving average approach from RFC3626 section 14.3. To maintain the hysteresis variable, two rules are introduced as explained in section 4.3. These are to be applied to the link quality when packets arrives as expected or when a packet-loss is detected. Packet-loss is detected by two mechanisms. The first one is based on announced emission interval and the second one is based on sequence numbering.

7.2.1 A long period of silence

Detection of what is referred to as “a long period of silence” in the RFC, is implemented by making the following changes to the existing link-sensing mechanism.

In link tuples the following is recorded, in addition to the fields specified in the RFC:

- next_hello_timer - a timer describing within what time a new HELLO should be received on this link.
- last_htime - last recorded htime value from the neighbor (which this link connects us to).

Upon receiving a HELLO, the next_hello_timer is set to now + (received htime * 1.5), and the last_htime is set to the received htime. The htime value is multiplied by 1.5 to add some slack since various issues like scheduler poll rate and transmission queuing could delay the reception of the next HELLO. The link set is constantly checked for timed out entries with regards to the next_hello_timer. If a links next_hello_timer is timed out, a long period of silence is considered detected and the instability rule is
applied to that links $\text{link\_quality}$. The $\text{next\_hello\_timer}$ is then set to $\text{now} + \text{last\_htime}$ to be able to detect another long period of silence. Note that this time no slack is added.

### 7.2.2 Missing sequence number

To detect the loss of an OLSR packet, section 14.3 of RFC3626 suggests checking packet sequence numbers. This is the sequence number of the generic OLSR packet header, not any of the message sequence numbers. To implement this, a node keeps track of the last received OLSR sequence number from all neighbor nodes. If an OLSR packet is received with a sequence number bigger than $\text{lastseqno} + 1$ then the instability rule is applied.

### 7.2.3 Problems

The usage of these two mechanisms combined leads to a highly undesired situation. Consider a scenario where nodes A and B are running OLSR with link-hysteresis using the proposed values (scaling=0.5, upper threshold=0.8 and lower threshold=0.3). We will look at the situation from A’s point of view.

A has received a continues series of HELLO messages from B which has set A’s entry of $\text{B->L\_link\_quality}$ to 0.99. A then misses on one of B’s HELLO messages. This leads to the instability rule being applied on A’s registered $\text{B->L\_link\_quality}$ due to a detected “long period of silence”. A’s registered link quality for B would now be 0.5. This is not beneath the lower threshold, so no change in link state is set. But as soon as A receives the next HELLO message from B, a missing package is detected due to the packet sequence number of the received OLSR packet. Now the instability rule is applied to $\text{B->L\_link\_quality}$ again. This time it ends up at 0.25 which is below the lower threshold therefore the link status is recalculated and the neighbor is set to be asymmetric. Of course, $\text{B->L\_link\_quality}$ is recalculated immediately since the stability rule is to be applied upon every received HELLO, but this will not bring the quality up above the upper threshold. Upon the next received HELLO (in sequence) the quality will be high enough for the status to be recalculated.

This means that missing out on one HELLO packet causes a link, and therefore a neighbor if no other symmetric links to it exist, to be set to asymmetric, and the MPR set and routing table to be recalculated. This is a double-counting of a lost packet. In olsrd this is avoided by incrementing the OLSR packet sequence number corresponding to the neighbor whenever such a loss of a HELLO packet is detected by a long period of silence.

### 7.3 MPR redundancy

As explained in section 4.5, MPR redundancy specifies how many MPRs should at best cover every 2-hop neighbor. The redundancy parameter can be set in the configuration file, it defaults to 1.

The MPR calculation algorithms were modified to add redundancy in the calculation. While two hop neighbors used to have a boolean value $\text{covered}$, stating if the node was covered by a MPR, they now have an integer variable that is updated with the number of MPRs covering the node during calculation. MPR calculation becomes a rather complex algorithm when considering that the calculation must respect willingness, including always adding nodes with willingness $\text{WILL\_ALWAYS}$ and never adding nodes with willingness $\text{WILL\_NEVER}$, redundancy and optimization of the calculated MPR set as described in section 8.3.1 of RFC3626. The MPR calculation algorithms are located in the `src/mpr.c` file.

### 7.4 TC redundancy

RFC3626 specifies three levels of TC redundancy deciding the amount of information passed in TC messages. This indirectly also sets what nodes should generate TC messages. A TC redundancy of 0 means
that only MPR selectors should be listed in TC messages. This is the standard used in core-OLSR. A TC redundancy of 1 specifies that all selected MPRs should be listed in TC messages as well, and a setting of 2 specifies that all symmetric neighbors should be listed. The TC redundancy parameter can be set in the configuration file, and it defaults to 0. This value can vary on individual nodes in the MANET, and no transmission of the actual parameter value is needed. Nodes just store the announced links from the TC message with no consideration for the TC redundancy used by the sender.

Implementing TC redundancy was a matter of updating the TC generation function to add more than just the MPR selector set if the redundancy parameter is 1 or 2. The TC generation function is run by the scheduler regardless of the MPR selector set. It is up to the function itself to decide whether or not a TC should be sent. If a TC message has been built, containing one or more links or if the send_empty_tc timer is valid, the message will be sent. Because of this, adding TC redundancy was only a matter of checking the TC redundancy value when adding addresses to the TC message being built. The main TC generation function is located in the file src/packet.c.

7.5 Link layer notification

Link layer notifications can be used for a variety of interesting functionality in ad-hoc networks, but here they will be considered only for setting some lower threshold for accepting data traffic. The goal is to make sure links are relatively robust before processing traffic received on these links. In addition, link layer information could be used in the hysteresis calculation to calculate link quality.

The WLAN drivers in GNU/Linux implement a "spy" functionality where one can register MAC addresses of nodes for which the driver should collect link quality statistics. The user tool iwspy can be used to add spy nodes and view link statistics. The source of iwspy was the first place to look when the work on link layer information was started. The ability to retrieve and maintain link quality information from the drivers is implemented in olsrd in a limited fashion. Communicating with the WLAN driver is done using ioctl commands located in include/linux/wireless.h. The ioctl commands SIOCGIWSPY and SIOCSTIWSPY are used for fetching and adding spy nodes. Luckily, this registration of traffic quality works in ad-hoc peer-to-peer mode as well as in the managed mode used when connecting to access points. The downside is that the drivers by default, only implement support for up to eight simultaneous spy addresses.

As mentioned, spy entries are registered on MAC addresses. To get a hold of the MAC address of a neighbor the ARP[55] cache is queried. This is done using the SIOCGARP ioctl. Unfortunately, MAC addresses are not cached until traffic to the IP address of the remote host is initiated. Therefore, if the local node is to find the MAC address of its neighbor A, it would first query the ARP cache. If no traffic has been unicasted to A from the local node, no ARP cache entry would be located. In that case the local node needs to trigger an ARP WHOHAS query by unicasting some traffic to A. In olsrd the MAC lookup is done this way. If no MAC address is mapped to the IP address in the ARP cache, then some traffic to that IP is initiated. This is done by sending a single ICMP PING message to the IP address. A thread is spawned to build and send the message, this thread exits as soon as the packet is sent. Upon the next ARP cache query, the MAC address will be allocated if the IP address was reachable. The process is illustrated in figure 7.3.

A function that polls for updates in the quality values of the registered neighbors is registered with the scheduler to be executed at a low interval. As of yet, the quality values are only stored and displayed to the user, meaning that no real action is taken based on the values.

The values returned by the iwspy ioctl are contained in a struct defined in include/linux/wireless.h:

```
struct iw_quality
{
    __u8 qual; /* link quality (%retries, SNR, %missed beacons or better...) */
    __u8 level; /* signal level (dBm) */
    __u8 noise; /* noise level (dBm) */
    __u8 updated; /* Flags to know if updated */
```
The link layer notification source is located in `src/linux/link_layer.c`.

### 7.6 Future work

The link layer functionality is the only unfinished part of the OLSR implementation. RFC3626 describes link layer notification in a very loose way leaving many decisions to the coder and the user. As shown, the basic functionality for initiating, retrieving and maintaining link quality is implemented. But some action should also be taken based on the values retrieved. The link layer values could trigger updates of the link set or decide what packets the socket parser should discard. But further work is required to be able to know what thresholds to use.

Also, there is the problem of the maximum limit of MAC addresses one can register with the drivers. To overcome this, a simple recompilation of the drivers with a higher max-value, can be enough. But if running olsrd requires recompiling kernel drivers, the software could be too complicated to set up for many end users, this could however be an alternative solution for advanced users. Therefore the implementation should not be locked to a certain maximum value. A solution could be a scheme where neighbors selected as MPRs are prioritized if the maximum limit is reached when registering neighbors with the driver.

The MAC address query mechanism could also be improved. One can imagine a solution where the MAC address is fetched from the lower layers when receiving OLSR control traffic. This information could be injected into the ARP cache directly. Such a solution would also eliminate the ARP lookups when nodes initiate regular traffic to neighbors.
Chapter 8

Olsrd plug-ins

“Civilization advances by extending the number of important operations which we can perform without thinking about them.”
– Alfred North Whitehead, Introduction to Mathematics (1911)

As MANETs are an area for research and development, the ability to add extensions or change normal operation in implementations of routing protocols for such networks, provides a great way of testing new solutions.

The MPR flooding and default forwarding algorithm used in OLSR makes this protocol very interesting to extend. Normal MANET routing suffers from lack of broadcast and multicast solutions. By letting OLSR carry traffic, one can provide a broadcast solution that is optimized. The OLSR daemon will then work as a flooding relay agent for local applications. Already existing services that requires a broadcast mechanism can be used in a MANET routed by olsrd if using an olsrd plugin to flood broadcasted traffic. Such services include domain name service (DNS) [48], service discovery mechanisms and key distribution schemes. Utilizing such protocols in MANETs have been studied in [21] and [40]. Other interesting extensions can be updating OLSR parameters at runtime, based on traffic analysis, or creating visualizations of the network topology.

As modularity was one of the main goals when designing and implementing olsrd, the idea of easily extending the protocol led to the design of a plugin interface. In this chapter this interface, areas of usage and some example plugin implementations are covered. The plugin interface is described in detail in [61].

8.1 Plugins

Olsrd supports loading of dynamically linked libraries, called plugins, for generation and processing of private package-types and any other custom functionality. A dynamically loadable library (DLL) is a piece of executable code that contains functions and data. Unlike normal executables, DLLs are not “fully” linked after compilation. They are set up in a way that allows the actual linking to take place at runtime. An application can load and run functions from a DLL dynamically, therefore the library is said to be dynamically linked.

One of the big advantages of DLLs is that they can be used simultaneously by multiple processes, still only one instance of the library will be maintained in memory. These kind of DLLs are typically libraries of functions shared by many processes. An example would be a Graphical User Interface (GUI) library. This is however not something taken advantage of when using DLLs as plugins. Plugins provide new functions to an existing application without altering the original application. An illustration of this is shown in figure 8.1. Olsrd uses DLLs in this fashion.

DLL functionality exist for all common operating systems. In Linux they are known as .so files while in
Microsoft Windows they are known as .DLL files.

8.1.1 Olsrd plugins

The plugin design was chosen for amongst others, the following reasons:

- No need to change any code in the OLSR daemon to add custom packages or functionality.
- Users are free to implement olsrd plugins and license them under whatever terms they like. Olsrd is GPL licensed meaning that any alteration of the olsrd code itself must in most cases, be publicly released.
- Plugins can be written in any language that can be compiled as a dynamic library.
- No need for people using extended OLSR functionality to rely on heavy patching to maintain functionality when new olsrd versions are released. The plugin interface will always be backwards compatible.

OLSR provides a default forwarding algorithm that allows for forwarding of OLSR messages of unknown types. This means that even if only a subset of the nodes in the network actually knows how to interpret a certain message-type, all nodes will forward them according to the MPR scheme. A wide variety of services designed for wired network environments rely on net-wide broadcasts. Services that needs to broadcast/multicast data can encapsulate data in a private OLSR message-type using an olsrd plug-in as illustrated in figure 8.2. However, some special considerations must be taken if such a plugin is to be transparent to the application. This is discussed in section 8.4.

The design of the various entities of olsrd allows one to easily add special functionality into most aspects of the program. One can both register and unregister functions with the socket parser, packet parser and scheduler, and one can update many variables, manipulate incoming and outgoing traffic and more. This opens up for possibilities like intercepting current operation and replacing it with custom actions. As an example, a plugin can provide its own HELLO message generation and parser functions. The plugin can unregister the default functions used by olsrd and replace them with its own. This relationship is illustrated in figure 8.3.

The modular design of olsrd really shows its strengths when dealing with plugins. A plugin can do things like establishing blocking sockets for communication of its own, without blocking olsrd operation. This is because the plugin can register its sockets with the socket parser in olsrd where the socket will be part of the main select set.
Figure 8.2: An example of how a plugin can enable the OLSR daemon to work as a relay for broadcasting. The Local application and the plugin communicate using interprocess communication.

Figure 8.3: A plugin can manipulate virtually every part of the olsr daemon.
8.2 The plugin interface

For a plugin scheme like this to work, one needs a well defined and easy-to-expand interface for communication between the OLSR daemon and the plugin. The interface should be well defined so that a plugin always knows what to expect from the daemon, and the daemon always knows what to expect from the plugin within some given set of functions. Still, the design should be flexible enough to allow for extending the functionality while keeping backwards compatibility.

The actual data that must be set up between the application and the plugin are pointers to variables and functions. The olsrd plugin interface is mainly based upon the function:

```c
int olsr_plugin_io(int cmd, void *data, size_t size)
```

This function is similar to the ioctl(2) function in syntax. One passes a command and a pointer to some allocated memory and the size of the allocated memory area. The return value indicates success or error, while actual data is put or read from the memory buffer pointed to by *data. This function is implemented in src/plugin.c. The function is in reality just a big switch statement. All defined commands must be implemented as a case statement in this switch. The command GETF__OLSR_REGISTER_SCHEDULER_EVENT retrieves a pointer to the olsr_register_scheduler_event which is used to register an event with the olsrd scheduler. The following case statement, implemented in plugin.c, takes care of setting up the pointer to the function:

```c
    case(GETF__OLSR_REGISTER_SCHEDULER_EVENT):
      ptr = &olsr_register_scheduler_event;
      memcpy(data, &ptr, size);
      break;
```

Here `data` is the pointer provided by the caller of the function.

Now let us look at an example of how a plugin can use a function implemented in olsrd. The function `get_msg_seqno()` returns the next message sequence-number for olsrd to use when transmitting an OLSR packet. If we want to be able to use this function in our plugin we will typically execute something like:

```c
    int olsr_plugin_io(float cmd, void *data, float size)

    /* Define this function-pointer somewhere */
    olser_u16_t (*get_msg_seqno)();

    /* Retrieve the function pointer */
    if(o1sr_plugin_io(GETF__GET_MSG_SEQNO, &get_msg_seqno, sizeof(get_msg_seqno)))
      {
        get_msg_seqno = NULL;
        return 0;
      }
```

The available commands (like GETF__GET_MSG_SEQNO) are defined in src/olsr_plugin_io.h in the olsrd codebase. All commands that fetches function pointers starts with GETF__. While all commands that fetches data-pointers starts with GETD__. No commands will be removed from this header-file, but new ones can be added. One should therefore always use the most recent version of this file to have access to as many functions and variables as possible when implementing a plugin.

To be able to access the olsr_plugin_io function, the plugin needs to be initialized from olsrd. The file src/plugin_loader.c implements the plugin loader code. For the plugin loader to be able to set up the needed pointers, the plugin must provide the following function (in addition to some variables and other functions):

```c
    int register_olsr_data(struct olsr_plugin_data *)
```
This function is called from the olsr plugin loader passing a pointer to a struct `olsr_plugin_data` which contains the pointers to olsr functions that the plugin needs to use to be able to set up all needed data-pointers. After this, the plugin is responsible for fetching all needed pointers from the olsr daemon. The process of initializing a plugin is illustrated in figure 8.4.

To make olsrd try to load a plugin at startup, the `LOAD_PLUGIN` directive is used in the configuration file.

### 8.3 Two example plugins

While extended functionality as described in chapters 11 and 12 is implemented as plugins, two plugins that perform more trivial tasks are implemented, more or less, as example code. Here follows a brief explanation of what they do and how they are designed.

Both plugins are part of the olsrd source code package available for download from [http://www.olsr.org](http://www.olsr.org). The example plugin source code resides in the `lib/` directory relative to the olsrd source code root directory.

#### 8.3.1 The power-status plugin

This plugin is to provide a solution where the power-status of nodes running the plugin in the MANET is distributed and registered. This information is made available to the user and other processes through IPC using a TCP socket. The plugin should not effect nodes running without this functionality.

A node is to periodically flood the network with a custom packet containing the following information:

- Whether or not the node is battery powered.
- Estimated lifetime left on the battery, if battery powered.
- Percentage of power left on the battery, if battery powered.

This should result in a scenario where all power-status enabled nodes have an up-to-date understanding of the power-status of all other nodes running the plugin. Even though the power-status enabled nodes might...
only be a subset of the nodes in the MANET, the default forwarding algorithm will ensure diffusion of the
information.
To take advantage of OLSRs default forwarding scheme, the power information, extracted from /proc/apm, has to be transmitted as an OLSR message. This message-format is displayed in figure 8.5. The message is encapsulated in a regular OLSR message header with a message type from the private message types (128-255). To transmit power-status messages on a periodic interval, a message generation function is implemented. This function is registered with the olsrd scheduler at plugin initialization. The function polls the /proc/apm file for power-info and builds a message based on the information. This message is then flooded through olsrd.

To keep an up-to-date database of power-info, an information repository similar to those used in olsrd, is implemented. This is based on hashed linked lists with statically allocated root elements as explained in section 6.6.2. A function that traverses the information repository and removes timed out entries is registered with the olsrd scheduler to run at a given interval. Since its not critical that timed out entries are removed as soon as possible, this function is not registered to be executed at every olsrd scheduler poll.

A message parse function is registered with the olsrd message parser at plugin initialization to receive all incoming power-status messages. This function updates the information repository based on the contents of incoming packages. The function is also responsible for forwarding the message.

To be able to access the stored power information, an application can connect to the plugin using IPC. IPC is done over a TCP socket via the loopback interface. The plugin generates regular output where all registers power-info is listed on a fixed interval. This communication is one-way, but since we want clients to be able to connect to the plugin at any time, the plugin must register a server socket with olsrd to listen for incoming connections and set up connections based on this. The socket listens for connections on TCP port 8888 and only accepts connections from the local host (127.0.0.1). For easy access to the information the user can initiate a telnet session to port 8888 on the loopback device, this way all information registered will be displayed in the users terminal.

### 8.3.2 The dynamic Internet gateway plugin

This plugin is meant for real life usage and is not really created for the sake of the example. But since it is a relatively light-weight plugin that performs tasks not related to message flooding, it is included as an example here.

Nodes in a MANET might dynamically obtain and lose Internet connectivity through interfaces not participating in the MANET routing. A typical scenario would be a laptop that might be connected to the Internet through an Ethernet link for a limited time while participating in a MANET through a wireless interface.

A plugin that dynamically updates the HNA information announced by the local node has been implemented. This plugin checks if the local node has an Internet-connection and updates the local HNA set based on this. This implementation is a good example of using plugins for other tasks than packet transmission. Combining this plugin with an automatic network cable detection daemon, such as NetPlug[50], would be a good idea. Only IPv4 is supported as of now.

The main object of this plugin is to poll for an Internet route and add or remove such a route from the local HNA set if a change is detected. An Internet-connection is identified by a default gateway with a hop-count of 0. This means that a route to 0.0.0.0/0 with metric 0 is considered an Internet route. Since olsrd sets a
hop-count/metric bigger than 0 on all routes, this plugin will not react to Internet gateways added by olsrd. To poll for route updates, a function that searches the kernel routing table for a default gateway is registered with the olsrd scheduler to be executed regularly on a given interval. If a new Internet route with metric 0 is discovered, the plugin will add this entry to the local HNA set by calling the function:

```c
void
add_local_hna4_entry(union olsr_ip_addr *net, union hna_netmask *mask)
```

This function has been fetched from olsrd through the plugin interface. Whenever such a registered Internet route is removed from the kernel routing table the local HNA entry is also removed using the function:

```c
void
remove_local_hna4_entry(union olsr_ip_addr *net, union hna_netmask *mask)
```

This enables nodes to act as Internet gateways whenever they have Internet connectivity not set up by olsrd. The dynamic Internet gateway plugin offers IPC to read debug output. Just like with the power-status plugin, all communication is outbound, but since clients should be able to connect at any time, the IPC server socket is registered with the socket parser of olsrd. The IPC socket listens on TCP port 9999, and only allows connections from the local host(127.0.0.1). A user can telnet to port 9999 at 127.0.0.1 to read the debug info.

### 8.4 Future work

The plugin interface on its own behalf does not require much further work. It will typically be expanded when new plugins are added, but it will always keep backwards compatibility. Adding new commands to the interface is only a matter of defining a command in the `olsr_plugin_io.h` file, following the conventions described there, and adding the corresponding case match in the main `switch` statement in the `plugin.c` file.

What would be a very interesting future project, with regards to using the plugin interface, would be to create a broadcast and possibly multicast, plugin that is totally transparent to applications. This would be a plugin that intercepts all outgoing broadcast/multicast traffic and forwards it using the routing protocol. The plugin must also have a transparent way of delivering such received messages to the local applications. This approach requires intercepting normal IP routing and would probably require altering the IP implementation of the operating system, but it should definitely be possible.
Chapter 9

A GUI front-end

“Seeing is believing.”
– Italian proverb

Olsrd directs all output information to the Standard Output (STDOUT) and the system log facility. The amount of information written to STDOUT depends on the debug level at which olsrd is running. For testing and debugging purposes this kind of output is sufficient, but for end-users, watching a lot of text information scroll in a terminal window might not be the best solution. A more interactive interface would make it easier for the user to follow olsrd operation.

9.1 A Graphical User Interface

To offer a more intuitive and interactive way to get status updates from olsrd, a Graphical User Interface (GUI) client-application has been implemented. This is in reality an OLSR packet analyzer that receives all OLSR traffic and maintains its own internal databases. Based on this, information about the MANET is displayed in various forms.

The GUI application is very much work in progress, and lots of features could be added and improved. It is not very well suited for debugging olsrd as it is not 100% reliable due to lack of extensive testing. As an example, the command `watch 'route -n'` in a shell will display a dynamically updated list of kernel routes which is fully reliable whereas the routes displayed in the GUI could be erroneous.

All GUI source-code is located in the `frontend/src` directory in the olsrd source code hierarchy.

9.1.1 Design

As seen in figure 9.1, the GUI consists of a window containing a list of tabs representing the different screens available to the user. Every one of these screens contain different information set up in an intuitive and sometimes interactive way. This section presents the different screens. Note that the screens Traffic and About has no content at current time.

The main screen (figure 9.1) is the default screen displayed when starting the GUI application. A dynamically updated list of all known nodes is displayed. The list shows various information about the nodes. Upon clicking a list entry, possible MID, MPR and HNA entries are displayed in the lower part of the screen.

The Packet screen (figure 9.2) lets the user “sniff” OLSR traffic in real-time. When the sniffer is activated the last 20 received packets is displayed with type, sender and size in the left-side list. Selecting a packet in the list displays the packet content on the right side area of the window. The content can be displayed in hexadecimal or decimal form.
Figure 9.1: The main screen displays a list of known nodes and information about them.

