UPPSALA
UNIVERSITET

Uppsala Master's Thesis in Computer Science
Examensarbete DV3, 1DT150
December 18, 2002

Porting AODV-UU Implementation to ns-2 and Enabling Trace-based Simulation

Björn Wiberg
bjwi7937@student.uu.se

Information Technology
Department of Computer Systems
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

Abstract

In mobile ad-hoc networks, autonomous nodes with wireless communication equipment form a network without any pre-existing infrastructure. The functionality of these networks heavily depends on so called ad-hoc routing protocols for determining valid routes between nodes.

The main goal of this master's thesis has been to port AODV-UU, a Linux implementation of the AODV (Ad hoc On-Demand Distance Vector) routing protocol, to run in the ns-2 network simulator. Automated source code extraction lets the ported version use the same source code as the conventional Linux version, to the extent this is possible.

The second goal was to use logfiles from the APE testbed, a Linux-based ad-hoc protocol evaluation testbed, for enabling trace-based simulations with AODV-UU. Results of trace-based simulations show that it is possible to obtain packet delivery ratios closely corresponding to those of real-world experiments.

Supervisor: Henrik Lundgren
Examiner: Christian Tschudin

Passed:

Acknowledgements

I would like to thank my supervisor, *Henrik Lundgren*, for setting up and organizing this master's thesis project. Without his help, this project would probably not have been carried out at all. I am very grateful for all the support that I have received.

Also, I would like to thank *Erik Nordström*, the author of AODV-UU, for his highly appreciated help during this project. Not only did he assist me and make me part of the development team during the initial stages of porting AODV-UU to the network simulator; he has also contributed with many good ideas that made the porting process easier, and helped me with implementation questions that popped up from time to time. *Stefan Lindström*, a very good friend of mine – also working on his master's thesis project – has helped me with C and C++ issues during the many hours of implementation and testing. Thanks, Stefan.

Finally, I would like to thank my examiner, *Christian Tschudin*, who kindly agreed to take on the examination role right from the beginning of this project, and all the other members of the CoRe (Communications Research) group at the Department of Computer Systems (DoCS) at Uppsala University, Sweden, for their pleasant company during the entire project.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Proposed goals . . . 1
1.3 Accomplishments . . . 2
1.4 Document overview . . . 2

2 Ad-hoc Networks **3**
2.1 Introduction . . . 3
2.2 Properties of ad-hoc networks . . . 3
2.3 Comparisons with wired networks . . . 4
2.3.1 Infrastructure . . . 4
2.3.2 Addressing . . . 4
2.3.3 Routing . . . 4
2.4 Ad-hoc networking issues . . . 5
2.4.1 Wireless medium access . . . 5
2.4.2 Addressing . . . 5
2.4.3 Network security . . . 6
2.5 Applications of ad-hoc networks . . . 6
2.6 Conclusions . . . 7

3 Ad-hoc Routing Protocols **8**
3.1 Introduction . . . 8
3.2 The need for ad-hoc routing protocols . . . 8
3.3 Classification of ad-hoc routing protocols . . . 8
3.4 Properties of ad-hoc routing protocols . . . 9
3.5 The IETF MANET working group . . . 10
3.6 Examples of ad-hoc routing protocols . . . 10
3.6.1 DSDV . . . 10
3.6.2 OLSR . . . 11
3.6.3 AODV . . . 12
3.6.4 DSR . . . 13
3.6.5 TORA . . . 14
3.7 Conclusions . . . 15

4 AODV-UU **16**
4.1 Introduction . . . 16
4.2 Installation and usage . . . 16
4.3 Configuration . . . 17
4.4 Interaction with IP . . . 17
4.4.1 The Netfilter framework . . . 18
4.4.2 Performance . . . 18

4.5 Software modules . . . 19
4.5.1 Non-kernel-related modules . . . 19
4.5.2 Kernel-related modules . . . 24
4.6 Packet handling . . . 25
4.6.1 Packet arrival . . . 25
4.6.2 Initial packet processing . . . 26
4.6.3 Data packet processing . . . 26
4.6.4 AODV control message processing . . . 26
4.6.5 AODV control message sending . . . 26
4.7 Current status . . . 26