Figure 9.2: The packet screen offers OLSR packet sniffing.
The *Routes* screen (figure 9.3) displays a dynamically updated list of all OLSR routes in the kernel routing table. As of yet, no interaction is available here.

The *Settings* screen (figure 9.4) displays some of the current settings used by the local olsr daemon. This includes various intervals, IP version and main address.

### 9.1.2 Implementation

The GUI application is implemented using the *Gimp Tool Kit (GTK)* 1.2 and relies on GTK shared libraries to run and GTK development libraries and header-files to compile.

The internal design of the client is illustrated in figure 9.5. The client mainly has two event entities. The packet parser and the *GUI event handler*. The packet parser receives all OLSR traffic from olsrd and updates the local repositories which again triggers necessary updates of the GUI components. The GUI event handling is maintained by GTK where all GUI component events are registered as *signals* using the function:

```c
void
gtk_signal_connect(GObject *object,
                   const gchar *name,
                   GSignalFunc func,
                   gpointer func_data);
```

where the action *name* on the widget *object* triggers the function *func*.

Instead of running the GUI event loop and packet parsing in different threads, a timer function offered by the GTK library is used to poll for OLSR traffic from olsrd. The function that polls OLSR traffic is added as a GTK timeout using the function:

```c
void
gtk_timeout_add (_GD *object, gpointer user_data);
```
Figure 9.4: The settings screen displays information about the local nodes settings.

Figure 9.5: The internal design of the olsrd GUI client.
gtk_timeout_add(guint32 interval, 
    GtkWidgetFunction function, 
    gpointer data);

The timeout of the local information repositories is also added as a timeout using this function.

9.1.3 Communicating with olsrd

The GUI front-end communicates with olsrd through IPC over a TCP socket. The GUI is therefore totally separated from the olsr daemon, and olsrd does not depend on the GUI in any way. The GUI is offered as an add-on tool.

There is no interactivity in the communication except from the actual process of connecting. This means that the GUI cannot provoke any action in olsrd other than the actual connection setup. All communication floats from olsrd to the GUI application. As of now, only GUI connections from the local host are allowed.

When a GUI client connects to olsrd, a special packet containing various information about the settings used by the local node and OLSR routes set up is generated and sent to the client. After this all OLSR traffic is forwarded to the GUI in addition to special packets sent to update the GUI on route changes.

9.2 Future work

As stated earlier, the GUI is work in progress and has not been highly prioritized in the development process. It is offered as an extra tool for end-users and as a pointer to what a complete GUI solution could be like. There are many functions that could be added and many things could be handled better. These include:

- A structured display of parsed packet content in the sniffer section.
- A traffic analysis section where various statistics are displayed.
- Setting of olsrd variables like emission intervals in real-time through the GUI.
- Graphically displaying the topology.
- Move the olsrd communication interface code out to a plugin.
Chapter 10

Usage analysis

“Statistician: A man who believes figures don’t lie, but admits that under analysis some of them won’t stand up either.”
– Evan Esar (1899 - 1995), Esar’s Comic Dictionary

Olsrd is a daemon, or service, that constantly runs on all nodes participating in the OLSR routing domain. Therefore it is important that it does not claim more resources than necessary. On modern generic PCs, as we will see later, the amount of processing time used by olsrd is microscopic. But olsrd is also aimed at small embedded systems that in no way provide the same processing power that regular PCs does. However, in MANETs the most scarce resource is probably bandwidth. As OLSR is a pro-active protocol it generates a near to constant overhead. In this chapter we will, in addition to looking at local resource usage, take a brief look at the network resources olsrd claims.

10.1 Local resource usage

The testbed used for the development grew from three Intel i386 based nodes to eight nodes including one ARM based and one MIPS based. Local resource usage tests has been done on the machines described in table 10.1.

To document the local resource usage, a topology including multi-homed nodes and an Internet Gateway, was used. The scenario is illustrated in figure 10.1. The scenario includes mobility created using the netfilter in the Linux kernel. This is manipulated using the user-space iptables tool. A script was created that would block various MAC addresses at given intervals to create a sense of mobility. Even though this does not create an authentic mobility scenario, regarding actual radio traffic, it does lead to topology changes.

When testing, the 0.4.4 release of olsrd was used. In all tests the OLSR parameters suggested in section 18 of RFC3626 have been used.

<table>
<thead>
<tr>
<th>Node</th>
<th>Model</th>
<th>CPU</th>
<th>Speed</th>
<th>Interface(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Desktop PC</td>
<td>1 Intel Pentium2</td>
<td>350Mzh</td>
<td>1 Ethernet NIC</td>
</tr>
<tr>
<td>F</td>
<td>Desktop PC</td>
<td>1 Intel Pentium3</td>
<td>500Mzh</td>
<td>2 Ethernet NICs</td>
</tr>
<tr>
<td>A</td>
<td>Laptop PC</td>
<td>1 Intel Pentium3 m</td>
<td>1000Mzh</td>
<td>1 Ethernet 1 WLAN</td>
</tr>
<tr>
<td>H</td>
<td>LinkSys WRT54G</td>
<td>1 MIPS(BCM3302)</td>
<td>125Mzh</td>
<td>1 WLAN</td>
</tr>
</tbody>
</table>

Table 10.1: The systems used for CPU usage testing. Node refers to what node in the scenario depicted in figure 10.1 the system was used as.
10.1.1 CPU usage

In the CPU usage test olsrd was ran for 1 hour on the network illustrated in figure 10.1. The CPU time used was measured using the Unix utility `time`. Time returns the following values:

- **Real** - The elapsed (real) time between invocation of the application being timed and its termination.
- **User** - The User CPU time, equivalent to the sum of the `tms_utime` and `tms_cutime` fields returned by the `times(2)` function for the process in which utility is executed.
- **Sys** - The System CPU time, equivalent to the sum of the `tms_stime` and `tms_cstime` fields returned by the `times(2)` function for the process in which utility is executed.

<table>
<thead>
<tr>
<th>Model</th>
<th>Real</th>
<th>User</th>
<th>Sys</th>
<th>CPU usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkSys WRT54G(MIPS 125Mhz)</td>
<td>61m 24s</td>
<td>8.69s</td>
<td>38.80s</td>
<td>1.29%</td>
</tr>
<tr>
<td>Desktop PC(P2 350Mhz)</td>
<td>63m 15s</td>
<td>1.00s</td>
<td>0.33s</td>
<td>0.04%</td>
</tr>
<tr>
<td>Desktop PC(P3 500Mhz)</td>
<td>63m 0s</td>
<td>1.25s</td>
<td>0.47s</td>
<td>0.05%</td>
</tr>
<tr>
<td>Laptop PC(Intel P3m 1000Mhz)</td>
<td>62m 32s</td>
<td>1.04s</td>
<td>2.12s</td>
<td>0.09%</td>
</tr>
</tbody>
</table>

Table 10.2: CPU time used by different nodes. The rightmost column shows the percentage of total CPU time used by olsrd.

The results of the test is shown in table 10.2. One can see differences on CPU usage based on the nodes placement in the topology. Even though node A is the one with the fastest CPU, it is placed in a very central position leading to lots of forwarding and recalculation of neighborhood and MPRs. And so it uses slightly more CPU time than node G which has a much slower CPU but never experiences any updates in neighborhood or needs to do any forwarding. One can also see that olsrd runs without any problems, on CPUs such as the 125Mhz MIPS used in the wireless router(WRT54G). A CPU usage of 1.29% on such a system is acceptable. It must also be noted that the WRT54G runs a highly experimental GNU/Linux system[34].

---

1The `tms_utime` field contains the CPU time spent executing instructions of the calling process
2The `tms_cutime` field contains the sum of the `tms_utime` and `tms_cutime` values for all waited-for terminated children.
3The `tms_stime` field contains the CPU time spent in the system while executing tasks on behalf of the calling process
4The `tms_cstime` field contains the sum of the `tms_stime` and `tms_cstime` values for all waited-for terminated children.
1 hop neighbors | 2 hop neighbors | topology | Bytes allocated
0 | 0 | 0 | 9846
1 | 0 | 0 | 10580
2 | 0 | 0 | 10848
3 | 0 | 0 | 11276
4 | 0 | 0 | 12192
4 | 1 | 0 | 13980
4 | 2 | 0 | 15000
4 | 3 | 0 | 16788
4 | 3 | 1 | 17934

Table 10.3: Memory used by one instance of olsrd with different topologies. All 2 hop neighbors are only reachable through one 1 hop neighbor. Some nodes are multi-homed.

10.1.2 Memory usage

A daemon is meant to run for long periods of time. Often it will run as long as the system is powered. Therefore it should be conservative in memory usage and the developer must be careful to eliminate all memory leaks. A memory leak is what happens if one allocates memory without freeing it when it is no longer to be used. This usually means that one has no pointers pointing to the allocated memory any more. On modern operating systems memory leaked by user-space applications is freed when the application terminates. But as daemons usually do not terminate until the system is rebooted, a memory leak in a daemon is quite serious and could lead to severe degradation of system performance. To detect memory leaks in the olsrd code, the tools memproof and valgrind has been used. In the CPU usage test illustrated in figure 10.1 node D ran valgrind for a 15 hour test. No memory leaks were reported.

The binary executable produced when compiling olsrd, without debugging or profiling enabled, is 125Kb in size. If stripped the size shrinks to 100Kb. The number of bytes allocated by olsrd is of course relative to the network topology. Table 10.3 displays memory usage in some simple scenarios measured using memproof. The number of bytes allocated is dynamic even in static topologies due to the internal processing in olsrd. So the numbers of bytes showed are all the average of the memory used. As one can see olsrd uses a very modest amount of memory.

10.1.3 Optimizing code - profiling execution

Looking at resource usage in the development process should also include looking at ways to optimize code. Optimization does not only include writing “better” code, it includes detecting which parts of the code that is executed most frequently and optimize these based on this information. Analyzing processes in this way is called profiling.

To generate profile reports the GNU profiler, gprof, has been used. To use gprof one must compile and link the code with the -pg switch. Compilation/linking has to be done using gcc. When running a program compiled with profiling enabled, an output-file, gmon.out, is automatically created containing information about the flow of the execution. This file is then fed into the gprof application to create a human readable profile of the execution.

In a 15-hour test, using the topology illustrated in figure 10.1, node C ran a version of olsrd compiled for profiling. Here are some excepts from the results:

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time seconds</td>
<td>seconds</td>
<td>calls</td>
<td>s/call</td>
</tr>
<tr>
<td>12.96</td>
<td>1.68</td>
<td>1.68</td>
<td>761491</td>
</tr>
</tbody>
</table>

5 strip is a tool that discards all symbols from object files. It is often used on embedded systems where storage and RAM (often the same device) are scarce resources.
### Explanation of data:

This was the 25 functions with the highest CPU usage. Here follows the timeout functions:

<table>
<thead>
<tr>
<th>cumulative seconds</th>
<th>ms/call</th>
<th>self seconds</th>
<th>ms/call</th>
<th>calls</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2804.80</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_check_dup_table_proc</td>
</tr>
<tr>
<td>2299.84</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>parse_packet</td>
</tr>
<tr>
<td>3114.14</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_check_dup_table_fwd</td>
</tr>
<tr>
<td>1970.12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_update_dup_entry</td>
</tr>
<tr>
<td>2286.78</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_hyst_calc_stability</td>
</tr>
<tr>
<td>1488.64</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>tc_ghstruct</td>
</tr>
<tr>
<td>3347.09</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_input</td>
</tr>
<tr>
<td>2299.84</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>lookup_link_entry</td>
</tr>
<tr>
<td>1573.43</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>mid_ghstruct</td>
</tr>
<tr>
<td>3114.14</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_forward_message</td>
</tr>
<tr>
<td>5910.66</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>if_ifaceaddr</td>
</tr>
<tr>
<td>2299.84</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>update_hysteresis_incoming</td>
</tr>
<tr>
<td>1267.56</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_ip_to_string</td>
</tr>
<tr>
<td>2062.55</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_hashing</td>
</tr>
<tr>
<td>4294.18</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>me_to_double</td>
</tr>
<tr>
<td>1842.13</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_add_dup_entry</td>
</tr>
<tr>
<td>1488.64</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_process_received_tc</td>
</tr>
<tr>
<td>2020.33</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_printf</td>
</tr>
<tr>
<td>2582.81</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_lookup_neighbor_table</td>
</tr>
<tr>
<td>2602.06</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_lookup_mpr_cache</td>
</tr>
<tr>
<td>5416.55</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>buffer_forward</td>
</tr>
<tr>
<td>1846.68</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_move_route_table</td>
</tr>
<tr>
<td>13014.99</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>olsr_malloc</td>
</tr>
</tbody>
</table>

Explanation of data:

- **% time** - The percentage of the total running time of the program used by this function.
- **cumulative seconds** - A running sum of the number of seconds accounted for by this function and those listed above it.
- **self seconds** - The number of seconds accounted for by this function alone. This is the major sort for this listing.
- **calls** - The number of times this function was invoked, if this function is profiled, else blank.
- **self ms/call** - The average number of milliseconds spent in this function per call, if this function is profiled, else blank.
- **total ms/call** - The average number of milliseconds spent in this function and its descendants per call, if this function is profiled, else blank.
- **name** - The name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis, it shows where it would appear in the gprof listing if it were to be printed.
Remember that all timeout-functions are called at every scheduler poll (default set to 0.1 seconds). With over half a million calls to the timeout functions, they are still reported to have used 0 seconds of CPU time. This is obviously wrong. This problem is caused by the fact that the scheduler runs in a thread of its own. On Linux kernels, gprof will only profile the main thread of a program, unless certain additional techniques are used[41]. But for all functions related to processing of incoming traffic (the main thread), the CPU usage time is also recorded. The initiator of all these is the listen_loop function which listens for incoming traffic. This function therefore is reported with a total ms/call not far from the total CPU usage time reported to be 12.96 seconds. This again is not including the CPU time from the scheduler thread.

The most CPU intensive function is reported to be mid_lookup_main_addr. This is the function called to check if an IP address is an “alias” for a node. This means that the address is one of the addresses listed in MID messages from this node, not the nodes main address. This function is frequently used by the link-sensing mechanism, HELLO message parsing and route calculation. If the address is not registered in the MID set, the function will traverse all entries in the set, and because of this it is quite CPU intensive. This function could be optimized by maintaining a “reverse” MID set where all alias addresses are indexed.

The second most CPU intensive function of the main thread, is reported to be olsr_check_dup_table_proc. This is the function called for all received messages to check for duplicate processing. If the message is not registered in the duplicate set this function will traverse the entire set, which can grow quite large due to the 30 seconds of message-caching.

The third most CPU intensive function in the main thread, is parse_packet. This function parses all incoming OLSR packets into messages and calls the corresponding message parser function. This operation includes error-checking and traversal of registered message parse functions. In the binary used for this test, quite a lot of debugging information was generated by this function, this is removed by default in the current olsrd version.

Out of the next three functions, two are related to the duplicate set which we have seen that contains lots of entries. But the last function is related to the hysteresis calculation which is performed on all incoming HELLO messages. The entire function is implemented as:

```c
float
olsr_hyst_calc_stability(float old_quality)
{
    return (((1 - hyst_scaling) * old_quality) + hyst_scaling);
}
```

The fact that this function is reported to have such a high CPU usage is rather strange since it is only called from the function update_hysteresis_incoming that does much more processing itself. To optimize this function it has now been declared as inline, which makes the compiler treat it like a macro, expanding calls to code rather than setting up a full function call stack.

Profiling has been used during the implementation process, so that the functions that are called most frequently are implemented with efficiency in mind.

### 10.2 Network resource usage

When designing distributed services such as routing protocols, the overhead traffic generated by control messages should always be taken into account. In MANETs bandwidth is a very scarce resource both due to the limitations of the wireless technology and due to the effects of multi-hop which include heavy interference. Hence it is very important to make sure that routing traffic overhead is kept to a minimum. In pro-active protocols the overhead is relatively constant, and it is therefore possible to measure the overhead of different topologies without having to take to much considerations on creating mobility. However, as TC messages in OLSR are triggered by mobility, the overhead is not completely unaffected by this factor.

When implementing a protocol such as OLSR, the network overhead is not that interesting since this is very much predefined by the design of the protocol. However, some simple tests have been done to show what
To measure the overhead traffic generated by olsrd the tcpdump tool was used to log all traffic and the trpr tool was used to parse these logs. These scenarios were tested:

1. A node on its own.
2. Two neighbor nodes.
3. Three neighbor nodes.
4. Three nodes “in a line”.
5. Four neighbor nodes.
6. Four nodes “in a line”.
7. Five nodes in a topology as illustrated in figure 10.2.
8. A bigger scenario including mobility similar to the setup illustrated in figure 10.1.

The traffic logging was done on the most centralized nodes in all scenarios. On test 8 traffic was logged at four different nodes. The results of tests 1-7 are displayed in table 10.4 while the results for test 8 are displayed in table 10.5.
Note that a RFC3626 incompliance that caused some extra unnecessary retransmissions was discovered in the implementation after these test had been done. However, this should not have too much impact on the results.

### 10.3 Real life testing

At the Wizards of OS 3 conference, introduced in section 5.7.1, bandwidth usage was measured in a network consisting of 20 nodes. The node recording the bandwidth was placed in a position where it had 11 symmetric neighbors. The overhead of control traffic is displayed in figure 10.3. The CPU usage from this test on a 1Ghz based laptop was:

```plaintext
real  5m6.348s
user  0m0.510s
sys   0m0.260s
```

This means that olsrd used 0.25% of the total available CPU time during this test. Profiling and debugging were enabled in the binary used. This is causing some extra CPU usage.

![Figure 10.3: Overhead of OLSR control traffic at the test network set up at WoS3.](image-url)
Chapter 11

Securing OLSR

“The author of the Iliad is either Homer or, if not Homer, somebody else of the same name.”
- Aldous Huxley

Today wired computer systems can be made secure to a high degree, but when it comes to wireless networks weak security is often used if any security measurements are taken at all. This affects the services running on wireless networks including MANET routing protocols.

In this chapter an extension to secure OLSR is presented. This extension only provides security extensions to OLSR and not the traffic being routed in the MANET. Also, the implemented solution only provides integrity and not confidentiality, although the solution is extendable and could include mechanisms to provide confidentiality.

As this thesis is not mainly focused on security, the concepts of encryption, key management and digital signatures will only be briefly presented. It is assumed that the reader is familiar with the concepts of confidentiality, integrity and availability.

11.1 Computer network security

As computers and computer communication has become an important part of the global trading and financial world, the need for secure communication has become crucial. Companies do business over the Internet and they transfer sensitive data from one branch to another over public telephone or data lines. Also ordinary people transmit private and sensitive information over the Internet when they shop or use Internet-based banking services. If these transactions are not properly secured, “everyone” can obtain information which can be used against one or both parts of the communication.

Without any security mechanisms, the Internet is vulnerable to a wide variety of attacks. Even with the security mechanisms used in todays Internet, attacks occasionally succeed in disrupting the function of the global Internet. Clearly, robust security mechanisms for the basic network services is needed in most kind of computer networks.

11.1.1 Security in WLAN

The Wired Equivalent Privacy protocol was supposed to provide wireless IEEE 802.11 based networks with the same amount of security as their wired counterparts. But WEP has several proved flaws in its architecture [19][63][59]. Because of the weak security provided by WEP and due to the fact that parts of MANETs might run on wired links where no encryption is used, some security mechanism could be provided by the routing protocol itself.
11.1.2 OLSR and security

For MANET, there are several security problems to be considered. Some of these issues are special problems related to the physical nature of the wireless links in the networks. Others are security problems, which also exists in the (wired) Internet.

Being a proactive protocol, OLSR periodically diffuses topological information. Hence, if used in an unprotected wireless network, the network topology is revealed to anyone who listens to OLSR control messages. In situations where the confidentiality of the network topology is of importance, regular cryptographic techniques such as exchange of OLSR control traffic messages encrypted can be applied to ensure that control traffic can be read and interpreted by only those authorized to do so.

In OLSR, each node is injecting topological information into the network through transmitting HELLO messages and, for some nodes, TC messages. If some nodes for some reason, malicious or malfunction, inject invalid control traffic, network integrity may be compromised. Examples of situations that may occur due to lack of data integrity functionality, are:

1. A node generates TC messages, advertising links to non-neighbor nodes.
2. A node generates TC messages, pretending to be another node.
3. A node generates HELLO messages, advertising non-neighbor nodes.
4. A node generates HELLO messages, pretending to be another node.
5. A node forwards altered control messages.
6. A node does not forward messages as required by OLSR.
7. A node forwards broadcast control messages unaltered, but does not forward unicast data traffic.
8. A node “replays” previously recorded control traffic from another node.

Authentication of the originator node for control messages (for situation 2, 4 and 5) and on the individual links announced in the control messages (for situation 1 and 3) may be used as a countermeasure. However, to prevent nodes from repeating old (and correctly authenticated) information temporal information is also required, allowing a node to positively identify such delayed messages.

OLSR is highly vulnerable to attacks directed at availability. Such attacks are referred to as a Denial of Service (DoS)[43] attacks. An attacker could launch OLSR packets containing false information in large amounts. This could lead to a situation where processing of this data could claim all resources on the receiving nodes, leaving them not able to handle any other tasks. Eventually the OLSR service could crash leaving the node not available. Integrity mechanisms can prevent an untrusted node, not having access to the key used, from performing such an attack.

11.1.3 Related work

Schemes to secure OLSR routing traffic have been proposed in both [17] and [33]. The solution proposed in [17] requires state to be kept for every received packet or signature and uses a rather complex time synchronization scheme. [33] proposes a solution using a single signature for entire OLSR packets, but the signature itself is not carried in an OLSR message, thus breaking RFC3626 compatibility. The latter solution does not prevent replay attacks since no timestamps/sequence-numbers are used.

As this thesis does not focus on security, no deeper analysis of related work, or security as a term, will be done.
11.2 A real life example

As explained in section 11.1.2, OLSR is vulnerable with regards to many types of attacks. In this section a real life example of how a host can compromise the integrity of the IP addressing protocol thereby compromising many of the services built on top of TCP/IP, is presented. As an example it is shown how a normal HTTP session can be “hijacked” by a rouge OLSR node. Of cause, this is only one of many possible attacks.

11.2.1 The scenario

Imagine a host wanting to be able to serve clients in an OLSR routed MANET with a fake web-page when clients request an often used web-server. The attacker needs to consider the following:

1. Become accepted as a router in the MANET routing domain.
2. Get Internet destined traffic routed to the local node.
3. Intercept traffic destined for the address for which to serve the fake web-page.
4. Reply to all requests for the address for which to serve the fake web-page.

The first step is easily accomplished if no security mechanisms are used on the lower layers. An example of such a mechanism would be the WEP protocol. Due to this protocols many weaknesses, the attacker could break this security-layer by using tools freely available[7]. When the attacker is able to communicate with the nodes in the MANET on the IP layer, she can participate in the OLSR routing by starting her own OLSR daemon.

Step 2 requires some more effort if all Internet traffic is to be routed to the attackers host. This could probably be accomplished to a high degree by emitting HELLO and TC messages declaring all nodes heard of in the MANET as symmetric neighbors, while emitting HNA messages declaring Internet connectivity. Another approach could be to try to bring all other Internet gateways down by DoS attacks. But by emitting HNA messages, the rouge node will get all Internet traffic from at least a subset of the MANET routed to itself. If the attackers node actually has Internet access, it is probable that no MANET nodes would be alarmed since commonly used services, such as DNS, should work normally.

The third step can be accomplished by many more or less sophisticated techniques, but the most trivial approach is for the attacker to assign the address of the destination it wishes to intercept to itself. The attacker still uses another address for OLSR routing, but as hosts can be configured with multiple IP addresses this poses no problem. Now all traffic to a given Internet address will be routed to the rouge node, and upon reception the node will not forward the traffic, but pass it up the network stack.

To accomplish step 4 the rouge node only has to run the service(s) that should be faked. In our example this will be a HTTP server running at TCP port 80.

The two latter steps could be done in a much more sophisticated manner. As an example one could use IP filtering technology to only intercept traffic destined for a certain TCP port and forward all other traffic to the actual Internet host. Another way could be to intercept all DNS traffic and resolve host names of interest to the nodes IP.

11.2.2 Implementation of the attack

To do a real-life version of the attack described in section 11.2.1, a very simple scenario was set up as illustrated in figure 11.1. The rouge node B announces Internet connectivity so that A will route all its Internet traffic to B. B wants to intercept all traffic to the geek news-site http://www.slashdot.org. B issues a DNS query for www.slashdot.org and receives the IP address 66.35.250.150. Node B running GNU/Linux sets up a virtual interface eth0:1 and assigns it the address 66.35.250.150. Then B starts a local
HTTP-server. When node A now points its web-browser to http://www.slashdot.org its DNS query will result in 66.35.250.150 and a HTTP GET command is sent to that address. The HTTP query will be received by node B, and B will not forward the traffic. B recognizes the TCP connection to be destined for itself, so B will be the actual end host on the TCP session. B then serves a fake web page to A. A screenshot of the result is shown in figure 11.2.

While this particular attack did not do to much harm, but one can imagine an attacker acting as a proxy to much more sensitive services, such as on-line banking. Even though such services usually are protected using Secure Socket Layer(SSL)[16] or Transport Layer Security(TLS)[29], users will often not have the knowledge to detect a phony certificate or the lack of SSL/TLS encryption.

11.3 Secure OLSR

In this section a solution that utilizes signatures to ensure integrity of OLSR control-traffic data is presented. A digest of the packet and a secret shared key is attached to all OLSR packets. Only a node with access to the secret key can produce such a signature.

In the implemented solution, all OLSR control traffic is signed for every hop. This means that one does not have to consider variable fields, such as hop-count and TTL, in messages. It also means that only one signature is needed although several OLSR messages are stacked in one OLSR packet. This hop-by-hop approach does not provide end-to-end signatures, which again means that the digest is not a true signature with respect to the originator, but rather a signature from the forwarder ensuring us that it trusts the source of the message in the previous hop. However, secure OLSR is designed to be as flexible as possible with regards to the encryption and hashing algorithms being used and the entire signature scheme.

The scheme and algorithms fields in the signature message header(figure 11.3) informs the receiver of what signature scheme and what algorithms are being used. In the implemented solution presented here, signatures created using the SHA-1[31] hashing algorithm are utilized to verify entire OLSR packets. A different scheme could include one signature for each OLSR message allowing for end-to-end signing. The signature message would then contain one signature for every OLSR-message in the OLSR-packet. One can also imagine that asynchronous or synchronous encryption could be used to ensure confidentiality by encrypting all data. These schemes could again utilize different algorithms for hashing and encryption, all being defined by the signature message header.

In the implemented solution, a node that does not have access to the shared secret key cannot produce a verifiable digest. Messages with non-verifiable digests are discarded by all receivers running secure OLSR. To prevent replay attacks secure OLSR uses timestamps. To exchange these timestamps, a two way timestamp exchange mechanism is utilized upon initial connection between two nodes. Signatures are transmitted in OLSR messages of their own. This is to ensure compatibility with nodes not running secure OLSR. Four different messages are defined. One which is the actual signature message as displayed in figure 11.3 and three messages used in timestamp exchange illustrated in figures 11.4, 11.5 and 11.6.
Figure 11.2: What node B in the scenario from figure 11.1 sees when requesting http://www.slashdot.org in its local web browser. This is obviously a forged version of the real slashdot page. Note the hostname in the address bar.

Figure 11.3: The basic signature message as used in the implementation. This is sent as the message body of an olsr message.

Figure 11.4: The timestamp exchange challenge message. This is sent as the message body of an olsr message.

Figure 11.5: The timestamp exchange challenge-response message. This is sent as the message body of an olsr message.

Figure 11.6: The timestamp exchange response-response message. This is sent as the message body of an olsr message.

85
11.3.1 Signatures

The signature-message, illustrated in figure 11.3, is attached to all outgoing OLSR packets. This message is to be the last message in the packet. Since the signature message is piggybacked to outgoing packets after the OLSR header for these packets is created, the OLSR packet header is adjusted to include the size of the signature message in the size field. The digest used as a signature is a hash created using the SHA-1 hashing algorithm which produces an irreversible 160-bit digest. The hash is based on the following:

- The OLSR packet header (with adjusted size).
- All OLSR messages in the packet except the signature message.
- The OLSR message header, the sub-header and the timestamp of the signature message.
- The shared secret key.

No considerations has to be taken regarding the variable fields of different headers (TTL, hop-count) as this signing is done hop-by-hop.

11.3.2 Key management

The implemented solution uses a shared key for signature creation and verification. This could however be done using individual keys per host, and possible encryption could be done using a public-key encryption scheme.

Whenever dealing with keys, one needs a way to distribute them. Key exchange techniques such as Diffie Hellmann is often used. But as modularity is one of the highest ranking design principles in this work, the secure OLSR protocol does not intend to cover key exchange/management or initial authentication. It is assumed that a shared secret key is made available to all hosts intended to be part of the MANET by some external means. This could be a key distribution service or a user actually typing the key on her local node.

11.3.3 Timestamps and freshness

In a situation like depicted in figure 11.7, an advisory records signed traffic and plays it back at a later stage. This can be prevented to some degree by sequence numbers which are already utilized in OLSR, but for traffic that is only to be sent one hop, like HELLO messages, this is of little or no help. An evildoer can simply record all messages transmitted by a node and move to another area of the network where the HELLO messages recorded was never heard. Here the evildoer can start a replay attack by transmitting the recorded messages. The OLSR sequence numbers are also weak because of their length. They are only 16-bit values and wrap-around will occur rather frequent. The wrap-around mechanism used in OLSR makes the sequence numbers even weaker with respect to freshness.

11.3.4 Timestamp exchange

In the solution proposed in this chapter, timestamps are used to determine freshness. This technique requires an exchange of timestamps between nodes. The timestamp exchange process introduces three new message types. These messages are processed regardless of the signature message validation. This process is likely to take place between neighbors that have no registered timestamp of each other, and therefore traffic between them will not be validated by the signature check. Because of this, all such messages are signed internally. This means that all the timestamp exchange messages carries their own digest and they are never stacked with other OLSR-messages but rather sent in OLSR-packets of their own.

The exchange of timestamps between two neighbor hosts A and B can be described as:
Figure 11.7: A replay attack on OLSR. The attacker records signed messages and plays them back on a later stage or to nodes that has no record of the sequence numbers of some of the recorded messages.

\[ A \rightarrow B : Ch_aD(M, K) \]
\[ B \rightarrow A : Ch_bT_sD(IP_b, Ch_a, K)D(M, K) \]
\[ A \rightarrow B : Ts_aD(IP_a, Ch_b)D(M, K) \]

When \( A \) receives a signed message from a neighbor \( B \), for which \( A \) has no registered time value, \( A \) initiates the timestamp exchange process. \( A \) first sends a challenge message (figure 11.4) to \( B \). This message is broadcasted since \( A \) might not have an actual route to \( B \). The challenge message contains a 32-bit nonce\(^1\) value, \( Ch_a \). \( A \) then signs this message with a digest of the entire message and the shared key \( D(M, K) \).

\( B \) now has to respond to this message with a challenge-response (figure 11.5) message. \( B \) first generates the digest of its IP address (if \( B \) is multi-homed the IP address fetched from the challenge message is used), the received nonce and the shared key \( D(IP_b, Ch_a, K) \). \( B \) then generates a 32-bit nonce, \( Ch_b \), and transmits the nonce, the timestamp of \( B \), the digest \( D(IP_b, Ch_a, K) \) and a digest of the entire message and the shared key \( D(M, K) \).

When \( A \) receives the challenge-response message from \( B \), it first tries to validate the data. If the digests \( D(IP_b, Ch_a, K) \) and \( D(M, K) \) can be verified, then the timestamp of \( B \) is used to create the difference of time between \( A \) and \( B \). \( A \) then generates a response-response message (figure 11.6) and broadcasts it to \( B \). This message contains As timestamp, a digest of \( A \)’s address (as received from \( B \)), the nonce received from \( B \), and the shared key \( D(IP_a, Ch_b, K) \) and a digest of the entire message and the key \( D(M, K) \).

Note that in addition to the data described here, the IP address of the destination is always sent in timestamp exchange packages.

When \( B \) receives the response-response message from \( A \), it tries to verify the digests. If they can be verified, \( B \) uses the received timestamp to register its time difference to \( A \). The timestamp exchange is then complete.

11.3.5 Working on timestamps

The solution does not require synchronized time, but the clocks are assumed to be relatively synchronized meaning that they are running on a relatively equal frequency.

All timestamps are represented with a 32 bit value containing seconds since the epoch\(^2\). Timestamps are

\(^1\) Number used Once. A random number. It is used to append random data to real data to prevent replay attacks.

\(^2\) The time and date corresponding to 0 in an operating systems clock and timestamp values. Under all Unix versions the epoch is 00:00:00 GMT, January 1, 1970
at first recorded as $T = T_L - T_R$ where $T_L$ is the local timestamp and $T_R$ is the remote timestamp received through the timestamp exchange. When receiving a signature message, a certain slack $S$ in the calculated timestamp difference is allowed. This means that a signature message with a verified digest and a timestamp difference $T_N$ so that $(T_O - S) < T_N < (T_O + S)$ where $T_O$ is the stored timestamp difference of the sender, is considered a verified signature message.

To compensate for a possible skew between clocks, the timestamp difference is recalculated for every received and verified signature message. The difference is recalculated as $(T_O + T_N)/2$ where $T_O$ is the recorded timestamp difference and $T_N$ is the difference calculated based on the received timestamp.

### 11.3.6 Robustness

The timestamp exchange process could be exploited by an adversary to create an overload of processing and network usage. This could lead to the attacked node not being able to participate in other timestamp exchanges or perhaps any communication at all. This would be a typical DoS attack.

An evildoer, or just a misconfigured host, could transmit thousands of the timestamp exchange challenge messages within a very short period of time, all aimed at the same host. This would cause the receiving host to generate and transmit signed replies to all the challenges. To avoid this, a timer is set for the originators of all received challenges. Any new received challenges from the same host while the timer has not timed out, are discarded. Due to the signing of the challenge messages, an attacker cannot spoof the sender address of challenge messages. An attacker could however, record all challenge messages directed to a host for a long period of time and launch them all within a short period of time. But as timestamp entries are cached within nodes, the timestamp exchange process will not be initiated very frequently. Therefore, this amount of messages would not be extensive.

### 11.4 Implementation

The secure OLSR proposal is implemented as an olsrd plugin. The implementation includes message signing and timestamp exchange, and it is part of the olsrd source code available for download at [http://www.olsr.org](http://www.olsr.org). Implementing functionality that was to work on all incoming and outgoing “raw” OLSR traffic, required an extension to the network output functionality in olsrd. Apart from this, the plugin could be implemented without problems due to the modular design of olsrd. An overview of the relations between the plugin and olsrd is illustrated in figure 11.8. The implementation is to be as transparent to the olsrd code as possible.

Therefore, all incoming traffic is passed to the plugin which verifies the packet and removes the signature message and updates the size field of the OLSR packet header. For outgoing traffic the opposite goes. All outgoing OLSR traffic is passed to the plugin which adds the signature and updates the packet size.

The key used is read from the file `/root/.olsrd/olsrd-key` and is 128-bits of size. If no key can be read from the file, the plugin will terminate the olsrd process with a warning message.

Other solutions might be implemented in future versions to handle local key management better or to be able to work in an integrated fashion with some authentication scheme.

#### 11.4.1 Intercepting incoming traffic

The secure OLSR plugin must be able to intercept all incoming OLSR traffic and check the signature if present. This is a matter of de-registering all the OLSR sockets (UDP port 698) from the socket listener, and then re-register them with the plugins own input function. This is done using the functions:

```c
int remove_olsr_socket(int, void(*)(int));

void
```
Figure 11.8: A illustration of the design of the secure plugin as related to olsrd.

```c
add_olsr_socket(int, void(*)(int));
```

which are implemented in `src/socket_parser.c`. The OLSR sockets can all be retrieved from the global interface list `ifnet`.

The plugin's own OLSR input function keeps the registered message parser functions and only differs from the original input function in olsrd on two points:

- An incoming packet is checked for timestamp exchange messages which are processed before the signature check. Keep in mind that these packets contain signatures of their own.
- An incoming packet is checked for an ending signature message:
  - If no such message is found, the packet is not considered sane and is discarded.
  - If a signature message is received from a neighbor for which no timestamp is registered, the timestamp exchange process is initialized.
  - If the neighbor is registered, the signature is checked.
  - If the signature cannot be verified, the packet is discarded.
  - If the signature is verified, the timestamp is checked.
  - If the timestamp validates the packet is passed on to the packet parser within olsrd.

### 11.4.2 Intercepting outgoing traffic

The plugin also needs to be able to intercept all outgoing traffic to add signature messages. To be able to do this, a new set of function pointers was added to olsrd. They are called `packet transformation` functions. A plugin can register its own packet transform functions with `olsrd`, and these functions are applied to every OLSR packet right before sending it. The function to add such a function pointer is declared in `src/net.h` as:

```c
```
The plugin registers a function that calculates and adds a signature to the end of all outgoing OLSR packets. To be sure the packet will have room for the signature message (especially when stacking messages) the max message size in olsrd is set to `maxmsgsize - sizeof(signaturemsg)`.

### 11.4.3 Timestamp exchange

The plugin maintains a repository of registered timestamps. Whenever a node receives a packet containing a valid signature message the timestamp repository is searched for a matching entry. If no timestamp entry is registered for the neighbor, the timestamp exchange process is initiated. The timestamp exchange is carried out as explained in 11.3.4. All timestamp messages are broadcasted within regular OLSR packets. These messages use the generic OLSR message header.

Since an initiator of a timestamp exchange has no timestamp entry registered for the peer node, the timestamp exchange messages must be processed even though regular packets from this host are discarded. This is achieved by checking for timestamp exchange messages prior to the signature verification test. This way timestamp exchange messages are processed even if the OLSR packet is discarded. This is still secure due to the usage of nonces and signatures in the timestamp exchange messages.

### 11.5 Future work

The security solution presented in this chapter is designed to be flexible. It operates independent of hashing and cryptographic algorithms, and it can use different schemes.

Implementing support for end-to-end signatures and possibility for public-key cryptography using individual key-pairs are things that could be implemented as a next step. Also, it should be considered moving the signature message to the front of the OLSR package since this message can be of variable size if using multiple schemes. If the message is put first one can avoid having to traverse all messages in the package to find the signature message. Adding IPv6 support would also be a natural next step for the implementation.

Combining the solution with some authentication and key management scheme would provide a more complete solution.
Chapter 12

Self-configuring networks

“Adapt or perish, now as ever, is nature’s inexorable imperative.”

– H. G. Wells (1866 - 1946)

Mobile ad-hoc networks should be self-configuring. This means that nodes who are to participate in a MANET should not need extensive knowledge of network parameters prior to joining the network. This should include automatic configuration of IP addresses. Although IP address auto-configuration is not explicitly mentioned in RFC2501, many ad-hoc networks would benefit from some generic auto configuration scheme. Not only to provide automatic IP address allocation, but also to detect more basic abilities of the MANET such as the routing protocol utilized.

In this chapter a proposed IP auto-configuration protocol for use in MANETs using a pro-active routing protocol, is presented. An implementation of the protocol has been made for olsrd. The protocol itself is to be released as an Internet draft.

12.1 Background

Any node in an IP network needs a valid IP address to be able to participate in routing and regular data communication. It is the IP address that uniquely identifies the node as an endpoint in network communication.

There are several ways to acquire an IP address. For wired networks, the two most common ways are either static configuration, or by the use of the Dynamic Host Configuration Protocol (DHCP)[30]. A static configuration can simply be that the node has been given a valid address by a system administrator and will always use this address. For a network with many nodes, this is not scalable. The DHCP framework is based upon a client-server model. This architecture leads to a single point of failure. If the DHCP server is out of reach the service is not available. DHCP clients also uses broadcasts to emit requests. For a MANET, both these factors are troublesome. MANETs should favor distributed operation as opposed to centralized designs as those used in DHCP. Broadcasting in a MANET will not reach all nodes in the network unless special broadcast extensions or other mechanisms are used. So DHCP is not a well suited solution for IP address auto configuration in MANETS.

12.2 Duplicate address detection

The work [49] is the most recent work in the subject of auto-configuration in MANETs. This work describes a mechanism for Duplicate Address Detection (DAD), but does not specify how to acquire an IP address in the first place. In [49] DAD is divided into strong $DAD$ and weak $DAD$. 
When a new node enters a MANET, it uses Strong DAD to check if its chosen address is already in use. This means that some sort of request message must be flooded throughout the MANET. If a node is already using, or possibly know of a node using, the requested IP address, then a response message is sent back to the originator of the request. This response message states that there is an address conflict.

Weak DADs purpose is to detect address duplication during MANET routing. Some mechanism must be present to dynamically detect the existence of duplicate IP addresses used in the network. This mechanism could also be used to detect merging of networks. A network merge detection should cause some special action to be taken as there might be lots of address duplicates.

### 12.3 Pro-Active Auto-config

Pro-Active Auto-config (PAA) is a proposed IP address allocation protocol including strong DAD, for use in MANETs. The protocol can be used with both IPv4 and IPv6. PAA takes advantage of the fact that nodes that are already members of a MANET domain, running a pro-active routing protocol, already has a relatively complete understanding of the topology. Such nodes are therefore well suited to allocate an IP address that is probably currently not used by any nodes in the network. PAA also does strong DAD to ensure that no other node is currently configured, or in the process of configuring itself, with a duplicate address. This is done by flooding the network using the underlying routing protocol.

PAA provides a node with the ability to automatically configure itself with an unique address and join a MANET. This configuration takes place without the node having any prior knowledge of the network parameters.

#### 12.3.1 Basic operation

As outlined in figure 12.1, PAA consists of three software components. An unconfigured node runs the PAA-client to allocate an unused IP address. The PAA-client communicates with one (or more) PAA-servers running on already configured nodes. These servers communicate with the routing daemons running on their local hosts. All communication between PAA-clients and PAA-servers is done using the link-local address space `192.168.0.0/16` when using IPv4 and `fe80::` when using IPv6. DAD flooding is done using the flooding mechanism offered by the routing protocol in addition to link-local traffic.

PAA packets uses the 64-bit header illustrated in figure 12.2, and all traffic is carried using UDP.
Proxy vs. broadcast solution

Two main approaches can be taken when requesting an IP address from nodes in a MANET. They are here referred to as the proxy solution and the broadcast solution.

If using the proxy solution, the unconfigured node elects one configured node that operates on behalf of it for the entire configuration session. In PAA this can be done, but the implementation on which this chapter is based, uses another approach. The main idea in the implementation of PAA, is that an unconfigured node is to be able be mobile while doing configuration. Because of this no proxy is used for configuration.

A node communicates with any configured nodes that are in communication range. Data is broadcasted from the unconfigured node and back from configured nodes. Because of this, this approach is called the broadcast solution. No state is kept in the configured nodes regarding configuration sessions. An unconfigured node can be mobile and use different configured nodes for different parts of the configuration process.

Both approaches have their pros and cons. The broadcast solution gives the advantage of mobility, but the proxy approach could provide a more robust DAD. If using a proxy, address conflict messages in strong DAD, can be unicasted back to the proxy node. When using the broadcast solution all such traffic is flooded throughout the MANET. Some specific scenarios where a node loses connectivity and does not receive conflicting messages could also be avoided when using the proxy solution, but new such problematic scenarios could be generated for the proxy solution as well.

An example of a configuration session

Figure 12.8 illustrates a configuration session where a new node joins the MANET. In this example, no address conflicts are detected. If using IPv4 the unconfigured host configures itself with a random link-local address, if using IPv6 the link-local address automatically assigned to the used interface is used. The PAA-client then broadcasts (IPv4) or multicasts (IPv6) an Address Request (figure 12.4). Since the IPv4 link-local address is generated using random numbers, conflicts can occur. Because of this, an identifier that the PAA-client generates based on the interfaces MAC address, is used for host identification. This identifier is also used when using IPv6, since all return traffic is also multicasted back to the client from the server.

The Address Request message might contain a preferred address that the unconfigured host wishes to use.
if available. When using IPv4 this is a regular IP address, when using IPv6 this is the MAC address of the interface on which PAA is running. Note that an IPv6 Address Request still contains a 128 bits preferred address field to allow for possible future changes and to maintain a consistent IPv4/IPv6 packet design. A PAA-server receiving this request forwards it to the PAA-plugin. When using IPv4 the PAA-plugin checks if the preferred IP address is available, if the address is not available or if no preferred address was received, it acquires a random available address. An available address is defined as an IP address that the configured node has not heard mentioned in any OLSR control traffic for a given time. If using IPv6, the client has to submit its MAC address in the request message. The server then forwards the request to the PAA-plugin that generates an address based on the MAC. The address is made up of a prefix from some predefined net-range and the MAC address. This address is then checked against the IP cache pool. If a conflict is discovered multiple "backup" net-ranges could be used to create an address based on the MAC. The allocated address is then sent back to the requesting host in an Address response message (figure 12.5).

The requester is now to perform strong DAD. This is done by broadcasting or multicasting a Address Test message (figure 12.6) link-local. All PAA-servers receiving this message is to flood it through the MANET using MPR flooding. This is done by encapsulating the message in an OLSR message and sending it as a regular OLSR message. The PAA-server must confirm that it has received the message from the requester by replying with a Test Confirm message. If a requester does not receive such a confirmation message, the emitted Address Test is not considered flooded and another Address Test message is broadcasted or multicasted immediately.

An OLSR daemon extended with a PAA-plugin that receives an Address Test message, is to forward this link-local if the node has received a Forward Request (figure 12.3) within a given time interval. Any node in the process of configuration, broadcasts or multicasts such forward request messages regularly. This is to make sure configured nodes within transmission range of any unconfigured host will forward all received Address Test messages link-local.

If a node in the process of configuration, receives an Address Test originating from another unconfigured node and this message contains the address the local node has been offered, it is considered an address conflict. This is because some other node (the sender of the Address Test message) is in the process of configuring itself with the same address the local node was offered. The node will then restart the configuration process by sending a new Address Request.
12.4 Implementation

The Pro-Active Auto-configuration solution is implemented for use with olsrd. Parts of the protocol is currently patent pending [14] by Thales Communications AS, and therefore the source code is not publicly available as of yet.

In the following sections we will look into the PAA server, PAA client and the olsrd plugin in detail.