5 The ns-2 Network Simulator 27
5.1 Introduction . . . 27
5.2 Performing simulations . . . 27
5.2.1 Main building blocks . . . 27
5.2.2 Simulation scenario contents . . . 28
5.2.3 Simulation execution and analysis . . . 28
5.3 The OTcl/C++ environment . . . 28
5.4 Class hierarchy . . . 29
5.5 Event scheduling . . . 29
5.6 Packet delivery . . . 30
5.7 Nodes . . . 31
5.7.1 Node basics . . . 31
5.7.2 Node configuration and creation . . . 31
5.7.3 Node addressing . . . 33
5.7.4 Routing . . . 33
5.8 Links . . . 33
5.9 Queues . . . 34
5.10 Packets . . . 35
5.10.1 Packet types and headers . . . 35
5.10.2 The common header . . . 36
5.10.3 Packet allocation and de-allocation . . . 36
5.11 Agents . . . 36
5.11.1 Agent state information . . . 37
5.11.2 Agent packet methods . . . 37
5.11.3 Agent attachment . . . 37
5.11.4 Connecting agents . . . 37
5.11.5 Application attachment . . . 38
5.11.6 Creating a new agent . . . 38
5.12 Timers . . . 39
5.13 Traffic generators . . . 40
5.14 Error models . . . 40
5.14.1 Usage in wired networks . . . 41
5.14.2 Usage in wireless networks . . . 41
5.15 Mobile networking . . . 42
5.15.1 Mobile nodes . . . 42
5.15.2 Packet transmission . . . 43
5.15.3 Packet reception . . . 43
5.15.4 Simulation scenario setup . . . 44
5.15.5 Mobile node configuration and creation . . . 44
5.15.6 Mobile node movements . . . 44

5.15.7 Routing in mobile networking 45
5.15.8 Miscellaneous features 46
5.16 Radio propagation models 46
5.16.1 Free space model 46
5.16.2 Two-ray ground reflection model 47
5.16.3 Shadowing model 47
5.17 Communication range adjustment 48
5.17.1 Network interface characteristics 49
5.17.2 Receive threshold calculation 49
5.18 Trace files 50
5.18.1 Trace configuration 50
5.18.2 Trace formats 51
5.19 Problems encountered with ns-2 54
5.20 Summary 54

6 Porting AODV-UU to ns-2 55
6.1 Introduction 55
6.2 Conversion overview 55
6.3 Initial decisions 55
6.4 Similar projects 55
6.5 Areas of concern 56
6.5.1 Environmental differences 56
6.5.2 Network interfaces 56
6.5.3 Packet handling 56
6.5.4 Kernel interaction 57
6.5.5 Variables 57
6.5.6 Timers 57
6.5.7 Logging 58
6.5.8 Non-applicable features 58
6.6 Different approaches of conversion 58
6.6.1 Monolithic class approach 59
6.6.2 Object-oriented approach 59
6.6.3 Daemon approach 59
6.7 The porting process 59
6.7.1 Choice of conversion approach 59
6.7.2 Conceptual overview 60
6.7.3 The AODVUU class 60
6.7.4 Methods 60
6.7.5 Source code extraction 61
6.7.6 Packet types and headers 63
6.7.7 Packet buffering and handling 65
6.7.8 Addressing 65
6.7.9 Timers 65
6.7.10 Logging 65
6.7.11 Tracing 66
6.7.12 Excluded features and modules 68
6.7.13 Configuration and usage 68
6.7.14 Extending the AODV-UU routing agent 70
6.7.15 General C vs C++ issues 70
6.7.16 Platform portability issues 71
6.7.17 Miscellaneous source code changes 71

6.7.18 Locating source code changes 72
6.7.19 The AODV-UU Makefile 72
6.7.20 Integrating AODV-UU with ns-2 72
6.8 Bugs found by simulation 74
6.9 Future work 74
6.10 Summary 74