12.4.1 PAA client

When an unconfigured node starts the PAA service, it is started in client mode. A flow diagram describing the IPv4 PAA-client operation is illustrated in figure 12.9. The first thing the IPv4 client does, is to generate a random link-local address. It will then configure an “alias” interface with this address. Such alias interfaces is the way one can configure an interface with multiple IPv4 addresses on GNU/Linux systems. If PAA is set to run on eth0, the alias interface will be eth0:0. This interface will be used by both the PAA-client and later the PAA-server. Due to the binding of sockets to devices as explained in section 6.2.3, olsrd cannot run on alias interfaces. Therefore the PAA-client/server uses an alias interface leaving the “real” interface to olsrd. The IPv6 client uses the link-local address automatically assigned to the interface PAA uses. IPv6 allows for several IP addresses to be set for the same interface, so there is no need to set up an alias interface.

The PAA-client must periodically broadcast or multicast a Forward Request message link-local to let already configured neighbors know that they should forward all received PAA control messages link-local. In the implementation this message generation is done in a thread of its own emitting a Forward Request every 2 seconds.

The client must generate an identifier that it will use to identify itself in PAA traffic since link-local address conflicts might occur. In the implementation, this ID is the lower 32-bits of the MAC address of the interface on which PAA runs. This diminishes the uniqueness of the ID, but the chance of two WLAN interfaces using the same last 32-bits in their MAC addresses is relatively small. The first six bytes of the MAC address are the “Organizational Unique Identifier”, while the lower six bytes are the actual factory serial number. One can however still risk an ID crash if two interfaces by different makes that share the last 4 bytes in their Organizational ID, with the same serial number were to meet. But in general, there is a much smaller chance for this happening than the upper 32-bits matching, which in many cases only would require having two interfaces bought from the same stock.

To receive a free IP address, an Address Request message is broadcasted or multicasted link-local. A node signs this request with its ID. Any neighbor that is already a configured member of the MANET routing domain and out of quarantine time, as explained later, will answer with an offered address if an address could be allocated. If no replies are received, the PAA-client can optionally configure its interface with a random address within a pre-defined address space and start the routing daemon, thus starting its own MANET.

Upon receiving the first Address Response carrying the correct ID, the requester will generate an Address Test message and broadcast or multicast it link-local to all neighbors. Any other Address Response message received for the same request is silently discarded. Any configured node that receives the Address Test message, sends an Address Test Response confirmation message so that the PAA-client can be sure that the test-message is flooded. If no Address Test Response is received by the PAA-client, a new Address Test is sent. This is repeated until at least one Address Test Response is received.

The PAA-client then waits for a given interval to receive a possibly conflicting Address Test message sent by other nodes. If this interval times out without any conflicting Address Test messages received, another Address Test message is broadcasted or multicasted. If another time interval passes without the node receiving any conflicting Address Test messages the address is considered valid and DAD is considered complete. If a conflicting message is received, the configuration process is restarted.

When a unique IP is allocated the PAA-client will configure the interface to use with the address and Forward Request messages will no longer be sent. The thread generating these is terminated. When the
Generate link-local address
 Configure "alias" interface with link-local address
 Generate ID

Broadcast Address Request
 wait for reply for 5 seconds

1–3 times

No valid reply received

Valid reply received

Retry 3 times

No response for third request

Start a new MANET?

Yes

No

Terminate

Generate random address within a predefined subnet
 configure interface start olsrd

Conflicting Address Test received

No address conflicts

Configure interface with offered address
 Start olsrd
 Go to PAA-server mode

Broadcast Address Test
 Wait 10 seconds
 Repeat once

Figure 12.9: The flow of the IPv4 PAA-client.
interface is configured, olsrd is started. The PAA-client must take care to ensure that the routing daemon will run on the correct interface. PAA is also responsible for terminating the routing-protocol process when it is terminated itself. When the routing-daemon is started, the PAA-client will put itself in server mode.

### 12.4.2 PAA server

The basic operations of the PAA-server is illustrated in figure 12.10.

The first thing PAA does when going from client- to server-mode is to connect to the routing-daemon via a TCP socket to the loopback device (127.0.0.1 or ::1).

Upon receiving an Address Request from a PAA-client the PAA-server queries the routing protocol for a free address. In the implementation, this means querying the PAA-plugin via IPC. If the Address Request contains a preferred address, this is passed to the OLSR daemon in the query message. When using IPv6 the MAC address of the sender is always passed in the message. If no free IP addresses can be allocated, the PAA-server will transmit an Address Response message with the flag set. If an IP offer is received from the routing protocol, an Address Response message is generated containing the offered IP, the ID and SEQNO from the received Address Request.

Upon receiving an Address Test message link-local, the PAA-server forwards this message to the routing protocol if the flag is not set in the message. This is to prevent forwarding of messages forwarded link-local from other PAA-servers, which can lead to loops. The PAA-server then sends an Address Test Response message back to the PAA-client to confirm that the message has been received and is currently being processed.

A PAA-server should only forward messages link-local if there exists any unconfigured 1 hop neighbors. To detect this, the PAA-server listens for Forward Request messages which are sent periodically by unconfigured nodes. Upon receiving a Forward Request the forward timer of the server is set to current time + a predefined holding time. Upon receiving an Address Test message from the routing protocol, the PAA-server broadcasts or multicasts this message link-local, setting the flag, if the forward timer is not expired.
12.4.3 PAA OLSR-plugin

The role of the PAA-plugin is illustrated in figure 12.11.

As mentioned earlier, the PAA design is based on the fact that in a proactive routing protocol such as OLSR, nodes will eventually have heard of close to all IP addresses currently used in the MANET. This can be used as a way of generating an IP address that has a much larger possibility of being unused than a random address. But rather than checking for free IP addresses by looking up all internal tables of the routing daemon, an IP cache “pool” of used addresses is maintained. This way IP addresses can also be cached for a longer periods than they would stay in the routing daemon’s internal tables. Upon receiving any kind of known routing control traffic, the plugin adds all the IP addresses listed in the message to the IP cache with a given timeout. This period is set to 30 seconds in the implementation. This also goes for PAA traffic. The address contained in Address Test messages is updated in the IP cache as well as addresses offered by this node as response to Address Request messages from the PAA-server.

The PAA-plugin needs to be able to hear all incoming OLSR traffic to update its IP cache. To do this, a function is registered with the message parser using the special PROMISCUOUS message type declared in src/praser.h. When registering a message parsing function with this type, all incoming packets are sent to the function. Bi-directional IPC also has to be maintained between the plugin and the PAA-server. This is done by registering the IPC socket and a parsing function with the olsrd socket parser.

To locate a free address, the routing daemon selects a random address in the net-range that the MANET uses or uses a possible preferred address provided by the PAA-server. When using IPv6, a MAC address must be provided in an Address Request, and this address is used to generate an IPv6 address. The address is then checked against the IP cache. If the address is already registered in the cache, a random IPv4 address is created within the MANETs net-range or an IPv6 address is generated for another net-range. This address is then checked against the cache. This process is repeated until an assumed free address is located or the process has been executed for a predefined maximum of times. If a generated address is not found in the cache, it is considered free and will be sent to the PAA-server that will offer it to the remote PAA-client, which again will perform DAD on the address.

Upon receiving an Address Test message from the PAA-server, the routing daemon encapsulates the Address Test message in a routing protocol message. This message is then flooded throughout the MANET by the flooding mechanism provided by the routing protocol. In the implementation, this means that the
Address Test message is encapsulated in a regular OLSR message packet (figure 3.1), and flooded using the MPR scheme as described in section 3.4. Upon receiving an encapsulated Address Test message carried by the routing protocol, the routing daemon decapsulates the packet and forwards it to the PAA-server. This is done in addition to forwarding the encapsulated message according to the MPR scheme.

12.5 Future work

Several updates could be done to make a more robust solution. One could, for example, consider implementing the proxy solution. Smaller updates like extending the ID field in PAA control traffic to contain the entire MAC + some random value would also increase robustness.

PAA is not in any way responsible for authentication of nodes. To have some sort of access control, one must apply an outer layer of security mechanisms. PAA could be used with a scheme like the signature solution proposed in chapter 11 combined with an authentication solution. Some distributed authentication system can be imagined, but physical distribution of keys using e.g. smartcards can be sufficient in many scenarios.

In a series of events as depicted in figure 12.12, two MANETs split and later merge. The MANETs have both configured new nodes while existing apart. When the networks merge again, an address conflict arises between already configured nodes. These conflicts should be detected by weak DAD. PAA does not perform weak DAD, and keeping a modular design in mind, this is not the responsibility of the IP configuration functionality. A scenario where PAA is used should also have mechanisms to detect address conflicts among already configured nodes.
Chapter 13

Gateway tunneling

“They have the Internet on computers, now??”
–Homer Simpson

A MANET has no fixed infrastructure, and services on the Internet might not be available in such networks. However, it is likely that nodes in an ad-hoc network in many cases want to connect to nodes on some external network, using services available there. For a widespread and successful deployment of MANETs, the ability to provide easy access to the Internet is therefore a prerequisite.

A common approach is to let a MANET node with Internet access operate as an Internet gateway and provide Internet access to other nodes in the MANET. There can naturally be several MANET nodes operating as gateways on the MANET at the same time. In OLSR, HNA messages, as explained in section 4.1, are used to announce the gateway service.

13.0.1 Routing Internet traffic

When an OLSR node communicates with an external host the destined external host will normally send return traffic to the source IP address of the outgoing packet. Thus, for IPv6, a MANET node configures an address under a global prefix managed by one of the gateways and uses this address as source IP address when communicating with external hosts on the Internet. Return traffic from the external nodes on the Internet is therefore routed back to the gateway, which in turn can forward the packets to the MANET node. However, for IPv4 there is a great scarcity of global IPv4 addresses. Thus, the gateway may be equipped with a very limited number of external IPv4 addresses. To allow different MANET nodes to share an address for external communication, the gateway may implement a Network Address Translator (NAT)[32].

All routing within an OLSR routed MANET is host based. This means that there exists one entry in the routing table for every host to which the local node has calculated a route. When using HNA gateways the routing table is aggregated. This means that all traffic to a defined network is sent to a certain gateway. For an OLSR routed MANET this means that IP packets that do not have an IP destination address known locally on the MANET are forwarded along a possible default route out of the MANET through the default gateway.

In addition to using default routes for outgoing packets, a mechanism is required to ensure that return traffic from the Internet gets routed back to the node in the MANET. A gateway that implements NAT will translate the source IP address of outgoing packets from the MANET node. It replaces the source IP address with an address of the NAT gateway which is route-able on the external network. Hence, an external host will return packets using the IP address of the NAT-gateway as destination IP address. The gateway can then replace the destination IP address with the IP address of the MANET node, and inject the return traffic into the MANET.
13.1 HNA problems

OLSR sets up hop-by-hop routes. This means that while calculating the complete route locally, olsrd will just enter the next hop on the path to the destination into the routing table. Therefore OLSR routing is depending on the distributed operation of the protocol since the sender has no control of where the next hop router routes the traffic. This also goes for HNA routes. If A has a static link to IN1 and no link to IN2 in the scenario depicted in figure 13.1, it will use GW1 as its Internet gateway. The route added to the kernel routing table will be:

```
Network: 0.0.0.0
Gateway: IN1
Metric: 2
```

IN1 will have a route entry:

```
Network: 0.0.0.0
Gateway: GW1
Metric: 1
```

If A, for some reason, wishes to communicate through GW2, it simply cannot. A cannot add GW2 as the gateway since a gateway is to be the next hop along the path. Because of this a route to GW2 would be:

```
Network: 0.0.0.0
Gateway: IN1
Metric: 3
```

But when this traffic arrives at IN1, it will be routed via GW1. This fact causes problems in several ways.
13.1.1 HNA and heterogeneous nodes

In section 12.7, RFC3626 states that all nodes in a MANET does not need to implement HNA support for external access to work in a distributed manner. This is not true. Because of the hop by hop routing explained in the previous section, all nodes need to implement HNA functionality. In the scenario depicted in figure 13.1, if IN1 does not implement HNA support it will still forward HNA messages from GW1 to A due to the default forwarding algorithm. A will therefore still add a default route setting IN1 as the next-hop. All Internet-destined traffic from A will then be sent to IN1. However, since IN1 did not process the HNA info from GW1, it has not set up a default route. IN1 will therefore drop all Internet traffic received from A.

13.1.2 Network Address Translation

In the scenario depicted in figure 13.1 node A is alternating between having a bidirectional connection to IN1 and IN2. This is a very simplified scenario but it demonstrates the basic problem, that A cannot decide which gateway to use. If one of the gateways (or both) implement NAT, TCP connections routed through the gateways, will break every time A moves from IN1 to IN2, and vice versa.

In a testbed set up as displayed in figure 13.1, it is easy to show the problem. The graph in figure 13.3 illustrates a TCP connection which breaks, as opposed to figure 13.2 where the gateways do not implement NAT. In both tests, A switches connectivity every 20 seconds. In figure 13.2 one can observe that it takes some time for the Internet route to be updated.

These tests were done using an earlier version of olsrd. Updates in the 2-hop neighbor sensing and the addition of link hysteresis should decrease the gaps between connections.

13.2 Gateway tunneling

To solve both the problems concerning NAT and heterogeneous nodes, one can utilize IP-in-IP tunneling. A tunnel is set up between the sending node and its chosen gateway as illustrated in figure 13.6. An IP-in-IP tunnel encapsulates IP packets by adding an extra outer IP header, to them. This means that the original packet, including header, is the payload of the new packet. This extra header is added by the transmitting endpoint of the tunnel. The packet is then routed to the receiving endpoint of the tunnel which decapsualtes
When using IP-in-IP tunneling, no external addresses are used when routing traffic destined for such external networks. This way nodes which implement HNA functionality can co-exist with nodes not implementing this functionality. This solves the heterogeneous nodes problem described in the previous section. Also, when using this tunneling scheme, a node is always in control of what gateway it uses. This way the NAT problems described in the previous section, is also solved.

In the gateway tunneling solution proposed here, a unidirectional tunnel is set up from the transmitting node within the MANET, to the gateway. All traffic destined for the Internet is routed through this tunnel. Returning traffic is routed normally back to the node from the gateway. This way nodes that act as gateways need only to be able to receive incoming encapsulated packets, and there is no need for them to set up reverse tunnels. This means that no negotiation or initialization between the transmitting node and the gateway is needed.

### 13.3 Implementation

An implementation of the gateway tunneling proposal has been made for olsrd. As of yet the code is very experimental and mostly written to be able to perform the testing needed for this chapter and the paper included in appendix G. In future versions of olsrd, the tunneling code will be made more flexible and it will possibly be moved out to a plugin. The source code is located in src/linux/tunnel.c in the olsrd code available for download at [http://www.olsr.org](http://www.olsr.org).

For the implementation to work, the Linux kernel on the system must support IP-in-IP tunnels. A easy way to verify this is to check for the tun10 interface by issuing the command `ifconfig tun10` in a shell.

To configure and activate tunnel interfaces the `SIOCCTLTUNNEL` and `SIOCADDTUNNEL` ioctls are used. The source code of the `ifconfig(8)` network interface configuration tool from the GNU net-tools package was consulted for these tasks.

### 13.3.1 Basic operation

For olsrd to use gateway tunneling it must be started with the `-tun1` command-line switch. Note that the configuration is very static, the tunnels are only set up for Internet routes and the first heard of Internet gateway is used throughout the session. As of yet the implementation is mostly created to be a proof of concept and it is not ready for end users.
Here follows a brief description of the functionality of the implementation.

**Gateway mode**

If the local node is to announce itself as a gateway to Internet(0.0.0.0/0), the node will enter the *gateway mode*. This means that the node will be setting up a point-to-multipoint tunnel endpoint at the tun10 interface. This interface will accept incoming IP-in-IP tunnel packets from all hosts.

To enable forwarding from this interface to the interface with the Internet connection, the proc entry /proc/sys/net/ipv4/conf/tun10/forwarding must be set to 1. After a lot of testing and arbitrary proc-file alteration, it turned out the /proc/sys/net/ipv4/conf/tun10/rp_filter proc entry needed to be set to 0 to prevent the incoming packets from being discarded. As mentioned in section 6.2.2 this entry controls *address spoof filtering* on the given interface. Address spoof filtering is filtering of packets with suspicious sender address based on the receiving interface.

**Client mode**

If the local node is not set up to announce itself as an Internet gateway, it will enter the *client mode*. The first incoming valid HNA message announcing Internet connectivity, will trigger the initialization of an IP-in-IP tunnel setup. The client sets up a point-to-point tunnel at interface tun11 (as the tun10 interface will not, for unknown reasons, allow this operation) from the local node to the announced gateway address. A default route is then set up through this interface causing all traffic destined for the Internet to be routed through the tunnel.

**13.3.2 Tests**

As seen in figure 13.3, a TCP session breaks as soon as a change in gateways occurs in the scenario from figure 13.1. Figure 13.4 illustrates how traffic is received from the different NAT gateways when node A alternates every 20 seconds between connectivity with IN1 and IN2. To be able to show this, unidirectional UDP traffic was sent from A to an Internet host. This traffic was recorded at the Internet host.

One reason for using gateway tunneling is to avoid TCP session breakage if the a gateway is using NAT. Figure 13.5 illustrates TCP traffic sent using the tunnel solution. Node A (in the scenario from figure 13.1) alternates between connection with IN1 and IN2 every 20 seconds just like in the previous tests. This time A sets up a IP-in-IP tunnel to GW1 and all Internet traffic is routed through this tunnel. The x axis of the graph displays TCP throughput. One can observe that the throughput is lowered with approximately one third every time the node goes from being connected to IN1 to IN2. This is due to the fact that IN2 routes the tunneled packets via IN1 to GW1 as illustrated in figure 13.6, while IN1 can route traffic directly to GW1. An extra hop like this causes the maximum throughput to sink due to the retransmission.

**13.4 Future work**

As mentioned earlier, the implemented solution is an *ad-hoc* solution. To have a usable solution in a generic MANET environment, several updates should be done and several extensions could be made.

**Dynamic tunnel setup**

The most obvious thing to update is the selection of the gateway to which a tunnel is set up. One can imagine a solution where a node waits for a predefined amount of time before setting up a tunnel. Within this time interval all HNA gateways should have been registered. The node can then decide what gateway to use based on a constraint like hop-count.
Figure 13.6: A scenario where A has set up a tunnel to GW1 and routes all Internet destined traffic through this tunnel. The leftmost figure shows the traffic path when A has a bidirectional link to IN1. The rightmost figure shows the path of the Internet traffic when A only has a bidirectional link to IN2.

The tunneling solution should also apply for any given subnet. Even though the obvious example is Internet gateways, several gateways could also exist for subnets.

The tunneling could be made dynamic. This would include dynamically check for the existence of TCP connections. If at a given time no such connections exist to nodes routed through the tunnel, a check for more suitable gateways (due to mobility) could be done. If a closer gateway is found the current tunnel is updated to have the closer gateway as the endpoint.

**Inter-gateway tunneling**

Mechanisms for setting up tunneling from gateway to gateway, using the MANET as a transit network could be considered. Such a scenario is illustrated in figure 13.7.

**Load balance**

Tunnels could also be set up to balance traffic load. One node could use different gateways for different kinds of traffic. This could be combined with some Quality of Service mechanism.

**Mobile IP**

The tunneling scheme could also be made interoperable with Mobile IP. To do this olsrd should also support distribution of Foreign Agent Advertisements. This could be implemented as a plugin.
Figure 13.7: Setting up a tunnel to connect two external networks.
Chapter 14

Conclusions

“The wireless music box has no imaginable commercial value. Who would pay for a message sent to nobody in particular?”
- David Sarnoffs associates in response to his urgings for investment in the radio in the 1920s.

The work on this master thesis has been very interesting and many lessons have been learned along the way. In this chapter some of the things learned in the implementation and testing process are described as possible updates to RFC3626. The future of the UniK OLSR daemon and extensions is also discussed, and some final conclusions are drawn.

14.1 Combining the extensions

The project of implementing the OLSR protocol turned out to embrace much more that just OLSR. Many extensions taking advantage of functionality in the protocol have been made, and a flexible plugin interface is defined and implemented to ease the task of implementing and maintaining extensions.

The OLSR daemon implements sufficient functionality to set up and run a MANET. But to have a fully self configuring and secure solution the software used should also implement security solutions, key management, authentication mechanisms, auto-configuration mechanisms and address management. The solutions proposed in 11 and 12, covers security and auto-configuration. Work has been done [46] [68] [64] [60] on the remaining issues.

14.2 Suggestions for RFC3626 updates

Through the work on the OLSR implementation some parts of RFC3626 have been found to be problematic. These parts could be improved by clarifying some points and making some changes. Some suggestions for updates of the RFC are listed here.

14.2.1 Link hysteresis

In section 7.2.3 the unclarity of the RFC with regards to appliance of the link-hysteresis instability rule, is described. Note that the exponentially smoothed moving average scheme described in the RFC is just a suggested approach.

The last paragraph of section 14.3 in the RFC states:
The loss of OLSR packet is detected by tracking the missing Packet Sequence Numbers on a per interface basis and by "long period of silence" from a node. A "long period of silence may be detected thus: if no OLSR packet has been received on interface I from interface N during HELLO emission interval of interface NI (computed from the Htime field in the last HELLO message received from NI), a loss of an OLSR packet is detected.

This could be updated to:

The loss of OLSR packet is detected by tracking the missing Packet Sequence Numbers on a per interface basis and by "long period of silence" from a node. A "long period of silence may be detected thus: if no OLSR packet has been received on interface I from interface N during HELLO emission interval of interface NI (computed from the Htime field in the last HELLO message received from NI), a loss of an OLSR packet is detected. When such a loss is detected, the registered Packet Sequence Number for that neighbor must be incremented by 1.

### 14.2.2 HNA message format

As discussed in section 7.1.4, the HNA message format could be updated to use the prefix length of the announced networks instead of the netmask. This would reduce the size of HNA messages greatly when using IPv6. The new reserved fields could be used to announce different types of information about the gateway such as bandwidth and current load.

Section 12.1 of the RFC states:

The proposed format of an HNA-message is:

```
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
+-----------------------------------------------
| Network Address                              |
+-----------------------------------------------
| Netmask                                      |
+-----------------------------------------------
| Network Address                              |
+-----------------------------------------------
| Netmask                                      |
+-----------------------------------------------
| ...                                          |
```

This is sent as the data part of the general packet format with the "Message Type" set to HNA_MESSAGE, the TTL field set to 255 and Vtime set accordingly to the value of HNA_HOLD_TIME, as specified in section 18.3.

Network Address

The network address of the associated network
### 14.2.3 HNA and heterogeneous nodes

As explained in section 4.1, HNA routing will not work if all nodes along a HNA route does not implement HNA functionality.

The RFC states:

#### 12.7. Interoperability Considerations

Nodes, which do not implement support for non OLSR interfaces, can coexist in a network with nodes which do implement support for non
14.2.4 Multiple gateways

The action to take when multiple gateways are available for the same network and with the same hop-count could be more specific. This is however very much an implementation issue and therefore no RFC text update is suggested here.

14.2.5 MID processing

As discussed in section 6.8.7, the first registration of information from a MID message does not initiate removal of duplicate entries of the originator. These entries are timed out instead. A better approach is to remove the entries when the MID message is being processed.