7 Testing the Functionality of AODV-UU in ns-2 76
7.1 Introduction 76
7.2 General setup 76
7.3 simple-wireless-aodv-uu.tcl 76
7.3.1 Scenario overview 76
7.3.2 Scenario setup 76
7.3.3 Scenario description and results 77
7.3.4 Conclusions 77
7.4 wireless1-aodv-uu.tcl 77
7.4.1 Scenario overview 77
7.4.2 Scenario setup 77
7.4.3 Results and conclusions 77
7.5 The Roaming Node scenario 77
7.5.1 Scenario overview 78
7.5.2 Scenario setup 78
7.5.3 Scenario description and simulation results 79
7.5.4 Comparisons with real-world results 80
7.6 Other differences between simulations and real-world experiments 81
7.7 Comparisons with existing AODV implementation 81
7.8 Conclusions 82

8 Enabling Trace-based Simulation 83
8.1 Introduction 83
8.2 The APE testbed 83
8.3 APE logging 83
8.4 Modeling connectivity in ns-2 84
8.4.1 Communication range adjustment 84
8.4.2 Error model usage 84
8.4.3 Existing error models 85
8.4.4 Conclusions 85
8.5 Proposed solution 85
8.6 Related research 85
8.7 The SourceDrop error model 85
8.7.1 Functionality 86
8.7.2 Technical details 86
8.8 APE logfile processing 86
8.9 Initial testing 86
8.10 Roaming Node revisited 87
8.10.1 Simulation scenario setup 87
8.10.2 Results 87
8.10.3 Discussion 88
8.11 Conclusions 88

9 Summary and Future Work **89**
9.1 Summary of ns-2 port of AODV-UU 89
9.2 Summary of trace-based simulation 89
9.3 Comparisons of results 89
9.4 General project compliance 90
9.5 Future work 90

A Glossary **95**

List of Figures

2.1 An example ad-hoc network . . . 3
2.2 The hidden terminal problem . . . 5
2.3 The exposed node problem . . . 6

4.1 Netfilter hooks for IP . . . 18
4.2 Packet handling of AODV-UU . . . 25

5.1 Partial ns-2 class hierarchy . . . 29
5.2 Discrete event scheduling in ns-2 . . . 30
5.3 Packet delivery in ns-2 . . . 30
5.4 Composite construction of a wired unicast node . . . 31
5.5 Composite construction of a uni-directional link . . . 34
5.6 Packet contents . . . 35
5.7 Mobile node schematics . . . 42

7.1 Layout of the wireless1-aodv-uu.tcl scenario . . . 78
7.2 Layout of the Roaming Node scenario . . . 79
7.3 Routing agent comparison (link layer feedback) . . . 81
7.4 Routing agent comparison (HELLO messages) . . . 82

List of Tables

5.1 Selected options for node configuration 32
5.2 Available routing modules 33
5.3 Agent state information 37
5.4 Typical path loss exponent (β) values 48
5.5 Typical shadowing deviation (σ_{dB}) values 48
5.6 Trace log contents (old trace format) 52
5.7 Trace log contents (new trace format) 53
5.8 Example application-level trace log contents (new trace format) 53

6.1 AODV-UU routing agent default configuration 69

7.1 Roaming Node Ping statistics (simulation) 80
7.2 Roaming Node Ping success ratio (real-world) 80

8.1 Roaming Node Ping statistics comparison (run 1) 87
8.2 Roaming Node Ping statistics comparison (run 2) 87

Chapter 1

Introduction

1.1 Background

During recent years, the market of mobile communication has literally exploded. Cellular phones and other mobile devices with built-in wireless communication have gained enormous popularity, and because of this, the term "connectivity" has come to mean much more than it did just a couple of years ago. Computer networks, traditionally viewed as infrastructure of a fixed form, have evolved into combinations of wired and wireless networks to suit today's needs of mobile communication. As the mobility of users continues to increase, a special type of networks will be gaining more and more attention – *mobile ad-hoc networks*.

In a mobile ad-hoc network, nodes do not rely on any existing network infrastructure. Instead, the nodes themselves form the network and communicate through means of wireless communication. Nodes will have to forward network traffic on behalf of other nodes to allow communication to take place between nodes that are out of each others' immediate radio range. Hence, routing of network traffic becomes a central issue in these networks. For this purpose, numerous ad-hoc routing protocol specifications have been proposed, mainly by the IETF working group MANET [1]. Many of these proposals have been evaluated through simulations in network simulators [2, 3], but only a few of them have been evaluated through real-world tests [4, 5].