Section 5.4 of the RFC states:

2 For each interface address listed in the MID message:

2.1 If there exist some tuple in the interface association set where:

I_iface_addr == interface address, AND
I_main_addr == originator address,

then the holding time of that tuple is set to:
\[ I_{\text{time}} = \text{current time} + \text{validity time}. \]

2.2 Otherwise, a new tuple is recorded in the interface association set where:

\[
I_{\text{iface_addr}} = \text{interface address},
I_{\text{main_addr}} = \text{originator address},
I_{\text{time}} = \text{current time} + \text{validity time}.
\]

This could be updated to:

2 For each interface address listed in the MID message:

2.1 If there exist some tuple in the interface association set where:

\[
I_{\text{iface_addr}} = \text{interface address}, \text{AND}
I_{\text{main_addr}} = \text{originator address},
\]

then the holding time of that tuple is set to:

\[
I_{\text{time}} = \text{current time} + \text{validity time}.
\]

2.2 Otherwise, a new tuple is recorded in the interface association set where:

\[
I_{\text{iface_addr}} = \text{interface address},
I_{\text{main_addr}} = \text{originator address},
I_{\text{time}} = \text{current time} + \text{validity time}.
\]

If there exists some tuple in the 2-hop Neighbor Set where:

\[
I_{\text{iface_addr}} = N_{\text{neighbor_main_addr}}.
\]

Then that tuple is removed.

### 14.2.6 Suggested intervals

The suggested message emission intervals in RFC3626 are:

- HELLO\_INTERVAL = 2 seconds
- TC\_INTERVAL = 5 seconds
Since flooded MID and HNA information in most cases will be very static data, one can benefit from using a larger interval on these. Emission of MID and HNA could in addition, like TC messages, be triggered by changes in the data on which these messages is based.

14.3 Future work

In chapters 6, 7, 9, 11, 12 and 13 future work on the implemented solutions has been discussed. The olsr daemon itself is is reaching a somewhat stable state, even tough there are still room for updates regarding some of the auxiliary functionality. But olsrd has become more and more of a framework for different solutions built on top of the routing protocol. Most of these solutions rely on using OLSRs MPR flooding.

A future project could be to implement the MPR functionality as a service of its own as discussed in [36]. Implementing a broadcast and multicast solution that is transparent to applications like discussed in section 8.4 is also a very interesting future project. Besides this, the extensions described in chapters 9, 11, 12 and 13 are all possible areas for future work in various degrees.

Olsrd will be maintained by the author and possibly others, due to its free license. The olsr.org web-page will also be maintained.

14.4 Final conclusions from the author

This master thesis has presented the work I have done on implementing OLSR and designing and implementing various extensions. I believe that in the end, people will, in most cases pretty much regardless of underlying technology, prefer tools that are easy to set up and that requires little maintenance. Users do not care to much about various protocol designs and underlying functionality, they want solutions that works.

For a MANET routing protocol to become dominant in this segment, it is important that robust and easy to use implementations are available. In my opinion olsrd is such an implementation. It is easy to set up, does not require much resources and it is easy to extend.

The fact that olsrd is implemented in pure C makes it very light-weight and results in few dependencies. But what really separates this implementation from others is the plugin interface. This very flexible functionality makes the implementation suitable for real life usage where companies wants to add some custom functionality to olsrd for their products, as well as for research where new ideas are being tested in real-life scenarios. As an example an U.S. ISP is using the dynamic gateway plugin in their wireless gateways, while a French research institution is using the plugin interface to create an experimental watchdog system that is to provide a more fair forwarding scheme.

The various extensions presented in this thesis are all areas for further research. Still, a solution like the security plugin is currently used for real life scenarios. Lately many free network communities have shown interest in OLSR and the olsrd implementation. This is basically because the implementation is easy to set up and because it works, but also because of the possibilities of adding extensions to tailor olsrd to the users needs. By creating this implementation I believe I have increased the popularity of OLSR with both researchers and end-users. The implementation has been mentioned in widely read technical press such as Der Standard[13], Golem[47] and Heise[58].

Lots can be predicted about the future of wireless communication, but one thing is for sure, wireless technology is here to stay. As more and more of the services currently operated over wired, centralized networks are migrated to wireless communication solutions, more focus will be put on the possibilities of moving beyond the centralized access point paradigm. The MANET working group has laid down some important initial work on mobile ad-hoc routing. However, I believe that MANET routing needs to take other constraints than just hop count into consideration when calculating routes. Issues like bandwidth, delay and
stability should all be taken into consideration. Commercial solutions taking these factors into account, have already emerged based on the proposed MANET protocols. Microsofts Link Quality Source Routing (LQSR) protocol[24], based upon DSR, is one such example.

The routing protocols proposed by the MANET working group today, might never come into large scale usage, but they have formed a fundament for work to come. I am very glad to have been able to be a part of this initial work that might shape our future technology.
Bibliography


Appendix A

olsrd configuration file

An example configuration file for olsrd:

# # UniK OLSR daemon config file # # Lines starting with a # are discarded #

# Debug level(0-9) # If set to 0 the daemon runs in the background

DEBUG 1

# IP version to use (4 or 6)

IPVERSION 4

# HNA IPv4 routes # syntax: netaddr netmask # Example Internet gateway: # HNA4 0.0.0.0 0.0.0.0

HNA4 0.0.0.0 0.0.0.0

# HNA IPv6 routes # syntax: netaddr prefix # Example Internet gateway:

#HNA6 :: 0

#HNA6 fec0:2200:106:: 48

# A list of whitespace separated interface names

INTERFACES eth0 eth1

# Olsrd plugins to load
# This must be the absolute path to the file
# or the loader will use the following scheme:
# - Try the paths in the LD_LIBRARY_PATH
#   environment variable.
# - The list of libraries cached in /etc/ld.so.cache
# - /lib, followed by /usr/lib
#
# ONE PLUGIN PR. LINE

LOAD_PLUGIN olsrd_dot_draw.so.0.1
LOAD_PLUGIN olsrd_secure.so.0.2
LOAD_PLUGIN olsrd_dyn_gw.so.0.1

# IPV4 broadcast address to use. The
# one useful example would be 255.255.255.255
# 'auto' uses the broadcastaddress
# every card is configured with

IP4BROADCAST auto

# IPV6 address scope to use.
# Must be 'site-local' or 'global'

IP6ADDRTYPE site-local

# IPV6 multicast address to use when
# using site-local addresses.
# 'auto' uses the default ff05::15

IP6MULTI-SITE auto

# IPV6 multicast address to use when
# using global addresses
# 'auto' uses the default ff0e::1

IP6MULTI-GLOBAL auto

# Polling rate in seconds(float).
# Auto uses default value 0.1 sec

POLLRATE auto

# Hello interval in seconds(float)
# auto uses RFC proposed value
# This is used on WLAN interfaces.

HELLOINTERVAL auto

# HELLO hold time as a multiplier
# of the HELLOINTERVAL. Auto is the
# RFC proposed interval

HELMULTI auto
# HELLO interval for sending
# interval/holding time for wired
# links. This is a multiplier of
# the HELLOINT value. Auto is 2

NWHELLOINT auto

# HELLO hold time for wired links,
# as a multiplier of the NWHELLOINT.
# Auto is NWHELLOINT * 3.

NWHELLOMULTI auto

# TC interval in seconds(float)
# auto uses RFC proposed value

TCINT auto

# TC hold time as a multiplier
# of the TCINT. Auto is the
# RFC proposed interval

TCMULTI auto

# MID interval in seconds(float)
# auto uses RFC proposed value

MIDINT auto

# MID hold time as a multiplier
# of the MIDINT. Auto is the
# RFC proposed interval

MIDMULTI auto

# HNA interval in seconds(float)
# auto uses 3*TCINT

HNAINT auto

# HNA hold time as a multiplier
# of the HNAINT. Auto is the
# RFC proposed interval

HNAMULTI auto

# TOS(type of service) value for
# the IP header of control traffic.
# Auto is 16

TOSVALUE auto

# The fixed willingness to use(0-7)
# or "auto" to set willingness dynamically
# based on battery/power status

WILLINGNESS    auto

# Allow processes like the GUI front-end
# to connect to the daemon. 'yes' or 'no'

IPC-CONNECT    no

# Whether to use hysteresis or not
# Hysteresis adds more robustness to the
# link sensing but delays neighbor registration.
# Used by default. 'yes' or 'no'

USE_HYSTERESIS yes

# Hysteresis parameters
# Do not alter these unless you know
# what you are doing!
# Set to auto by default. Allowed
# values are floating point values
# in the interval 0,1
# THR_LOW must always be lower than
# THR_HIGH!!

HYST_SCALING    auto

HYST_THR_HIGH   auto

HYST_THR_LOW    auto

# TC redundancy
# Specifies how much neighbor info should
# be sent in TC messages
# Possible values are:
# 0 - only send MPR selectors
# 1 - send MPR selectors and MPRs
# 2 - send all neighbors
# auto - defaults to 0

TC_REDUNDANCY   auto

# MPR redundancy
# Specifies how many MPRs a node should
# try select to reach every 2 hop neighbor
# Can be set to any integer >0
# auto - defaults to 1

MPR_COVERAGE    auto
Appendix B

olsrd manual page

olsrd(8)  olsrd(8)

NAME
olsrd - Optimized Link State Routing protocol daemon

SYNOPSIS
olsrd [-i interface1 [interface2 ...] ] [-f configfile ] [-d debuglevle] [ -ipv6 ] [ -ipc ] [-dispin ] [-dispout ] [ -tnl ] [ -bcast broadcastaddress ] [ -delgw ] [-hint HELLO interval for wireless interfaces ] [ -tcint TC interval ] [ -midint MID interval ] [ -hmaint HNA interval ] [ -hhold HELLO validity time ] [ -thold TC validity time ] [ -tos TOS value ] [ -nhint HELLO interval non WLAN ] [ -nhhold HELLO validity time non WLAN ] [ -T scheduler poll rate ]

DESCRIPTION
olsrd is an implementation of the Optimized Link State Routing protocol for Mobile Ad-Hoc networks (MANET). The protocol is described in RFC3626. It is designed to be run as a standalone server process but as it is still in an experimental stage most users will prefer running it with some debug output which is directed to STDOUT.

This manual page only lists the command line arguments. For details of the configuration file see the comments included in /etc/olsrd.conf. Note that none of these options need to be set at the command line - all these options and others can be set in the configuration file.

The homepage of olsrd is http://www.olsr.org

OPTIONS
- i interface1 ... interfaceN
    This option specifies on what network interfaces olsrd should run. These interfaces cannot be aliased interfaces such as eth0:1.

- f configfile
    This option overrides the default configuration file path used by olsrd - /etc/olsrd.conf
-d debuglevel
This option specifies the amount of debug information olsrd should write to STDOUT. If set to 0 olsrd will run in the background.

-ipv6
This option instructs olsrd to use the Internet Protocol version 6. The default is version 4.

-ipc
This option allows the GUI front-end created from olsrd to connect to olsrd at runtime.

-dispin
This option, when set, causes olsrd to display all incoming packet data on STDOUT. When using IPv4 the data is displayed in decimal format, when using IPv6 the data is displayed in hexadecimal format.

-dispout
This option, when set, causes olsrd to display all outgoing packet data on STDOUT. When using IPv4 the data is displayed in decimal format, when using IPv6 the data is displayed in hexadecimal format.

-tnl
When this option is set olsrd will use IP-in-IP tunneling to Internet gateways. This is very experimental code and it should not be used as of yet.

-delgw
If this option is set olsrd will remove any default routes set prior to adding an Internet route based on OLSR routing.

-bcast broadcastaddress
This option specifies what IPv4 broadcastaddress to use for OLSR control traffic. The only value that currently makes sense when setting broadcast address manually is 255.255.255.255. The default action is to use the broadcastaddress that the network interface is preconfigured with (per interface).

-hint seconds
This value sets the interval on which HELLO messages should be generated. The value is a floating point number representing seconds.

-nhint seconds
This value sets the interval on which HELLO messages should be generated on interfaces that are not detected to be wireless. The value is a floating point number representing seconds.

-tcint seconds
This value sets the interval on which TC messages should be generated. The value is a floating point number representing seconds.

-midint seconds
This value sets the interval on which MID messages should be
generated. The value is a floating point number representing
seconds.

-htmaint seconds
  This value sets the interval on which HELLO messages should be
generated. The value is a floating point number representing
seconds.

-nhhold multiplier
  This option sets the announced HELLO validity time as a multi-
plier of the HELLO interval.

-nhhold multiplier
  This option sets the announced HELLO validity time on non-
WLAN interfaces, as a multiplier of the HELLO interval.

-thold multiplier
  This option sets the announced TC validity time as a multi-
plier of the TC interval.

-tos TOS-value
  This option sets the type of service value that should be set
in the OLSR control traffic packet IP headers.

-T seconds This option sets the polling interval of the scheduler. The
default is 0.1 seconds. This option should only be considered
if running with really low emission intervals.

FILES /etc/olsrd.conf

SEE ALSO
iwconfig(8) route(8).

Jun 2004

olsrd(8)
Appendix C

olsr_protocol.h

This is the headerfile defining all OLSR protocol specific packets, datatypes, constants and default values.

/*
 * OLSR ad-hoc routing table management protocol
 * Copyright (C) 2003 Andreas Tømmesøn (andreto@ifi.uio.no)
 *
 * This file is part of the UniK OLSR daemon.
 *
 * The UniK OLSR daemon is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * The UniK OLSR daemon is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with the UniK OLSR daemon; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
*/

/*
 * Values and packet formats as proposed in RFC3626 and misc. values and
 * data structures used by the UniK olsr daemon.
 */

#ifndef _PROTOCOLS_OLSR_H
#define _PROTOCOLS_OLSR_H

/* Port for OLSR to use */
#define OLSRPORT 698

/* Default IPv6 multicast addresses */
#define OLSR_IPV6_MCAST_SITE_LOCAL "ff05::15"
#define OLSR_IPV6_MCAST_GLOBAL "ff0e::1"

/* types */
#include <sys/types.h>

typedef u_int8_t olsr_u8_t;
typedef u_int16_t olsr_u16_t;
typedef u_int32_t olsr_u32_t;
typedef int8_t olsr_8_t;
typedef int16_t olsr_16_t;
typedef int32_t olsr_32_t;

/* IPv6 address format in6_addr */
#include <netinet/in.h>

union olsr_ip_addr
{
    olsr_u32_t v4;
    struct in6_addr v6;
};

/*
* Emission Intervals
*/
#define HELLO_INTERVAL 2
#define HELLO_INTERVAL_NW HELLO_INTERVAL * 2
#define REFRESH_INTERVAL 2
#define TC_INTERVAL 5
#define MID_INTERVAL TC_INTERVAL
#define HNA_INTERVAL TC_INTERVAL

/*
* Holding Time
*/
#define NEIGHB_HOLD_TIME 3 * REFRESH_INTERVAL
// *extra: time to delete for non-wireless interfaces */
#define NEIGHB_HOLD_TIME_NW NEIGHB_HOLD_TIME * 2
#define TOP_HOLD_TIME 3 * TC_INTERVAL
#define DUP_HOLD_TIME 30
#define MID_HOLD_TIME 3 * MID_INTERVAL
#define HNA_HOLD_TIME 3 * HNA_INTERVAL

/*
* Scaling factor
*/
#define VTIME_SCALE_FACTOR 0.0625

/*
* Message Types
*/
#define HELLO_MESSAGE 1
#define TC_MESSAGE 2
#define MID_MESSAGE 3
#define HNA_MESSAGE 4
#define MAX_MESSAGE 4

/*
 * Link Types
 */
#define UNSPEC_LINK 0
#define ASYM_LINK 1
#define SYM_LINK 2
#define LOST_LINK 3
#define HIDE_LINK 4
#define MAX_LINK 4

/*
 * Neighbor Types
 */
#define NOT_NEIGH 0
#define SYM_NEIGH 1
#define MPR_NEIGH 2
#define MAX_NEIGH 2

/*
 * Neighbor status
 */
#define NOT_SYM 0
#define SYM 1

/*
 * Link Hysteresis
 */
#define HYST_THRESHOLD_HIGH 0.8
#define HYST_THRESHOLD_LOW 0.3
#define HYST_SCALING 0.5

/*
 * Willingness
 */
#define WILL_NEVER 0
#define WILL_LOW 1
#define WILL_DEFAULT 3
#define WILL_HIGH 6
#define WILL_ALWAYS 7

/*
 * Redundancy defaults
 */
#define TC_REDUNDANCY 0
#define MPR_COVERAGE 1

/*
 * Misc. Constants
 */
#define MAX_JITTER 4
#define MAX_TTL 0xff
/*
 * Sequence numbering
 */
/* Seqnos are 16 bit values */
#define MAXVALUE 0xFFFF

/* Macro for checking seqnos "wraparound" */
#define SEQNO_GREATER_THAN(s1, s2)
   \((s1 > s2) \&\& (s1 - s2 <= (MAXVALUE/2))\)
   \| \((s2 > s1) \&\& (s2 - s1 > (MAXVALUE/2)))\)

/*
 * Macros for creating and extracting the neighbor
 * and link type information from 8bit link_code
 * data as passed in HELLO messages
 */
#define CREATE_LINK_CODE(status, link) (link | (status<<2))
#define EXTRACT_STATUS(link_code) (((link_code & 0x1C)>>2)
#define EXTRACT_LINK(link_code) (link_code & 0x3)

/*
 * Macros for comparing and copying IP addresses
 */
#define COMP_IP(ip1, ip2) (!memcmp(ip1, ip2, ipsize))
#define COPY_IP(to, from) memcpy(to, from, ipsize)

/****************************
 * OLSR packet definitions   *
 ****************************/

/*
 * HELLO message
 */
struct hellinfo
{
   olsr_u8_t link_code;
   olsr_u8_t reserved;
   olsr_u16_t size;
   olsr_u32_t neigh_addr[1]; /* neighbor IP address(es) */
};

struct hellomsg
{
   olsr_u16_t reserved;
   olsr_u8_t htime;
   olsr_u8_t willingness;
   struct hellinfo hell_info[1];

127

*/
*IPv6
*/
struct hellinfo6
{
  olsr_u8_t    link_code;
  olsr_u8_t    reserved;
  olsr_u16_t   size;
  struct in6_addr neigh_addr[1]; /* neighbor IP address(es) */
};

struct hellomsg6
{
  olsr_u16_t   reserved;
  olsr_u8_t    htime;
  olsr_u8_t    willingness;
  struct hellinfo6 hell_info[1];
};

/*
 * Topology Control message
 */
struct neigh_info
{
  olsr_u32_t    addr;
};

struct tcmsg
{
  olsr_u16_t    ansn;
  olsr_u16_t    reserved;
  struct neigh_info neigh[1];
};

/*
 *IPv6
 */
struct neigh_info6
{
  struct in6_addr   addr;
};

struct tcmsg6
{
  olsr_u16_t    ansn;
  olsr_u16_t    reserved;
  struct neigh_info6 neigh[1];
};

/*
 *Multiple Interface Declaration message
 */
/\*  
  * Defined as a struct for further expansion  
  * For example: do we want to tell what type of interface  
  * is associated with each address?  
  */  
struct midaddr  
{  
osr_u32_t addr;  
};  
struct midmsg  
{  
  struct midaddr mid_addr[1];  
};  
/*  
* IPv6  
*/  
struct midaddr6  
{  
  struct in6_addr addr;  
};  
struct midmsg6  
{  
  struct midaddr6 mid_addr[1];  
};  
/*  
* Host and Network Association message  
*/  
struct hnapair  
{  
osr_u32_t addr;  
osr_u32_t netmask;  
};  
struct hnamsg  
{  
  struct hnapair hna_net[1];  
};  
/*  
* IPv6  
*/  
struct hnapair6  
{  
  struct in6_addr addr;  
  struct in6_addr netmask;  
};  
struct hnamsg6  
{  
  struct hnapair6 hna_net[1];  
}
*/
 * OLSR message (several can exist in one OLSR packet)
 */

struct olsrm 
{
  olsr_u8_t  olsr_msgtype;
  olsr_u8_t  olsr_vtime;
  olsr_u16_t olsr_msgsiz 
  olsr_u32_t originator;
  olsr_u8_t  ttl;
  olsr_u8_t  hopcnt;
  olsr_u16_t seqno;

union 
{
  struct hellomsg hello;
  struct tcmsg   tc;
  struct nnmsg  hna;
  struct midmsg mid;
} message;
};

/*
 *IPv6
 */

struct olsrm6 
{
  olsr_u8_t  olsr_msgtype;
  olsr_u8_t  olsr_vtime;
  olsr_u16_t olsr_msgsiz 
  in6_addr originator;
  olsr_u8_t  ttl;
  olsr_u8_t  hopcnt;
  olsr_u16_t seqno;

union 
{
  struct hellomsg6 hello;
  struct tcmsg6   tc;
  struct nnmsg6  hna;
  struct midmsg6 mid;
} message;

/*
 * Generic OLSR packet

*/

struct olsr
{
  olsr_u16_t olsr_packlen; /* packet length */
  olsr_u16_t olsr_seqno;
  struct olsmesg olsr_msg[1]; /* variable messages */
};

struct olsr6
{
  olsr_u16_t olsr_packlen; /* packet length */
  olsr_u16_t olsr_seqno;
  struct olsmesg6 olsr_msg[1]; /* variable messages */
};

/* IPv4 <-> IPv6 compability */

union olsmesg_message
{
  struct olsmesg v4;
  struct olsmesg6 v6;
};

union olsmesg_packet
{
  struct olsr v4;
  struct olsr6 v6;
};

#endif
Appendix D

The UniK - OLSR plugin library

To be presented at the OLSR interop workshop, San Diego, August 6-7 2004
The UniK - OLSR plugin library
Andreas Tønnesen, Andreas Hafslund, and Øivind Kure

Abstract—In this paper we present a plugin for OLSR. This plugin is used for adding an interface for between the olsrd daemon and other applications. Using the plugin other applications, ranging from network services such as DNS, to broadcast voice traffic can use the broadcast mechanisms from the OLSR protocol. This means that the applications can flood the ad hoc network using the optimized MPR-functionality, and thus reducing the network load.

Index Terms—Mobile. Ad Hoc Network, OLSR, dynamically linked libraries, service broadcasting.

Introduction
As mobile ad hoc networks (MANETs) [1] are an area for research and development, the ability to add extensions or change normal operation in implementations of routing protocols for such networks, provides a great way of testing new solutions.

The MPR, flooding and default forwarding algorithm used in OLSR [2] makes this protocol very interesting to extend. Normal MANET routing suffers from lack of broadcast and multicast solutions. By letting OLSR carry traffic, one can provide a broadcast solution that is optimized. The OLSR daemon will then work as a flooding relay agent for local applications. Other interesting extensions can be updating OLSR parameters at runtime (for instance changing MPR willingness based on power consumption), based on traffic analysis or creating visualizations of the network topology.

Already existing services that require a broadcast mechanism can be used in a MANET routed by olsrd if using an olsrd plugin to flood broadcasted traffic. Such services include DNS, service discovery mechanism, key distribution schemes etc. Utilizing such protocols in MANETs have been studied in [2] and [3].

As modularity was one of the main goals when designing and implementing the UniK olsrd, the idea of easily extending the protocol led to the design of a plugin interface. In this paper this interface, areas of usage and some example plugin implementations are covered.

Flow of OLSR Plugins
Olsrd supports loading of dynamically linked libraries, called plugins as explained later, for generation and processing of private packet types and any other custom functionality. There are two main reasons for using the OLSR plugins: (1) for sending broadcast traffic in the MANET using OLSR, (2) for changing the OLSR functionality using the plugin interface. The former is for other applications than the

\[\text{Figure 1: Example of how a plugin intercepts an applications program flow and adds its own.}\]

DLL functionality exists for all common operating systems. In Linux they are known as .so files while in Microsoft Windows they are known as .DLL files.

B. Why use plugins?
The plugin design was chosen for, amongst others, the following reasons:

- No need to change any code in the OLSR daemon to add custom packages or functionality.
- Users are free to implement olsrd plugins and license them under whatever terms they like. Olsrd is GPL licensed meaning that any alteration of the olsrd code itself must be publicly released.
- Plugins can be compiled in any language that can be compiled as a dynamic library.
- No need for people using extended OLSR functionality to rely on heavy patching to maintain functionality, when new olsrd versions are released. The plugin interface will always be backwards
OLSR provides a default forwarding algorithm, that allows for forwarding of OLSR messages of unknown types. This means that even if only a subset of the nodes in the network actually know how to interpret a certain message type, all nodes will forward it according to the MPR scheme. A wide variety of services designed for wired network environments rely on net-wide broadcasts. Services that need to broadcast/multicast data can encapsulate data in a private OLSR message type using an olsrd plugin as illustrated in Figure 2.

![Figure 2: Example of how a plugin can enable the OLSR daemon to work as a relay for broadcasting. The Local application and the plugin communicate using the interprocess communication.](image)

The design of the various entities of olsrd allows one to easily add special functionality into most aspects of the program. Olsrd can both register and unregister functions with the socket parser, packet parser and scheduler, and one can update many variables, manipulate all outgoing traffic and more. This opens up for possibilities like intercepting current operation and replacing it with custom actions. As an example, a plugin can provide its own HELLO message generation and parser functions. The plugin can then unregister the default functions used by olsrd and replace them with its own. This means that the OLSR tables can be freed, and a new set of operations can be executed on the tables. This relationship is illustrated in Figure 3.

![Figure 3: A plugin can manipulate virtually every part of the olsrd daemon.](image)

The modular design of olsrd really shows its strengths when dealing with plugins. A plugin can do things like establish blocking sockets for communication of its own without blocking olsrd operation. This is because the plugin can register its sockets with the socket parser in olsrd where the socket will be part of the main socket(2) set.

III. THE OLSR PLUGIN INTERFACE

For a plugin scheme like this to work, one needs a well-defined easy-to-expand interface for communication between the OLSR daemon and plugins. The interface should be well-defined so that a plugin always knows what to expect from the daemon, and the daemon always knows what to expect from the plugin within some given set of functions. The design should be flexible enough to allow for extending the functionality while keeping backwards compatibility.

The actual data that must be set up between the application and the plugin are pointers to variables and functions. The olsrd plugin interface is mainly based on the function:

```c
int olsr_plugin_init(int cmd, void *data, size_t size)
```

This function is similar to the `ioctl(2)` function in syntax. One passes a command and a pointer to some allocated memory and the size of the allocated memory area. The return value indicates success or error, while actual data is put or read from the memory buffer `*data`. This function is implemented in `src/plugins.c`, while all the commands are defined in `src/olsr_plugin_cmds.h`.

Let us look at an example of how a plugin can use a function implemented in olsrd. The function `get_msg_seqno` returns the next message sequence number for OLSR to use when transmitting an OLSR packet. If we want to be able to use this function in our plugin, we will typically execute something like:

```c
/* Define this function-pointer somewhere */
int olsr_plugin_init (_register olsrd_plugin_data
    struct olsrd *olsrd)
{
    set_others(olsrd); /* do some initialization */
    return 0;
}
```

To be able to access the `olsr_plugin_init` function, the plugin needs to be initialized from olsrd. The file `src/plugins.c` implements the plugin loader code. For the plugin loader to be able to set up the needed pointers, the plugin must provide the following function (in addition to some variables and other functions):

```c
int register_olsr_plugin(struct olsrd *olsrd)
```

This function is called from the olsrd plugin loader passing a pointer to a struct `olsrd_plugin_data`, which contains the pointers to olsrd functions that the plugin needs to be able to set up all needed data-pointers. After this the plugin is responsible for fetching all needed pointers from the olsrd daemon. The process of initializing a plugin is illustrated in Figure 4.
IV. Two example plugins

Two plugins that perform trivial tasks are implemented as example code. Here follows a brief explanation of what they do and how they are designed. Both plugins are part of the olsrd source code package available for download from http://www.olsrd.org. The example plugin source code resides in the lib/ directory relative to the olsrd source code root directory.

A. The powerstatus plugin

This plugin is to provide a solution where the powerstatus of nodes running the plugin in the MANET are distributed and registered. This information is made available to the user and other processes through IPC using a TCP channel. The plugin should not affect node running without this functionality. The reason we use a TCP channel for the communication is to show how the application can communicate with the plugin over standard TCP. This could be configured using the default OLSR configuration.

A node is to periodically flood the network with packet containing the following information:

- Whether or not the node is battery powered.
- Estimated lifetime left on the battery, if battery powered.
- Percentage of power left on the battery, if battery powered.

This should result in a scenario where all powerstatus-enabled nodes have an up-to-date understanding of the powerstatus of all other powerstatus-enabled nodes. Even though the powerstatus-enabled nodes might only be a subset of the nodes in the MANET, the default forwarding algorithms will ensure diffusion of the information. The following is implemented in the plugin:

A message format to carry the power information:

To take advantage of the default forwarding scheme in OLSR, the power information, extracted from proc/stat, has to be transmitted as an OLSR message. This message format is displayed in Figure 5.

The message is encapsulated in a regular OLSR message header with a message type from the OLSR private message types (126-255).

<table>
<thead>
<tr>
<th>Bit</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Powerstatus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: The message format used by the powerstatus plugin. This data is sent as the data part of a regular OLSR message.

An information repository:

To keep an up-to-date database of information of power, an information repository similar to those used in olsrd, is implemented. This is based on IP based output lists with statically allocated root elements. Periodic generation of powerstatus messages based on powerstatus:

To transmit messages about the status of power on a periodic interval, a generation function is implemented. This function is registered with the olsrd scheduler at plugin initialization. The function polls the /proc/stat file for information about the power status and builds a message based on the information. This message is then flooded through olsrd.

Parsing of powerstatus message:

A message parse function is registered with the olsrd message parser at plugin initialization to receive all incoming powerstatus messages. This function updates the information repository based on the contents of incoming packages. The function is also responsible for forwarding the message.

Timeout of the repository:

A function that traverses the information repository and removes timed out entries is registered with the olsrd scheduler to run at a given interval. Since it is not critical that timed out entries are removed as soon as possible, this function is not registered to be executed at every olsrd scheduler poll.

IP functionality:

IP is done over a TCP socket via the loopback interface. Since this plugin example is not to listen for input, but just to do an output, there seems to be no need to register the TCP socket with the olsrd socket parser. However, as we want to be able to connect to this interface to the plugin at any time, the plugin must register the socket with olsrd to listen for incoming connections. The socket listens for connections on TCP port 8888 and only accepts connections from the local host (127.0.0.1).

B. The dynamic Internet gateway plugin

Nodes in a MANET might dynamically obtain and lose Internet connectivity through interfaces not participating in the MANET routing. A typical scenario would be a laptop that might be connected to the Internet through an Ethernet link for a limited time.

A plugin that dynamically updates the IFN information announced by the local node has been implemented. This plugin utilizes the local node has an Internet connection and updates the local IFN list based on this.

This plugin is a good example of using plugins for other
tasks than packet transmission. Combining this plugin with an automatic network cable detection daemon such as [5] would be a good idea. Only IPv4 is supported by the plugin as of now. This plugin has been used in our recent paper [6].

**Detecting Internet routes**

The main object of this plugin is to poll for an Internet route and add or remove such a route from the local FNA set if a change is detected. An Internet connection is identified by a default gateway with a hop count of 0. So a route to 0.0.0.0/0 with metric 0 is considered an Internet route. Since OLSR uses a hop count metric bigger than 0 on all routes, this plugin will not react to Internet gateways added by olsrd.

To poll for route updates, a function that searches the kernel routing table for a default gateway is registered with the olsrd scheduler to be executed regularly on a given interval. If a new Internet route with metric 0 is discovered, the plugin will add this entry to the local FNA set by calling the function:
```
void add_local_fna_entry(union olsr_ip_addr *net, union intip_netmask *mask);
```

This function has been fetched from olsrd through the plugin interface.

Whenever such a registered Internet route is removed from the kernel routing table, the local FNA entry is also removed using the function:
```
void remove_local_fna_entry(union olsr_ip_addr *net, union intip_netmask *mask);
```

This enables nodes to act as Internet gateways whenever they have some Internet connectivity not set up by OLSR itself.

**IPC**

The dynamic Internet gateway plugin also offers IPC to read output. Just like with the powersys plugin, all communication is outbound, but since clients should be able to connect at any time, the IPC server socket is registered with the olsrd parser of olsrd.

The IPC socket listens on TCP port 9999 and only allows connections from the local host (127.0.0.1).

**Conclusion and Further Study**

We have designed and implemented a plugin for the Unix OLSR routing daemon. The plugin functionality has been used for adding extended functionality to the OLSR. Also, the plugin has been used for sending broadcast traffic different than the regular routing messages. We have used the plugin for:

- A secure extension to the OLSR protocol [7].
- An autoconfiguration mechanism [8].
- Multicasting/broadcasting of Voice using OLSR, to be released this autumn.
- Distributed DNS using OLSR, to be released this autumn.

There are several aspects with the OLSR plugin we wish to improve. One such thing is making the plugin more modular with respect to the applications. This means that applications using broadcast traffic could function without any changes with regards to the OLSR plugin. This is not the case today, but will be included in later releases.

**Acknowledgment**

We would like to thank Ron Bjornn Robseik and Jon Anderson at Thales Communications AS for their valuable help during our discussions.

**References**


136
Appendix E

Secure extension to the OLSR protocol

To be presented at the OLSR interop workshop, San Diego 6-7, August 2004
Secure Extension to the OLSR protocol

Andreas Hafslund, Andreas Tornesen, Roar Bjørgum Rotvå, Jon Andersson and Øivind Kure

Abstract—In this paper we present a mechanism for securing the OLSR protocol. This mechanism is implemented as an extension to the OLSR source code provided by UniB, and can be downloaded freely from the Internet. The proposed and implemented mechanism is based on signing each OLSR control packet with a digital signature to authenticate the message. Also, the mechanism provides a timestamp exchange process. The timestamps are used to prevent replay attacks on the routing protocol. This security mechanism does not need synchronized time.

Index Terms—Mobile Ad Hoc Network, Security, OLSR, Digital signatures

I. INTRODUCTION

A mobile ad hoc network (MANET) [1] is usually considered to be a network consisting of mobile nodes with wireless network interfaces. Each node can function both as an end-host, but also as an intermediate router for other nodes in the network. The mobility of the nodes makes the network topology dynamic.

Today wired computer systems can be made secure at a high degree, but when it comes to wireless networks weak security is often used if any security measurements are taken at all. This affects the services running on wireless networks including MANET routing protocols.

The OLSR [2] protocol is the current target for this investigation. We have made an implementation of a system for improving the security of the OLSR protocol with digital signatures in the routing messages. The implementation is an extension to the OLSR protocol, and secures only the routing messages itself, not the user traffic being routed in the MANET. Also the solution only provides integrity and non-repudiation, although the solution is extendable and could include mechanisms to provide confidentiality at a later stage.

This paper is divided into 5 sections. In section 2 we outline our proposed mechanisms for improving the security of OLSR, whereas Section 3 briefly considers some implementation issues. We discuss related work in Section 4, and conclude this paper in Section 5.

II. SECURING THE OLSR PROTOCOL

In this study we have chosen a security mechanism based upon signing each OLSR control packet with a digital signature for authenticating the messages. The digital signature is based on asymmetric keys.

A. Overview

All OLSR control traffic is signed for every hop. This means that one does not have to consider variable fields in messages, such as hop count and TTL. It also means that only one signature is needed, although several OLSR messages are stacked in one OLSR packet. Using this hop-by-hop approach does not provide end-to-end signatures, which again means that the digest is not a true signature with respect to the originator, but rather a signature from the forwarder, ensuring that it trusts the source of the message in the previous hop.

A node that does not have access to the shared secret key cannot produce a verifiable digest. All receivers running secure OLSR discard messages with non-verifiable digests. Signatures are transmitted in OLSR messages of their own.

This is to ensure compatibility with nodes not running secure OLSR, and because a timestamp is transmitted in addition to the signature.

Four different messages are defined. One is the actual signature message, as displayed in Figure 1, and three messages (Fig. 2 to Fig. 4) used in the timestamp exchange. All the messages are sent as the message body of an OLSR message.

To prevent replay attacks, timestamps are used in our secure extension to OLSR. To exchange these timestamps upon initial connection between two nodes, a two-way timestamp exchange mechanism is utilized. The solution does not rely on synchronized time.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Scheme</th>
<th>Algorithm</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>Timestamp</td>
<td>Signature(160 bits)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: The basic signature message.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Random value &quot;challenge&quot;</td>
</tr>
<tr>
<td></td>
<td>Signature(160 bits)</td>
</tr>
</tbody>
</table>

Figure 2: The timestamp exchange challenge message.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Random value &quot;challenge&quot;</td>
</tr>
<tr>
<td></td>
<td>Timestamp</td>
</tr>
<tr>
<td></td>
<td>Response Signature(160 bits)</td>
</tr>
<tr>
<td></td>
<td>Signature(160 bits)</td>
</tr>
</tbody>
</table>

Figure 3: The timestamp exchange challenge-response message.
The signature message, illustrated in Fig. 1, is attached to all outgoing OLSR packets. This message is to be the last message in the packet. The OLSR packet header is adjusted to include the size of the signature message in the sizeof.

The scheme and algorithms fields in the signature message header, informs the receiver of what signature scheme and what algorithms is being used. The timestamp field is for preventing replay attacks. The digest used as a signature is a hash created using the SHA-1 hashing algorithm, which produces an irreversible 160-bit digest. The hash is based on the following:

- The OLSR packet header (with adjusted size).
- All OLSR messages in the packet except the signature message.
- The OLSR message header, the sub header and the timestamp of the signature message.
- The shared secret key.

No considerations have to be taken regarding the variable fields of different headers (such as TTL or hop count) as this signing is done hop-by-hop.

The timestamp approach needs some further study, especially considered for time synchronization problems, MAC delay etc.

A flexible solution

Secure OLSR is designed to be as flexible as possible with regards to the encryption and hashing algorithms being used, and the entire signature scheme. In the implemented solution digest created by the SHA-1 hashing algorithm, of the message and a shared secret key is used to ensure integrity.

A different scheme could include one signature for each message allowing for end-to-end signing or asynchronous or synchronous encryption could be used to ensure confidentiality as well. These schemes could again utilize different algorithms for hashing and encryption all being defined by the signature message header.

E. Timestamps and freshness

An attacker can record signed traffic and play it back at a later stage (replay attacks). This can be prevented to some degree by sequence numbers, which are already utilized in OLSR. However, for traffic that is only to be sent one hop like HELLO messages, this is of little or no help. An attacker can simply record all messages transmitted by a node, and then move to another area of the network where the HELLO messages recorded was never heard. Here the attacker can start the replay attack by transmitting the recorded messages.

The OLSR sequence numbers are also weak because of their length. They are only 16-bit values and wrap-around will occur rather frequent. The wrap-around mechanism used in OLSR makes the signature numbers even weaker with respect to freshness.

The exchange of timestamps between two neighbor hosts A and B can be described as:

\[ A \rightarrow B: \text{Ch}(M, K) \]
\[ B \rightarrow A: \text{Ch}(\text{ID}(\text{IP}_x, \text{CH}_K), D(M, K), K) \]
\[ A \rightarrow B: \text{Ts}(\text{IP}_x, \text{CH}_K, D(M, K), K) \]

When A receives a signed message form a neighbor B for which A has no registered time value, A initiates the timestamp exchange process. A first sends a challenge message (Fig. 2) to B. This message is broadcasted since A might not have an actual route to B. The challenge message contains the IP address of B and a 32-bit nonce (number used once), \( \text{CH} \). This is a random number, which is used to append random data to real data to prevent a replay attack. A then signs this message with a digest of the entire message and the shared key \( D(M, K) \).

B now has to respond to this message with a challengeresponse message. B first generates the digest of its IP address (if B is multi-homed, the IP address fetched from the challenge message used), the received nonce and the shared key \( D(M_x, CH, K) \). B then generates a 32-bit nonce and transmits the IP address of A, the nonce, the timestamp of B, the digest \( D(IP_x, CH, K) \) and a digest of the entire message and the shared key \( D(M, K) \).

When A receives the challenge-response message from B, it first tries to validate the data. If the digests \( D(IP_x, CH, K) \) and \( D(M, K) \) can be verified, then the timestamp of B is used to calculate the difference of time between A and B. A then generates a response-message (Figure 5) and broadcasts it to B. This message contains the IP address of the receiver (\( B_x \)), a timestamp, a digest of its address (as received from B), the nonce received from B, the shared key \( D(M, K) \) and a digest of the entire message and the key \( D(M, K) \).

When B receives the response-message from A, it tries to verify the digests. If they can be verified then B uses the received timestamp to register its time difference to A. And the timestamp exchange is complete.

The solution does not require synchronized time but the clocks are assumed to be relatively synchronized, meaning that they are running on a relatively equal frequency. All timestamps are represented with a 32 bits value containing the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero. The time steps are divided into 16-bit units with each step containing 16-bit units. The number of 16-bit units elapsed since the last zero is counted as the number of 16-bit units elapsed since the last zero.
seconds since the epoch. Time-stamps are then at first reported as
\( T' = T_l - T_r \), where \( T_l \) is the local timestamp and
\( T_r \) is the remote timestamp received through the timestamp
exchange.

When receiving a signature message a certain slack \( S \) in
the calculated timestamp difference is allowed. So that a
signature message with a verified digest and a timestamp
difference \( T_r \) so that \( T_l - S < T_r \) and \( T_l + S > T_r \), where \( T_r \) is the stored timestamp difference of the sender, is
considered a verified signature message.

To compensate for a possible skew between clocks, the
timestamp difference is recalculated for every received and
verified signature message. The difference is recalculated as
\( T_o + T_r/T_o \), where \( T_o \) is the recorded timestamp difference
and \( T_r \) is the difference calculated based on the received
timestamp.

F. Robustness

The timestamp exchange process could be exploited by an
adversary to create an overload of processing and network
usage, which could lead to the node not being able to
participate in other timestamp exchanges or perhaps any
communication at all. This would be a typical Denial of
Service (DoS) attack. An attacker or a badly configured host
could for instance transmit thousands of the timestamp
exchange challenge messages within a very short period of
time, all aimed at the same host. This would cause the
receiving host to generate and transmit signed replies to all
the challenges.

To avoid this a timer is set for the originators of all
received challenges. Any new received challenges from the
same host while the timer has not timed out are discarded.
Due to the signing of the challenge messages, an attacker
cannot spoof the sender address of challenge messages. An
attacker could however record all challenge messages directed
to a host for a long period of time and launch them all within
a short period of time. However, as timestamp entries are
cached within nodes, this amount of messages would not be
extensive.

III. IMPLEMENTATION ISSUES

Our secure OLSR proposal is implemented as an osd*
plugin. The UniK OLSR plugin is described in [3]. The
implementation includes message signing and timestamp
exchange, and is part of the osd source code available for
download at www.unik.org.

A. Overview

Implementing functionality that was to work on all
incoming and outgoing "raw" OLSR traffic required an
extension to the network output functioning in osd*. An
overview of the relations between the plugin and osd* is
illustrated in Fig. 5. The implementation is to be as
transparent to the osd* code as possible. Therefore all
incoming traffic is passed to the plugin, which verifies the
packet and removes the signature message and updates the
size field of the OLSR packet header. For outgoing traffic
the opposite goes. All outgoing OLSR traffic is passed to the
plugin, which adds the signature and updates the packet size,

\[ r = \frac{r_{\text{old}} + r_{\text{new}}}{2} \]

Figure 5: An illustration of the design of the secure plugin,
as related to osd*.

B. Shared Key

The key used is read from the file /root/sockenv/sock-env.key.
It is 128-bits of size. If no key can be read from the file, the
plugin will terminate the osd* process with a warning
message. Other solutions might be implemented in future
versions to handle local key management better or to be able
to work in an integrated fashion with some authentication
scheme.

C. Interception of incoming traffic

The secure OLSR plugin must be able to intercept all
incoming OLSR traffic and check the signature if present.
This is done quite easily due to the modular structure of osd*.
The plugin de-registers all the OLSR sockets from the socket
listener, and then registers them again with the plugins own
input function. This is implemented in socket_passer.c.
The OLSR sockets can all be redirected from the global
interface list osd. The plugins own OLSR input function
keeps the registered message parser functions and only differs
from the original input function in osd* on two points: (i) An
incoming packet is checked for timestamp exchange
messages, which are processed before the signature check.
Keep in mind that these packets contain signatures of their
own, (ii) An incoming packet is checked for an ending
signature message. If no such message is found the packet
is not considered sane and is discarded. If a signature message
is received from a neighbor for whom no timestamp is
registered, the timestamp exchange process is initialized.
If the neighbor is registered the signature is checked. If the
signature cannot be verified, the packet is discarded. If the
signature is verified the timestamp is checked. If the
timestamp validates the packet is passed on to the packet
parser within osd*.

D. Interception of outgoing traffic

The plugin also needs to be able to intercept all outgoing
traffic to add signature messages. To be able to do this a new
set of function pointers was added to osd*. They are called
packet transform functions. A plugin can register its own
packet transform functions with osd*, and these functions are
applied to every OLSR packet to be transmitted right before
sending it. It is guaranteed that no changes will be made to a

\[ r = \frac{r_{\text{old}} + r_{\text{new}}}{2} \]
OLSR packet after these functions are called. The function to add such a function pointer is implemented in src/ndef.c: int add_iflfix (**(char *int **));

The plugin registers a function that calculates and adds a signature to the end of all outgoing OLSR packets. To be sure the packet will have room for the signature message (especially when stacking messages) the maximum message size in olid is set to maxmsgsize - sizeof(signature msg).

4.1. Timestamp exchange

The Secure OLSR plugin maintains a repository of registered timestamps. Whenever a node receives a packet containing a valid signature message for which it has a timestamp entry registered the timestamp exchange process is initiated. All timestamp messages are sent as broadcast within regular OLSR packets. The messages use the OLSR message header. All incoming OLSR packets are checked for such messages before the signature check. This way these messages are not discarded even if they arrive as timestamp from a non-registered host. The packets are however passed on for verification even tough they only contain the timestamp exchange message. This is to prevent link hysteresis to consider these packets as lost if the node is verified. A scenario where a host, which the receiving host already has done a timestamp exchange with, can arise if the remote host has restarted and is turned out its timestamp entry for the receiving host.

IV. RELATED WORK

The work [4] has a similar approach for securing the OLSR protocol as described in this paper. As our approach, [4] uses digital signatures for authenticating OLSR messages. Also, they propose a timestamp mechanism against replay attacks, and outline two public key infrastructure systems for MANETs. Even though our approach is similar to [4], there are some important differences.

In [4] they propose to include one signature for each OLSR message, not one for each OLSR packet. In addition to this, one timestamp is provided for each signature. The timestamps are for estimating the freshness of the messages. Thus avoiding replay attacks. The signature is encapsulated and transmitted as an ordinary OLSR message. In contrast to our approach, the signature message in [4] does not need to be in the same packet as the message it is a signature for. This means that the signature and the message can travel in separate packets and separate routes from the originator. Also, the proposed system in [4] is an end-to-end system, whereas our approach is a link-based system.

The timestamp exchange protocol proposed in [4] is a rather complex solution. It relies on a timestamp exchange message being periodically transmitted by each node. The timestamps are not considered fresh until the handshake is complete.