Both these approaches have their respective advantages and disadvantages. In simulations, experiments can easily be repeated and different parameters varied, which allows the impact of them to be studied. However, the drawback is that simplifications often are made in the models of the simulator, e.g. the models of wireless signal propagation. In real-world experiments, a routing protocol implementation can be evaluated without any specific assumptions or simplifications, but the testing environment can make it difficult to correlate observed results with details of the implementation. Therefore both approaches are useful, and complement each other.

1.2 Proposed goals

The proposed goals of this master's thesis are the following:

- To port AODV-UU [6], one of the existing implementations of the mobile ad-hoc routing protocol AODV [7], to run in the ns-2 network simulator [8].
- To enable trace-based AODV-UU simulations using logs from experiments conducted in the APE testbed [4], a Linux-based testbed for evaluation of ad-hoc protocols.
- To compare results of trace-based simulations with real-world results, and, if possible, improve existing models in the ns-2 network simulator.

1.3 Accomplishments

All the proposed goals were successfully achieved during this master's thesis project. AODV-UU was ported to the ns-2 network simulator, using the same code base for simulations as for real-world experiments. Trace-based simulations with AODV-UU were performed using a custom-made error model to model connectivity between wireless nodes, and the results indicate that trace-based simulations allow packet delivery ratios to be obtained that are very close to those of real-world experiments.

On a personal level, this master's thesis project has provided a considerable amount of insight on how to plan, perform and account for a scientific project. The initial question marks regarding the porting of AODV-UU were soon replaced by curiosity and a certain degree of self-confidence as I got acquainted to AODV-UU and the ns-2 network simulator. I was very happy to see the ported version of AODV-UU making it to the version 0.5 release in mid-June 2002 – this was one of the major milestones during the project.

1.4 Document overview

The rest of this document is outlined as follows.

Chapter 2 gives an introduction to mobile ad-hoc networks in general and describes how they differ from conventional networks. Problems specific to wireless communication in mobile ad-hoc networks are pointed out, and some example applications of ad-hoc networks are mentioned.

Chapter 3 describes the need for and properties of ad-hoc routing protocols. Some common ad-hoc routing protocols are reviewed, along with their suitability given different network and mobility conditions.

Chapter 4 gives a technical overview of the AODV-UU implementation of the AODV routing protocol. The functionality of its software modules is described, and the packet handling is studied in detail.

Chapter 5 gives an introduction to the ns-2 network simulator. Some of its network components are reviewed, and brief examples are provided to illustrate their usage. Emphasis is put on agents, mobile nodes and wireless communication, as this is of particular importance for the remaining chapters.

Chapter 6 describes the process of porting the AODV-UU implementation to run in ns-2. The required changes to the implementation and the steps of integrating it with the network simulator are described in detail. Instructions on configuration and usage are provided as a reference.

In Chapter 7, the functionality of the ported version of AODV-UU is tested through a number of simulation scenario test-runs. Simulation results are compared to expected results and real-world results, and some differences between simulations and real-world experiments are pointed out.

Chapter 8 describes the usage of logs from real-world experiments for enabling trace-based simulations with AODV-UU. An introduction to the APE testbed is given, followed by a brief investigation of its logging features. Different methods of incorporating support for trace-based simulations in the network simulator are reviewed, and the resulting model for modifying network connectivity in the simulator is described. Several test-runs are conducted using trace-based simulation support, and the results are evaluated.

Finally, Chapter 9 summarizes the outcome of the entire project, compares the results to the proposed goals, verifies general project compliance and provides some notes on future work.

Chapter 2

Ad-hoc Networks

2.1 Introduction

A wireless *mobile ad-hoc network (MANET)* is a network consisting of two or more mobile nodes equipped with wireless communication and networking capabilities, but lacking any pre-existing network infrastructure. Each node in the network acts both as a mobile host and a router, offering to forward traffic on behalf of other nodes within the network. For this traffic forwarding functionality, a routing protocol is needed.

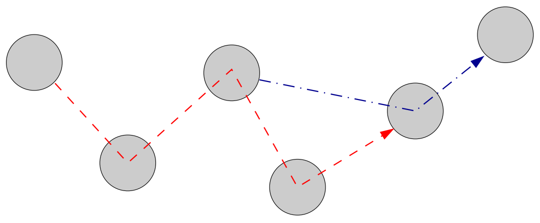


Figure 2.1: An example ad-hoc network. Each node acts both as a host and a router, forwarding traffic on behalf of other nodes.

The term “ad-hoc” suggests that the network can take on different forms, suitable for the task at hand, which in turn implies that ad-hoc networks are of a highly adaptive nature. In the following sections, the properties and applications of ad-hoc networks are investigated further.

2.2 Properties of ad-hoc networks

Perhaps the most important property of ad-hoc networks is that they do not rely on any pre-existing network infrastructure. Instead, these networks are formed in an on-demand fashion as soon nodes come sufficiently close to each other. This eliminates the need for stationary network components, such as routers and base stations, as well as cabling and central administration.

Nodes in an ad-hoc network should offer to forward network traffic on behalf of other nodes. If they refuse to do so, the connectivity between nodes in the network is affected in a negative manner. The functionality and usefulness of an ad-hoc network heavily depends on this forwarding feature of participating nodes.

Ad-hoc networks are often *autonomous* in the sense that they only offer connectivity between participating nodes, but no connectivity to external networks such as the Internet. In theory, however, nothing prevents a multi-homed node (with connections to both the ad-hoc network and one or more external networks) from acting as a gateway between those networks.

The dynamic topology imposed by ad-hoc networks is another very important property. Since the topology is subject to frequent changes, due to node mobility and changes in the surrounding environment, special considerations have to be taken when routing protocols are selected for the nodes. Traditional routing protocols such as OSPF [15] and RIP [16, 17] will fail to efficiently adapt to a dynamic topology of this kind, since frequent topology changes are not part of their normal operation. Therefore, special routing protocols are needed for ad-hoc networks.

Differences in the radio transmitter and receiver equipment of nodes, such as different transmission ranges and reception sensitivities, may lead to uni-directional links which could complicate routing in the ad-hoc network. Furthermore, not only communications equipment may differ between nodes, but also other resources such as battery capacity, CPU capacity and the amount of memory available. As an effect, nodes in an ad-hoc network may have very different abilities to participate in the network, considering the amount of service that they are willing provide to other nodes.

2.3 Comparisons with wired networks

2.3.1 Infrastructure

A conventional network consists of a more or less fixed infrastructure, built of nodes, routers, switches, gateways, bridges, base stations and other network devices, all connected by wires. The main property of these networks is that their topologies are more or less fixed. Any wish to reconfigure the network or add network devices requires physical intervention, and possibly a loss of service to some or all of the nodes while these changes are carried out. Furthermore, the administration of these networks is often centralized, because many of the nodes in the network usually rely on central servers for storage, access and processing of data.

Wireless networks in general and ad-hoc networks in particular can resolve some of these issues. Using wireless communication instead of fixed cabling solves the problem of cabling reconfiguration and the possible downtime caused by this. Ad-hoc networks add to this the ability of forming networks on-the-fly without any existing infrastructure. Ideally, the result is an on-demand network having all the advantages of wireless networks combined with virtually hassle-free setup and operation, as opposed to that of conventional, wired networks. Finally, since ad-hoc networks are formed by the nodes themselves, and the network topology may change rapidly, centralized solutions are not as common as in conventional networks.

2.3.2 Addressing

In a conventional network, addresses and other network parameters are either manually assigned or handed out to nodes using special protocols, such as DHCP (Dynamic Host Configuration Protocol) [18]. The hardware address of the network interface can be checked by an address allocation entity (e.g. a DHCP server), and an IP address assigned to the node.

In an ad-hoc network, the addressing issue becomes more complicated since there is no obvious, central authority in the network responsible for handing out addresses. Furthermore, there is no guarantee that the addresses taken by (or somehow assigned to) nodes reflect their geographical locations at all, due to node mobility.

2.3.3 Routing

Routing in a conventional network is performed by special routers; either hardware-based routers specialized for this task, or computers equipped with several network interfaces and accompanying software to perform the actual routing. In an ad-hoc network, routing is instead carried out by the participating nodes themselves. Each node has the ability to forward traffic for others, and this ability is what makes it possible for traffic to flow through the ad-hoc network over multiple hops. Unlike a stationary router in a conventional network, an ad-hoc node does not need to be equipped with several network interfaces, since all communication is usually done through a single wireless channel.