A more thorough discussion of the advantages/disadvantages between the security mechanisms proposed in [4] and our proposed system is left for further study.

V. CONCLUSION AND FURTHER STUDY

We have designed and implemented a security extension for the OLSR protocol. The extension is tested and the correctness of the implementation is verified. However, this is not included in this paper.

Our solution adds extra overhead to all OLSR packets. The extra overhead is relative to the size of the OLSR packets, since the signature size is static. The larger the OLSR packets, the secure plugin will cause less effect on overhead. If using IPv6 addresses, this relative increase in overhead will be even smaller.

The implemented solution uses shared (symmetric) keys for signature creation and verification, and it is assumed that this key is accessible for all (trusted) hosts intended to be part of the MANET. Our secure extension to OLSR does not intend to cover key exchange/management or initial authentication. However, the secure extension is intended to be a part of a larger security scheme that does also cover these aspects. This is left for further study.

REFERENCES


Appendix F

IP Address Autoconfiguration For Mobile Ad-Hoc Networks

*Presented at the Proceedings of IEEE International Conference on Communication (ICC’2004), Paris, June 20-24, 2004*
IP Address Autoconfiguration for Proactive Mobile Ad Hoc Networks

Andreas Tjonnesen, Andreas Hardlund and Paul Engelsdorff

Abstract—A node needs an IP address to participate in a mobile ad hoc network (MANET). If a node lacks a globally unique IP address, it needs to autoconfigure an IP address and check that this address is unique throughout the network before it can join the ad hoc network and participate in routing and regular communication.

This study specifies a mechanism for IP address autoconfiguration in proactive MANETs, i.e., MANETs that are routed with a table-driven routing protocol such as OLSR [3]. With this mechanism, any node can get a valid address, and then participate in the routing and the regular data communication. With proactive routing protocols, each node has a routing table that contains normally a relatively complete list of all nodes already present on the MANET. Thus, the mechanism first ensures that the chosen autoconfigured address is not present in the routing system, before the network is flooded to detect a possible duplicate address. In this way, the chances for address conflicts are reduced drastically before the network is flooded. The proposed solution applies equally well to IPv4 and IPv6.

Index Terms—Mobile Ad Hoc Network, Autoconfiguration, OLSR, Linux

INTRODUCTION AND MOTIVATION

Any node in an IP network needs a valid IP address to be able to participate in the routing and the regular data communication. It is the IP address that uniquely identifies the node in the ad hoc network that the node belongs to.

There are several ways to acquire an IP address. The traditional way was to use static configuration in which the system administrator manually configures a node with a valid address that this node always uses. For a network with many nodes, this is not easily manageable task, and autoconfiguration is a better solution.

Stateful autoconfiguration with the use of DHCP (Dynamic Host Configuration Protocol [4]) is the most common configuration mechanism today. DHCP is based on a centralized DHCP server that leases IP-addresses to clients. The DHCP server ensures that addresses are uniquely assigned to different nodes.

IPv6 aims at removing the need for DHCP infrastructure by introducing stateless autoconfiguration on the link. The node configures an address suffix and ensures that it is unique on the link. Combining the suffix with globally routable prefixes obtained from next-hop routers forms addresses. With the introduction of IPv4 Link Local Addresses [ref stateless autoconfiguration is also an option for IPv4 subnets, although link local addresses are not globally routable, and must be treated similar to private IP addresses.

With both stateful and stateless autoconfiguration, the network prefix is normally manually configured throughout the network comprising different links with different prefixes. Since prefixes are uniquely assigned to different routers, it is enough to ensure that the address suffix is uniquely assigned on the link.

Mobile ad hoc networks (MANETs) [1] calls for special solutions to address configuration. Many nodes have been manually configured an IP address and must rely on address autoconfiguration to be able to participate in the network. In MANETs, autoconfiguration cannot rely on an installed structure of manually configured prefixes of next-hop routers, such as in the wired network case. Indeed, all nodes participate as routers. The node must ensure that the autoconfigured address is unique throughout the whole multihop network, i.e., not only on the link. Moreover, using a centralized DHCP server to assign addresses and enforce address uniqueness is not considered suitable for MANETs. Due to the high dynamics of such network, the solution should not rely on such kind of infrastructure. Instead, a distributed stateless multihop autoconfiguration mechanism is required.

In this paper we propose a scheme for address autoconfiguration that has not any specific server for the allocation of addresses in the network, i.e., it is a fully distributed approach. We have implemented it in an MANET routed with the OLSR routing protocol, and we demonstrate how address autoconfiguration can take advantage of OLSR as the underlying routing protocol.

Routing in mobile ad hoc networks can be divided into different approaches reactive and proactive. A reactive routing protocol does not know anything about the network in beforehand, but must find a route to a given destination on demand. AODV (Ad hoc On-demand Distance Vector) [2] is perhaps the leading on-demand ad hoc routing protocol. A proactive routing protocol seeds to always have a complete up-to-date knowledge of the network topology. This means that the routing protocol always knows which addresses that are in use. OLSR (Optimized Link State Routing) [3] is an example of a proactive ad hoc routing protocol.

Our approach uses the fact that the proactive routing protocol always know which addresses that are in use. On the basis of this knowledge, any node already within the network can assign an address to a new, requesting node. The only thing the assigning node must check is if the address it intends to assign to the requesting node is in use in the routing protocol or the address assigning cache.

This paper is organized as follows. In Section 2 we present related work for autoconfiguration. Section 3 gives an overview of OLSR functionality, whereas Section 4 presents our approach towards autoconfiguration. The system model is
thoroughly described and discussed in Section 5. Section 6 presents implementation details for our work. Finally, Section 7 briefly comments about the security of the protocol, while Section 8 gives our concluding remarks.

II. RELATED WORK

For wireless, ad hoc networks, there have only been many contributions to the subject of address autoconfiguration, although a minority considers address autoconfiguration in proactive MANETs and few— if any— have focused on how to take advantage of the underlying routing protocol such as OLSR.

The work [5] is the most common work on this subject. It describes a mechanism for how mobile nodes can autoconfigure themselves with unique and valid addresses. The method for determining if any other node in the network has taken the same address is taken from the reactive ad hoc routing protocol AODV [2]. The new node assigns itself an address on random, and then broadcasts a query to the ad hoc network asking if this address is already taken or not. Note, for determining which address to use for the new node, this autoconfiguration mechanism does not involve any other node currently inside the ad hoc network. The new node itself generates the address it wants.

The work [6] for the MANETs’ autoconfiguration mechanism is the work mostly related to our proposal. In this mechanism, a node outside the ad hoc network has to request a valid address from a node currently inside the ad hoc network. Any node inside the network can assign an address, and this address must be taken from a range with unsigned addresses. Then the node inside the network will broadcast a query into the network, asking if this address is available, that is, not taken by anyone else.

The work [7] is the most recent work in the subject of autoconfiguration. This work describes mechanisms for Duplicate Address Detection (DAD), but does not specify how to acquire the address in the first place. In [7] they divide DAD into Strong DAD and Weak DAD.

When a new node enters an ad hoc network, it has to use Strong DAD to check if its chosen address is already taken or not. The new node selects two addresses, one temporary and one tentative. Then the node broadcasts an Address Request for checking if the tentative address is already used. If so, any node knowing that the tentative address is already used, answers back to the new node (temporary address) with an Address Response. The new node now has to generate a new tentative address and check again.

The purpose of weak DAD is to detect address duplication during ad hoc routing. A given node inside the ad hoc network has a virtual IP address that is the combination of the current address and a key. When the node receives any control message from the other nodes in the network, the node checks if the address is used by itself, or any other node the current node knows about. The key is used for uniquely identifying the nodes by their virtual address, not only the IP address.

The work in [5] describes basically the same mechanisms. For DAD, any new node entering an ad hoc network must choose two addresses, a temporary one and the actual one. The uniqueness check is based upon sending out a broadcast Address Request, and waiting for an Address Response if the address is already in use.

The DAD mechanism described in [6] is also based upon the same idea, sending out a broadcast test to see if the chosen address is already in use or not. In comparison to [5], and [1], [6] lets the responder do the DAD. In the two other works, the requester will handle the DAD by itself.

Another work that is similar to the mechanisms described in [6] is the study [10]. The proposal describes mechanisms for acquiring an IP address, for strong and weak DAD, and also mechanisms for network merging. The last is done with the use of a network identifier. This is periodically broadcasted into the network. The protocol uses an Address Authority (AA) to maintain state information in the network. The AA caches information such as a network identifier, IP addresses of all nodes, and the associated lifetime of each address. Their autoconfiguration protocol can be integrated with both reactive protocols such as AODV, or proactive protocols such as OLSR. In [1] they present simulations showing that mobile nodes can acquire an IP unique address fast and without wasting resources for overhead. The simulations are based on using AODV as the ad hoc routing protocol.

III. OLSR FUNCTIONALITY

Our work with autoconfiguration for MANETs is based on using OLSR [3] as the ad hoc routing protocol. OLSR is a proactive and table driven routing protocol. All nodes in the network exchange topology information with the other nodes in the network periodically through routing messages. Selected Multipoint Relays (MPR) do forwarding of broadcast messages and routing control traffic. All nodes select the MPRs among their one-hop neighbors, such that for any given node, its two-hop neighbors should be covered by at least one MPR. The MPRs are responsible for forwarding of topology messages in the network, such that all nodes will have a complete view of the network topology. The MPR functionality minimizes the overhead of flooding of routing control traffic.

IV. IPv4 PROACTIVE AUTOCONFIGURATION

IPv4 ProACTIVE AutocOnfig (PAA) is a proposed IPv4 address allocation protocol with duplicate address detection (DAD) for use in MANETs. PAA takes advantage of the fact that nodes that are already members of a MANET domain, running a proactive routing protocol, already has a relatively complete understanding of the topology. The nodes are therefore well suited to allocate IP addresses that have a high probability of currently not being used by any nodes in the network. PAA also does strong DAD to ensure that no other
node is currently configured or in the process of configuring itself, with a duplicate address. This is done by flooding the entire network, and the underlying routing protocol is used for the flooding. PAA provides a node with the ability to automatically configure itself with a unique address and join a MANET without any prior knowledge of the network parameters other than of course that PAA is used in the network.

PAA defines two run-time states: client- and server-mode. An unconfigured node will run the PAA-client to contact already configured nodes, which run the PAA-server. Ideally all nodes in the MANET will run the PAA-server. However, this is not a requirement as long as there is some message flooding mechanism provided by all nodes.

Two different architectural schemes are considered for PAA. It can either be implemented such that a PAA-client broadcasts traffic to all one-hop neighbors. The configuration session is independent of changes in neighborhood. This solution is referred to as the broadcast solution, see Figure 1. Another approach would be for the PAA-client to select only one neighbor to act on behalf of it for the entire session. The chosen neighbor would act as a proxy for the PAA-client, see Figure 2.

The broadcast solution provides full independence and therefore allows a high degree of mobility during the configuration session. However, this solution relies on all traffic being flooded throughout the network since no configured node can unicast messages to reach the PAA-client. The proxy scheme allows for traffic to be sent as unicast back to the proxy since it is already a part of the MANET routing domain.

![Figure 1: Broadcast PAA solution.](image1)

In either case a PAA-client requests an address from a neighbor. A mechanism working with the routing protocol on configured nodes keeps a cache of all addresses processed by the proactive routing protocol for a given time. When receiving a request from a PAA-client, the PAA-server will query this cache. The client can request a preferred address, and if this address is available, it will be returned. If the requested address is not available the PAA-server will generate a random available address and return this to the client. Using the broadcast scheme, the DAD is performed by the client. If the proxy solution is utilized, the neighbor chosen as proxy, will perform DAD on behalf of the client.

![Figure 2: Proxy PAA solution.](image2)

V. Basic System Operation

A. Outline of Mechanisms

PAA consists of three software components. They are outlined in Figure 3.

![Figure 3: Software components of PAA.](image3)

An unconfigured node runs the PAA-client to allocate a "first" address. In this context free means an address not used by any other node in the MANET. The unconfigured node will communicate with one or more of the already configured nodes. A configured node runs the PAA-server that communicates with the routing daemon using interprocess communication (IPC). All communication between unconfigured hosts (PAA clients) and configured PAA-servers is done using the link-local address space 169.254.0.0/16 while all DAD flooding is done using OLSR's MPR flooding in addition to link-local traffic. PAA packets uses the 64bit header illustrated in Figure 2, and all traffic is carried using UDP.

B. Proxy versus Broadcast Solution

The main idea in the implementation of PAA, is that an unconfigured node is to be able to mobile while doing configuration. Because of this no proxy is used for configuration. A node communicates with whatever
configured nodes that are in communication range. Data is
broadcasted from the unconfigured node and sent as unicast
back from the configured nodes. No state is kept in the
configured nodes regarding configuration sessions. This
means that an unconfigured node can be mobile and use
different configured nodes for different parts of the
configuration process.

PAA Packet Formats
The packet formats for PAA is outlined in Figures 4 to 9.

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Seqno</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Size</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message body</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: The generic PAA packet.

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: The Forward Request message.

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Preferred address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: The Address Request message.

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Offered address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: The Address Response message.

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Candidate address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: The Address Test message.

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: The Address Test Confirmation message.

An Example of a Configuration Session
Figure 10 illustrates a configuration session where a new
node wants to join the MANET. In this example no address
collisions are detected.

![Diagram](image)

Figure 10: An example of a conflictless PAA IP allocation
session.

After configuring itself with a random link-local address
the PAA-client broadcasts an Address Request message.
Since the link-local address is generated on random address
conflicts can occur. Because of this an identifier that the
PAA-client generates based on the interfaces MAC address is
used for host identification. The Address Request message
might contain a preferred address, which the unconfigured
host would prefer to use if available. A PAA-server receiving
this request forwards it to the OLSR daemon PAA-plugin,
which checks if the preferred address is available. If the
address is not available or if no preferred address was
received it acquires a random available address. An available
address is an IP address that the node has not heard
mentioned in any OLSR traffic for a given time. All IP
addresses received in OLSR control traffic is kept in a IP-
cache for this interval. The allocated address is then sent back
unicast to the requesting host on an Address response
message.

The requester is now to perform DAD. This is done by
broadcasting an Address Test message link-local. All PAA-
servers receiving this message is to flood it through the
MANET using MPR flooding. This means that the message is
to be encapsulated in an OLSR message header and sent as a
regular OLSR message. Any OLSR daemon extended with a
PAA-plugin receiving an Address Test message is to forward
this link-local (in addition to regular MPR forwarding), if and
only if the node has received a Forward Request within a
given time interval. Any node in the process of configuration
itself broadcasts such Forward Request messages regularly to
make configured nodes within transmission range forward all
received Address Test messages link-local.

If a node receives an Address Test message containing the
address the node has been offered it is considered an address
conflict because some other node (the sender of the Address
Test message) is in the process of configuring itself with the
same address the local node was offered. The node will then
restart the configuration process by sending a new Address
Request.

To make sure the Address Test message is injected into the
MANET, the PAA-server must confirm that it has received
the message from the requester. If a requester does not receive
such a confirmation message the emitted Address Test message is not considered flooded and another Address Test message is broadcast immediately. A register is to flood two Address Test messages, and be idle for 5 seconds after each successful transmission. If no conflicting Address Test message is received within 5 seconds after the transmission of the second successful Address Test flood, the address is considered valid and strong DAD is complete. The PAA-client then configures the communication interface with the offered address, starts the OLSR daemon and goes into PAA-server mode.

V. IMPLEMENTATION DETAILS

The ProActive Autoconfig solution is implemented for Debian [ref obriop]. Parts of the protocol functionality is currently applied for patenting by Theaker Communications AS and therefore the source code is not publicly available as of yet. In the following sections we will look into the PAA server, client and plugin in detail. This will also be a look at implementation issues related to the existing GNU/Linux implementation.

APAA Client

When an unconfigured node starts the PAA service, it is started in client mode. The PAA client has these main responsibilities:

- Configure a virtual interface with a random link-local address.
- Generate an ID to use in PAA communication.
- Querying neighbors for an address offer using link-local broadcast.
- Perform DAD on the offered address.
- Configure the local node with the fake address (or one of no unique addresses could be found).
- Start the routing protocol.
- Go to server operation mode.

A flow diagram describing the PAA-client operation is illustrated in Figure 11.

The first thing the client does is to generate a random link-local address. It will then configure an "alias" interface with this address. Such alias interfaces is the way one can configure an interface with multiple IPv4 address on GNU/Linux systems. If PAA is set to run on eth0, the alias interface would be eth0:0. This interface will be used by both the PAA-client and later the PAA-server.

The PAA-client must periodically broadcast a Forward Request message link-local to let already configured neighbors know that they should forward all received PAA control messages on the link-local broadcast address.

Figure 11: Flow diagram of the PAA-client.

The client must generate an identifier that it will use to identify itself in PAA traffic (since link-local address conflicts might occur). In the implementation this ID is the lower 32-bits of the MAC address of the interface on which PAA runs. This diminishes the uniqueness of the ID, but the choice of two WLAN interfaces using the same last 32-bits in their MAC addresses are very small. The first six bytes of the MAC address are the "Organizational Unique Identifier" while the lower six bytes are the actual factory serial number. One can however still risk a ID clash if two interfaces by different makes that share the last 4 bytes in their Organizational ID with the same serial number were to meet. But in general there is a much smaller chance for this happening than the upper 32-bits matching which would only require having two interfaces bought from the same stock in many cases.

To get a free IPv4 address an Address Request message is broadcasted link-local. A node signs this request with its ID. Any neighbor that is already a configured member of the MANET routing domain and out of quarantine time as explained later, will answer with an offered address if an address could be allocated. If no replies are received, the PAA-client can optionally configure its interface with a random address within a predefined address space (currently 192.168.60.0/16) and start the routing daemon, thus starting its own MANET.

Upon receiving the first Address Response (with the correct ID) the requester will generate an Address Test message and broadcast it link-local to all neighbors. Any other Address Response message received for the same request (carrying the same sequence number) is silently discarded.
Any configured node that receives the Address Test message sends an Address TestResponse message so that the PAA-client can be sure that the test-message is flooded. If no Address Response is received or no Address Test message is received, this is because it is crucial for the non-configured node to be sure that the test-message is received by at least one configured node. If not, no DAD will be done.

The PAA-client then waits for a given interval to receive a possibly conflicting Address Test messages sent by another node. If this interval times out without any conflicting Address Test messages received, another Address Test message is broadcasted. If another time interval passes without the node receiving any conflicting Address Test messages the address is considered valid and DAD is considered complete. If a conflicting message is received the configuration process is restarted.

When a unique IP is allocated the PAA-client will configure the interface specified with the address. Forward Request messages will no longer be sent. When the interface is configured the routing daemon, in this case client, is started. The PAA-client must take care to ensure that the routing daemon will run on the chosen interface.

When the routing daemon is started the PAA-client will put itself in server mode.

II. PAA Server

The PAA server has these main responsibilities:

- Connect to the local routing daemon (plugin).
- Listen for Forward Request messages and maintain a forward timer.
- Listen for Address Request messages.
- Query the routing protocol (plugin) for free addresses.
- Send Address Response messages to PAA-clients that have issued Address Request messages.
- Forward Address Test messages received from PAA-clients to the routing protocol.
- Forward Address Test messages received from the routing protocol to PAA-clients if the forward timer is not set.

The flow diagram of the PAA server is illustrated in Figure 12.

![Figure 12: Flow diagram of the PAA server.](image)

The first thing PAA does when going from client- to server-mode is to connect to the routing daemon via a TCP socket to the hoophold device. A PAA-server should only forward messages link-local if there exists any unconfigured hop neighbors. To detect this, the PAA-server listens for Forward Request messages, which are sent periodically by unconfigured nodes.

Upon receiving a Forward Request from a PAA-client the PAA-server queries the routing protocol for a free address. In the implementation this means querying the oler PAA-plugin via IPC. If the Address Request contains a preferred address this is passed to the plugin in the query message.

If no free IP addresses can be allocated, the PAA-server will transmit an Address Response message with the FAILURE flag set. If an IP offer is received from the routing protocol a Address Response message is generated containing the offered IP, the ID and SEQNO from the received Address Request.

Upon receiving an Address Test message on the link-local interface, the PAA-server forwards this message to the routing protocol if and only if the FORWARDED flag is not set in the message. This is to prevent forwarding of messages forwarded link-local from other PAA-servers, which can lead to loops. The PAA-server then sends an Address test Response message back to the PAA-client.

Upon receiving an Address Test message from the routing protocol the PAA-server broadcasts this message link-local (setting the FORWARDED flag) if and only if the forward timer is not expired.

III. PAA OLSR-plugin

The PAA OLSR-plugin has these main responsibilities:

- Make sure it does not serve any IP address before it has run for a certain amount of time (quarantine).
- Caching all heard-of IP addresses for a given period of time.
- Listen for Address Request messages (from the PAA-server).
- Process Address Request messages by trying allocate free IP addresses.
- Flood the network with Address Test messages received from the PAA-server.
- Forward Address Test messages received in OLSR messages to the PAA-server.

A proactive routing protocol such as OLSR will eventually have heard of close to all IP addresses currently used in the MANET. But rather than checking for free IP addresses by looking up all internal tables of the routing table, an IP cache "pool" is maintained. This way IP addresses can also be cached for a longer periods than they would stay in the routing daemon internal tables.

Upon receiving any kind of known routing control traffic, the plugin adds all the IP addresses listed in the message to the IP cache with a given timeout. This period is set to 30 seconds in the implementation. It also goes for PAA traffic. The address contained in Address Test messages are updated in the IP cache as well as addresses offered by this
node as response to Address Request messages from the PAA-server.

![Image](image.png)

**Figure 13: The PAA-plugin.**

To locate a free address the routing daemon selects a random address in the net-range that the MANET uses, or it uses a possible preferred address provided by the PAA-server, and checks this against the IP cache. If the address exists a random address is created within the MANETs net-range and checked against the IP cache again. This is done for a predefined maximum of times. If a generated address is not found in the cache it is considered free and will be sent to the PAA-server that will offer it to the PAA-client that again will perform DAD on the address.

Upon receiving an Address Test message from the PAA-server the routing daemon encapsulates the Address Test message in a routing protocol message. This message is then flooded throughout the MANET by the flooding mechanism provided by the routing protocol. In the implementation this means that the Address Test message is encapsulated in a regular OLSR message packet and flooded using the MPR scheme.

Upon receiving an encapsulated Address Test message carried by the routing protocol, the routing daemon decapsulates the packet and forwards it to the PAA-server in addition to forwarding the encapsulated message according to the forwarding algorithm used by the routing protocol.

The role of the PAA-plugin is illustrated in Figure 13.

D.

VII. SECURITY CONSIDERATIONS

PAA is not in any way responsible for authentication of nodes. To have some sort of access control one must apply an outer layer of security mechanisms. PAA could be used in combination with a scheme like the signature solution proposed in [Secure OLSR ref]. However, this solution does not deal with authentication either. Some distributed authentication system can be imagined, but physical distributing through e.g. smart cards can be sufficient in many scenarios.

VIII. CONCLUSION

We have implemented and tested a solution for autoconfiguration of mobile nodes in a mobile ad hoc network running a pro-active routing protocol. We have chosen OLSR as our routing protocol to base our implementation upon. However, the mechanisms behind the autoconfiguration are independent of which pro-active routing protocol we chose for the MANET.