Another difference in ad-hoc network routing is that no subnetting assumptions are made, i.e., routing tables may end up consisting of separate IP addresses (which need not have any correlation with each other). As an effect, a flat addressing structure is imposed. In a conventional network, routing tables instead usually contain network prefixes – not entire addresses – paired with network interface identifiers.

2.4 Ad-hoc networking issues

In this section, some of the practical problems encountered in wireless ad-hoc networks are presented along with proposed solutions found in literature and related research material. Since nodes in an ad-hoc network use wireless communication, many of the issues encountered in wireless LANs apply to ad-hoc networks as well.

2.4.1 Wireless medium access

Since a mobile wireless ad-hoc network uses a wireless medium for data transmission, access to this medium must be controlled to prevent nodes from attempting to transmit data simultaneously. This control is provided by a MAC (Media Access Control) protocol, such as IEEE 802.11 [34].

A well-known problem in this context is the *hidden terminal problem*. It occurs when two nodes, out of each others' radio range, simultaneously try to transmit data to an intermediate node, which is in radio range of both the sending nodes. None of the sending nodes will be aware of the other node's transmission, causing a collision to occur at the intermediate node. This is shown in Figure 2.2.

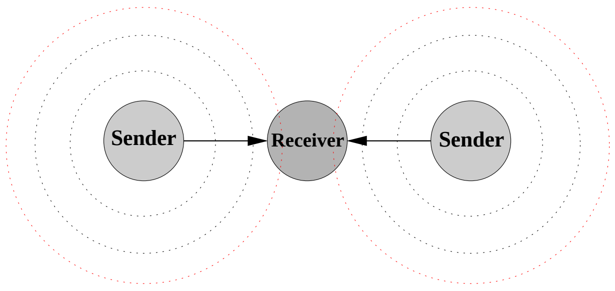


Figure 2.2: The hidden terminal problem. The sending nodes are unaware of each others' transmissions, resulting in a collision at the receiving node.

To avoid this problem, a *handshake protocol* could be used, such as the RTS-CTS handshake protocol. A node that wishes to send data is required to ask for permission before doing so, by sending a RTS (Request To Send) message to the receiving node. The receiving node then replies with a CTS (Clear To Send) message to grant this request. The CTS message can be heard by all nodes within radio range of the receiving node, and instructs them not to use the wireless medium since another transmission is about to take place. The node that requested the transmission can then begin sending data to the receiving node.

This however does not provide a perfect solution to the hidden terminal problem. For instance, if RTS and CTS control messages collide at one of the nodes, that node will be unaware of both these messages. Since it believes that the channel is free to use, it could reply to future RTS messages from other nodes, causing a collision with the ongoing data transfer.

Another problem in wireless medium access is the *exposed node problem*. It occurs when a node overhears another transmission and hence refrains to transmit any data of its own, even though such a transmission would not cause a collision due to the limited radio range of the nodes. This is shown in Figure 2.3.

Clearly, the exposed node problem is leads to sub-optimal utilization of the wireless medium. Some proposed solutions are the usage of directional antennas (instead of omni-directional antennas) or separate channels for control messages and data [13]. Finally, a technique called *transmission power control* [11] could serve a dual purpose in this context. By adjusting the transmission power of nodes, interference can be reduced at the same time as nodes save valuable energy.

2.4.2 Addressing

The issue of assigning unique identities to nodes in an ad-hoc network, e.g. in form of IP addresses, is another area of research. It is not obvious how such address allocations should be made, given the fact that mobile ad-hoc networks potentially could grow very large and the mobility of nodes may be very high. Proposed solutions

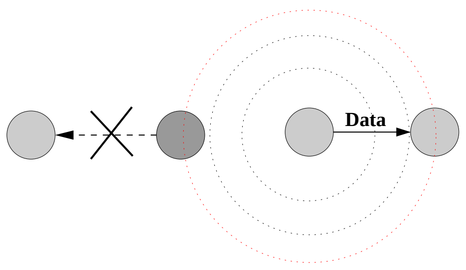


Figure 2.3: The exposed node problem. A node wishing to transmit data refrains to do so because it overhears an ongoing transmission.

to the addressing problem can e.g. be found in [12], where nodes select their own IP addresses at random, and use ARP (Address Resolution Protocol) [14] for translating network hardware addresses into IP addresses and vice versa. Address collisions are detected by listening to ARP traffic from other nodes, spotting any attempts to use IP addresses that are already occupied.