ACKNOWLEDGMENT

We would like to thank Rour Bjorgan Rovik and Jon Anderson for valuable and fruitful discussion regarding our autoconfiguration mechanisms and implementation.

REFERENCES

Appendix G

Internet Connectivity for Multi-Homed Proactive Ad Hoc Networks

To be presented at the The First IEEE International Conference on Sensor and Ad hoc Communications and Networks, October 4-7 2004
Internet Connectivity for Multi-Homed Proactive Ad Hoc Networks

Paul E. Engelsstad  
TeleNor R&D  
Fonebu, Norway  
paul@telenor.com

Andreas Tornesen  
Thales Communications  
Oslo, Norway  
andreas@telenor.com

Andreas Hafslund  
University of Oslo (Unik)  
Kjeller, Norway  
andreas@unik.no

Geir Egeland  
TeleNor R&D  
Fonebu, Norway  
geir.egeland@teleenor.com

Abstract—A prerequisite for a widespread and successful deployment of proactive ad-hoc networking technology is its ability to provide easy access to the Internet. Normally, proactive routing protocols provide routing messages that establish default routes to ensure connectivity for outgoing IPv4 packets destined for the Internet. However, mechanisms to ensure connectivity for incoming IPv4 packets from the Internet are yet poorly documented in published material. Possible solutions include implementing a modified Mobile IPv4 Foreign Agent (MIP-FA) or Network Address Translation (NAT) on each Internet Gateway node in the ad hoc network. In this paper we discuss different strategies for providing Internet access to proactive ad hoc networks. We also describe problems experienced in our lab test-bed with default routes under the condition of site multi-homing. Based on this experience, we propose working solutions for Internet access from proactive ad hoc networks.

Keywords - manet; Internet; proactive; multi-homing; NAT.

I. INTRODUCTION

IPv4-based applications, such as web browsing, e-mail, telnet, and ftp, mainly communicate with servers or peers over the Internet. A mobile ad-hoc network (MANET [1]) has no fixed infrastructure, and services on the Internet might not be available in such networks. A likely scenario is that nodes on an ad-hoc network in some cases also want to connect to nodes on the Internet, using services available there. For a widespread and successful deployment of MANETs, the ability to provide easy access to the Internet is therefore a prerequisite.

A common approach is to let a MANET node with Internet access operate as an Internet gateway and provide Internet access to other nodes in the MANET. There can naturally be several MANET nodes operating as gateways on the MANET at the same time.

In this paper we address the lack of a good mechanism for IPv4 Internet access in a proactive MANET, i.e., a MANET that is routed with a table-driven routing protocol, such as the Optimized Link State Routing (OLSR) [2] protocol or the Topology Dissemination Based on Reverse-Path Forwarding (TBRPF) [3] routing protocol.

Proactive routing protocols normally allow gateways to use special routing messages to set up default routes on the MANET. IPv4 packets that do not have an IPv4 destination address known locally on the MANET are forwarded along the default route out of the MANET through the gateway.

Normally, the destined external host on the Internet will send return traffic to the source IP address of the outgoing packet. Thus, for IPv6, a MANET node configures an address under a global prefix managed by one of the MANET gateways [4] and uses this address as source IP address when communicating with external hosts on the Internet. Return traffic from the external nodes on the Internet is therefore routed back to the gateway, which in turn can forward the packets to the MANET node.

However, for IPv4, which is the focus of this paper, there is a great scarcity of global IPv4 addresses. Thus, the gateway may be equipped with a very limited number of external IPv4 addresses. To allow different MANET node to share an address for external communication, the gateway may implement a Network Address Translator (NAT) [5] or a Mobile IP Foreign Agent (MIP-FA) [6]. Both solutions have been proposed and have been implemented for MANET gateways [7-9].

Since MANETs normally are without infrastructure and have limited capabilities for internal coordination, it is difficult to control which of the MANET nodes operate as gateways and which mechanism (e.g., NAT or MIP-FA) each gateway implements. Thus, there might be both NAT-based and MIP-FA based gateways present on the MANET simultaneously. This issue has not been studied in previous work that proposes the use of MIP-FA gateways for proactive MANETs [7-8].

This paper explores problems with Internet access for a multi-homed MANET, i.e., a MANET where more than one MANET node operate as gateways, and where gateways may use different gateway technologies. Much of the problem is caused by the use of default routes that proactive protocols use to get packets routed out of the MANET.

Based on experience from our test-bed, we propose solutions for multi-homed MANETs, which work well both for NAT-based and MIP-FA-based gateways. Although our main focus is on the OLSR routing protocol, the analyses and proposed solutions are also applicable to other proactive routing protocols such as TBRPF.

Section II presents in depth solutions for Internet connectivity on multi-home MANETs. In Section III we use a test-bed experiment to show deficiencies of using default routes in a multi-homed network with NAT-based gateways. Instead, we propose a new working solution for Internet connectivity. The relevance of our experimental results to MIP-FA based gateways is discussed in Section IV. Here, we propose a working solution also for this scenario. Section V shows how to achieve dynamic change of NAT-based gateways, and Section VI presents experimental results on performance constraints with mobility and Internet access. Conclusions are drawn in Section VII.
II. BACKGROUND

A. Proactive ("table-driven") routing protocols

With proactive protocols, the network is periodically flooded with route information, so that the routing tables contain complete information of routes to all nodes present on the MANET. The most widely studied and popular proposals include the OLSR and the TBRPF protocols. Both protocols introduce optimizations of the classical link state algorithm tailored to the requirements of a mobile wireless LAN. Both provide shortest path routes in terms of number of hops.

OLSR uses multipoint relays (MPRs), which are selected nodes that forward broadcast messages during the flooding process. Thus, the message overhead of the flooding process is reduced substantially compared to a classical link state algorithm. Furthermore, the number of control messages flooded in the network is minimized, since it is only nodes elected as MPRs that generate link state information. An MPR node may also choose to report only links between itself and its MPR selectors, allowing partial link state information to be distributed in the network. The protocol is particularly suitable for large and dense networks as the technique of MPRs works well in this context.

TBRPF is based on source trees and reverse path forwarding. Each node running TBRPF computes a source tree, based on partial topology information stored in its topology table using a modification of Dijkstra's algorithm. The tree provides paths to all reachable nodes on the MANET. To minimize overhead, each node reports only part of its source tree to neighbors. TBRPF uses a combination of periodic and differential updates to keep all neighbors informed of the reported part of its source tree. Each node also has the option to report additional topology information (up to the full topology) to provide improved robustness in highly mobile networks.

B. Acquiring Internet connectivity for outgoing traffic

Both OLSR and TBRPF allow a MANET router to advertise prefixes that are topologically correct for the external networks to which they are connected. Network prefixes are advertised in Host and Network Association (HNA) messages for OLSR or in Network Prefix Association (NPA) messages for TBRPF. The message binds a set of network prefixes to the IP address (OLSR) or Router ID (TBRPF) of the node attached to the external networks. Each message contains one or many network prefixes, each specified by a network (OLSR) or a prefix length (TBRPF).

The messages are transmitted periodically and the information expires after a specified time. Each node also maintains information specifying which nodes may act as gateways for associated hosts and networks by binding a gateway address to the prefix of the external network. Upon reception of a message, a node updates the routing table with the prefix information contained in the message, before flooding it further throughout the MANET.

Hence, both OLSR and TBRPF use default routes to announce reachability to the Internet. A MANET node that has Internet access over an external network to which it is connected, operates as a gateway and advertises Internet connectivity as a 0.0.0.0 default route. All packets destined for addresses without a route on the local MANET will be routed out along the default route to the gateway and forwarded further onto the Internet.

C. Acquiring Internet connectivity for incoming traffic

In addition to using default routes for outgoing packets, a mechanism is required to ensure that return traffic from the Internet gets routed back to the MANET.

If the gateway implements a Mobile IPv4 Foreign Agent (MIPv4-FA) and if the MANET node runs Mobile IPv4, the MANET node may register the care-of-address of the gateway with the Home Agent (HA) [6]. When connecting an external host on the Internet, it uses its home address as source IP address. The return traffic is therefore routed to the HA, which encapsulates the packets and tunnels them to the care-of-address of the MIP-FA gateway. The gateway can then inject the return traffic into the proactive MANET.

A gateway that on the other hand implements Network Address Translation (NAT) will translate the source IP address of outgoing packets from the MANET node. It replaces the source IP address with an address of the NAT gateway, which is routable on the external network. Hence, an external host will return packets using the IP address of the NAT gateway as destination IP address. The gateway can then replace the destination IP address with the IP address of the MANET node, and inject the return traffic into the MANET.

A drawback with a MIP-FA solution is that it requires changes to the Mobile IP implementation on both the Mobile Node (MN) side (i.e., on the source node requiring Internet access) and on the Foreign Agent (FA) side (i.e., on the gateway). Since the MN and the FA are no longer on-link, both sides will have to deal with Agent Solicitations and Agent Advertisements in a different way; TTL values and IP destination addresses must be set differently; ARP must be used differently and MAC addresses are no longer relevant for communication between MN and FA. The solution differs so much from regular Mobile IP operation, that a special purpose "MIPv4-for-MANET" code is probably required (i.e., it is not beneficial to reuse the regular MIPv4 code).

Moreover, independent coexisting implementations of both the MANET routing protocol and the "MIPv4-for-MANET" code are not trivially manageable, since both implementations will make unsynchronized modifications to the routing table.

Another drawback that limits the applicability of a MIP-FA based solution is that it requires that the care-of-address of the gateway and the home IP address of MN are globally routable. However, since the IPv4 address space is a scarce resource, nodes that require Internet access might only be able to acquire private IPv4 addresses.

A NAT-based mechanism, on the other hand, appears as an alternate solution. The NAT functionality may be in the form of Basic NAT, however NATP (i.e., NAT with port translation) is a more applicable solution, since many MANET gateways might only be able to acquire a single IP-address on the external network to which they are connected [5]. NATs allow other MANET nodes to use private addresses. NAT-devices
can be nested, and this solution will work even when the MANET gateway acquires a private IP address from the external network.

III. INTERNET CONNECTIVITY USING NATs

A. UDP traffic and default routes

We tested site multi-homing in a proactive MANET with two NAT-based gateways present. The source node (SN) communicating with an external host (XH) over the Internet. All external communications pass through an Intermediate Node (IN1 and IN2), respectively. The test configuration is illustrated in Figure 1.

![Figure 1](image1)

The OLSR-Unix implementation for Linux [10] was used as routing modules on all MANET nodes (4a SN, IN1, IN2, GW1 and GW2), and WLAN 802.11b was used for the wireless communication. All nodes were located in the same room (50x50 meters), and the IP-tables feature of Linux was used to emulate the SN was not in direct radio-range with the gateways.

We added mobility to the network by allowing the SN to alternate between being within radio range of IN1 and being within radio range of IN2, on 20 seconds time intervals. That is, the SN was first within radio range of IN1 and outside radio range of IN2 for 20 seconds. Then, after 20 seconds, the SN got within radio range of IN2 and outside radio range of IN1, and so forth. Before we started measurements, the network was granted a transient period of some minutes to ensure convergence of the routing system with respect to the non-mobile nodes. After the transient period, the SN started sending UDP packets of 512 bytes at a transmission rate of 204 kbps destined for the XH.

The HNA messages from GW1 and GW2 form default routes to the Internet on a shortest-hop basis. Hence, while connected to IN1, the SN uses IN1 as the default route to the Internet, which in turn uses GW1 as next hops of the default route. However, as soon as it connects to IN2, the default route is recalculated, and the SN uses IN2 as the default route to the Internet, which in turn uses GW2 as next hops of the default route.

The experimental results are shown in Figure 2. The UDP traffic alternates between GW1 and GW2. However, there is an intermediate period after the SN has switched between IN1 and IN2 where all traffic is dropped, before the packets again get correctly routed out through one of the gateways. The figure shows that with the default hello interval of OLSR of 2 seconds, it takes 5 seconds (4a 3 hello packets) after the SN has moved from IN1 to IN2 to discover that the route to IN1 is no longer valid.

![Figure 2](image2)

Thus, the SN may use maximum 6 seconds before it recalculates the HNA route for the gateway and establishes IN2 as the next hop for the outgoing route. Apart from the fact that there is a required time for the routing to converge, everything works fine with UDP traffic.

B. TCP traffic and default routes

We did a similar test. This time, however, the SN established a TCP session with the XH. In this case, we experienced as expected that a TCP session would always break as the default route was shifted to a different gateway. This experimental result is illustrated in Figure 3.

The reason the TCP connection breaks can be explained as follows: The first packets were all routed to the NAT module of one of the gateways, say GW1, corresponding to the intermediate node, IN1, that the SN was initially connected to. Source IP address of all these packets were translated by the NAT module of the gateway.

However, when the SN eventually established connectivity with IN2 and discovered that connectivity with IN1 was lost, TCP packets were forwarded via GW2. When the outgoing TCP packets passed out through the NAT-module of GW2, the module naturally translated the source IP address of the TCP packets to a different address than the one used by the NAT-module at GW1. The packets were not recognized when they finally arrived at the XH, since TCP uses also the source address to identify the connection. Upon reception of the first
"misrouted" packet, the XH immediately returned a TCP-RESET message to the unknown source address (i.e., it was routed back to the SN through the NAT module of GW2), and the TCP session broke as shown in the figure.

![Figure 3. A TCP session may break when there are two NAT-based gateways and when default routes are being used.](image)

C. Working solution using explicit tunneling

To avoid that TCP sessions break, we propose to use explicit tunneling to one of the gateways instead of using default routes (Figure 4).

![Figure 4. Tunneling of packets for external host via gateway, e.g., using IP-in-IP encapsulation.](image)

The SN knows that a host is external if it cannot find the destination IP address in the routing table. Instead of using the default route, the SN simply tunnels the IP packets (e.g., by IP-in-IP encapsulation [11] or minimal IP encapsulation [12]) to the IP address of the gateway, which is found in the Originator ID field of HNA messages. No changes to the OLSR specification are required. However, gateways must be able to accept and decapsulate tunnelled packets.

Before sending the first packet to the XH, the SN chooses an appropriate gateway (e.g., the gateway closest to the SN). The SN consequently uses the same gateway for subsequent packets belonging to the same communication session, as long as it has a route to the gateway.

We repeated the same experiment with the tunneling solution. The experimental results are shown in Figure 5.

![Figure 5. Testbed experimental results with explicit tunneling to a NAT-based gateway. The return traffic in terms of TCP ACK packets is not shown, since the amount of this traffic is small as indicated in Figure 3.](image)

Figure 5 shows that with explicit tunneling all packets are consistently routed throughout the same gateway even if the SN is mobile, and hence the TCP session does not break.

The figure also shows that the throughput was approximately 30% lower when packets are routed through the gateway that is one hop further away from the SN. This is as expected, since the packets have to be transmitted by an additional node. Further performance results from our test-bed are detailed in a follow-on paper [13].

IV. INTERNET CONNECTIVITY USING MOBILE IP

A. Problems with different coexisting gateway technologies

Bernard et al. [7-8] propose a solution for OLSR with MIPFA based gateways. For outgoing connectivity, the solution relies on the default routes established by the HNA routing messages.

A problem with the use of default routes is the same as before: the SN cannot control through which gateway an outbound packet is routed. As demonstrated by our test-bed experiments presented above, the default route may change as the SN moves around on the MANET, and packets may be routed out through different gateways correspondingly.

The use of Mobile IP easily leads to triangle routing. In a MANET this means that outbound packets may go out through one gateway (e.g., the closest one) while return packets enters the MANET via the MIPFA gateway to which the MANET node is registered. However, it is likely that this gateway is behind a firewall. Most firewalls today are stateful: the first packet going out of the access network sets soft-state in the firewall and return packets are temporarily allowed to enter the access network. Thus, if the first outbound packet (e.g., TCP-SYN) does not exit the MIPFA gateway, the return packet (e.g., TCP-SYN+ACK) will be stopped by the firewall.

Furthermore, in a multi-homing scenario, some gateways may be NAT-based, others MIP-FA based, and others based on
other technical solutions. This generates a problem for the solution proposed by Benzaied et al. [78], because one gateway technology may undermine correct functionality of another gateway technology.

For example, if there are one MIP-FA based gateway and one NAT-based gateway present on the MANET, outgoing traffic from a MIP-enabled SN may easily be mistakenly routed along a default route out through the NAT-based gateway. The NAT will translate the source IP address and forward the packet to the external node. Hence, the IP packet will not be recognized and the communication session (e.g., TCP session) is likely to break, similar to what we demonstrated above (e.g., in Figure 3).

Alternatively, the NAT may drop the packet if it only accepts tunneled packets according to the explicit tunneling solution proposed above. In either case, the TCP session will eventually break.

B. Problems with ingress filtering

Ingress filtering [14] of the outgoing traffic is becoming more and more common on routers in access networks. Ingress filtering means that a router will not accept on its ingress interface packets with a source IP address that is not topologically correct for that interface. The motivation is to prevent IP address spoofing.

It is quite probable that many MANET gateways will desire to implement ingress filtering. Thus, if the outgoing packet ends up at another gateway than the MIP-FA based gateway that the SN is registered with, the gateway may drop the packet to prevent a possible Denial-of-Service attack.

Ingress filtering might also be implemented on routers further back in the access network to which the MANET gateway is connected. Since the SN uses its own home IP address as source address of outgoing packets, and since this address would normally not be probably topologically correct on the access network, SN's packets are easily filtered away. Ingress filtering is another issue that the MIP-FA solution by Benzaied et al. [78] did not cover.

MIP-4 Reverse Tunneling [15] can be used as an antidote against this. The mobile node requests reverse tunneling service from the FA when it registers with it. The FA tunnels the mobile node's traffic back to the HA, using a topologically correct IP address as source address of the encapsulating IP header. The HA will decapsulate received packets and forward them to the SN.

It is only the MIP-4/NA-based gateway with which the SN is registered that will offer reverse tunneling for the SN, and the HA will not accept packets tunneled from FAs that the SN has not registered with. As a consequence, it is utmost important that all outgoing traffic is sent over the MIP-4/NA based gateway with which the SN is registered.

C. Working solution using explicit tunneling

To avoid that the TCP sessions break, that packets are dropped by the gateway, or that packets are subject to ingress filtering, we propose to use explicit tunneling to one of the gateways instead of using default routes, also for the MIP-FA based solution (Figure 6).

If the SN uses MIP-4 Reverse tunneling, the explicit tunneling solution coincides with the Encapsulating Delivery Style of Reverse Tunneling [15] (Figure 6).

V. INTERNET CONNECTIVITY USING MOBILE IPv6 OVER NAT-BASED GATEWAYS

A. Problem with NATs and dynamic change of gateways

A SN can easily change between two MIP-FA-based gateways without breaking the communication session. The SN simply registers the new MIP-FA with its own HA. With a NAT-based gateway, on the contrary, the session of a SN is bound to the NAT that the session passes through. As the SN moves around on the MANET, it may move close to another gateway. In this situation, the SN might desire to change gateway dynamically without breaking the communication session.

B. Working solution using NAT traversal

Mobile IP with NAT traversal [16] can be used to enable dynamic change of NAT-based gateways. The NAT traversal is based on IP-in-UDP-in-IP encapsulation, using the Mobile IP Home Agent UDP port 434 for encapsulated data traffic.

The SN must implement a MIP-4 client and uses an extension in its Registration Request to indicate that it is able to use Mobile IP UDP tunneling instead of standard Mobile IP tunneling. If the HA sees that the Registration Request seems to have passed through a NAT, it sends a successful registration reply. The home agent will then use UDP tunneling to the SN, using the same UDP ports and IP addresses that appeared in the registration request, but in reverse order. The SN use MIP UDP to tunnel packets back to the SN, using the same ports and IP addresses as in the original registration request message. The source port may vary between new registrations (e.g. when the SN changes to a new NAT-based gateway), but remains the same for all tunneled data and for reregistrations. In addition, the SN may periodically send polling messages to the HA to keep the soft state mapping in the NAT valid, which otherwise normally would time out within a couple of minutes.
Since the MANET can be multi-homed, this solution requires an additional IP-in-IP tunneling from the SN to the gateway, to ensure that the source IP address - and possibly also the UDP source port number - are translated consistently by the same NAT-based gateway, so that the traffic will be recognized by the HA (Figure 7).

A clear advantage of this approach over using the regular MIPv4 code (enabled for NAT traversal) is that it also uses the Internet. The SN is not required to implement special "MIPv4-for-MANET" code. The solution allows for dynamic change of gateways between NAT-based gateways and MIPv4-based gateways. (However, an additional special "MIPv4-for-MANET" code is probably required if the SN shall also be able to use MIPv4-based gateways.)

A disadvantage is that the solution requires an additional IP and UDP header's worth of tunneling overhead (i.e., 28 bytes) between the SN and the gateway. However, this can be a small price to pay compared to the bandwidth penalty of not routing packets out through the closest gateway, as shown in Figure 5 of Section III.C.

There are also ongoing efforts using header compression methods to reduce the tunneling overhead of MIPv4 NAT traversal [17].

VI. CONCLUDING REMARKS

Solutions for Internet connectivity must be designed for the possibility that there might be multiple gateways present on the MANET, and the gateways may implement different gateway solutions, including NAT and MIPv4.

In this paper we have shown that these solutions do not work well in the presence of default routes used by OLSR and TRRP. With default routes, outgoing traffic might be directed to other gateways in a non-deterministic way, depending on dynamics and mobility in the network. The gateway solutions, however, require that packets be routed consistently over a specific gateway.

We have proposed to replace default routes with explicit tunneling between the SN and the gateway. This requires that the SN explicitly discovers available gateways, e.g., by means of the HNA messages of OLSR. Ideally, the SN would select a gateway based on its capabilities (which might, for example, be discovered through a new extension to the HNA messages) and on how far away from the SN it is located.

When the SN wants to send a packet that is destined for an IP address not present in the routing table, it tunnels the packet to the selected gateway. All packets belonging to the same session should normally be tunneled to the same gateway.

We also proposed a solution where source nodes may use Mobile IPv4 with NAT-traversal to change host-to-host gateways dynamically. The solution requires 28 bytes of extra overhead in each packet, but - as shown in Section III.C - this can be a small price to pay compared to the bandwidth benefits of getting traffic routed through the closest gateway, even when the SN is a NAT.

A drawback of the tunneling of outgoing packets is that it requires additional overhead in each packet, e.g., IP-in-IP encapsulation would require an additional 20 bytes of overhead in every outgoing packet. The fraction of additional overhead will be relatively high for certain types of traffic that require short packages, such as Voice-over-IP. To save overhead, one may use minimal IP encapsulation [2] or header compression mechanisms [16, 18].

The analyses in this paper are also relevant for the provision of Internet connectivity in IPv6-based MANETs [4]. The use of default routes might be less damaging for IPv6, since NATs are not anticipated here. As we demonstrated above, only one single packet that was mirrored through a NAT was enough to close down the entire TCP session. However, the problems with ingress filtering and with stateful firewalls, as described in Section V, are still present in an IPv6 setting.

For IPv6 one may replace IP-in-IP tunneling with a routing header to source-route outbound packets via the selected gateway [4]. To reduce the overhead even further, one may use multiproto tunneling by using source address based default routing [19]. The latter would not be possible with an IPv4-based MANET, since the source IPv4 address (i.e., the home address with MIPv4-based gateway or any of the IP-in-IP tunnel's local address with a NAT-based gateway) would not correspond to a prefix of the gateway.

The source code of the NAT-based gateway solution with explicit tunneling proposed in this paper can be downloaded from [10].