The addressing problem is one of the major problems restraining the usage of ad-hoc networks today, since all nodes need to be assigned unique addresses for any unicast packet transmission to take place. It is of great importance that this issue will be investigated further, to allow ad-hoc networks to be created on-the-fly as intended.

2.4.3 Network security

Network security in ad-hoc networks is a concern of great importance. Transmission of sensitive data, forwarded by intermediate nodes whose intentions are unknown, may turn out to be unsuitable at best and fatal at worst. Countermeasures need to be taken to prevent unwanted exposure of sensitive information in the ad-hoc network. This could be achieved by using conventional encryption; however, key exchange techniques and public key infrastructure (PKI) solutions are harder to apply to ad-hoc networks, because of the lack of network infrastructure and authorities of trust.

Network security could also be used for dealing with denial-of-service (DoS) or spoofing attacks against an ad-hoc network. For instance, security-enhanced versions of ad-hoc routing protocols could be used to ensure that the operation of the routing protocol (and hence, largely, the operation of the ad-hoc network) remains unaffected by attempts to forge or alter routing protocol control messages.

A good overview of wireless ad-hoc network security issues can be found in [21], which covers topics such as trust, key management, secure routing, intrusion detection and availability, and also provides references to current research material.

2.5 Applications of ad-hoc networks

Several applications of ad-hoc networks have been proposed. Some of the most common ones are:

- *Spontaneous networks:* Business colleagues or participants of a meeting could use an ad-hoc network for sharing documents, presentation material and notes on-the-fly.
- *Disaster areas:* In the event of a disaster, such as earthquake or fire, the existing network infrastructure may very well be out of order. Ad-hoc networks could be used by emergency personnel (police, medical staff, rescue coordinators, fire brigades etc.) for establishing an on-site communications network. As a result of this, valuable time can be saved in these situations, perhaps saving lives.
- *Building of wireless networks:* Ad-hoc networks could be used for building wireless networks where installation of network cabling is difficult, impossible or too expensive. Examples include ancient build-

ings that may not be modified, or are built in such a way that wiring becomes difficult. Mobile ad-hoc nodes could replace both conventional base stations and servers.

- *Wireless sensor networks* is another application of ad-hoc networks, where thousands or even ten thousands of electronic sensors populate the network. These sensors collect data and report to selected nodes at predetermined times or with certain intervals. The collected data can subsequently be used in larger computations, with the ultimate intention of generating overall statistics based on the large number of reports. Wireless sensor networks could be used in the military (e.g. for determining if an area is safe in terms of radiation), but also in everyday applications such as quality control and gathering of weather information.

2.6 Conclusions

To conclude, ad-hoc networks possess special properties that make them attractive, but these properties come at a certain cost, e.g. in terms of network security. It is up to the participants of such a network (i.e., the nodes) to decide if they are willing to pay this price, given a specific task at hand. Hopefully, future research in ad-hoc networking will solve some of the practical issues that these networks are facing right now. In pace with the increased mobility of users, ad-hoc networks certainly have the potential to become very useful and popular.

Chapter 3

Ad-hoc Routing Protocols

3.1 Introduction

For a packet to reach its destination in an ad-hoc network, it may have to travel over several hops. The main purpose of a *routing protocol* is to set up and maintain a *routing table*, containing information on where packets should be sent next to reach their destinations. Nodes use this information to forward packets that they receive.

In this chapter, the need for *ad-hoc routing protocols* is discussed, and some ad-hoc routing protocols are described. The intention is not to present all existing protocols, nor to provide full technical details, but rather to describe the main ideas behind some of them and how they work. For this master's thesis, the AODV routing protocol (described in section 3.6.3) is of particular interest.

3.2 The need for ad-hoc routing protocols

Routing is not a new issue in computer networking. Link-state routing (e.g. OSPF [15]) and distance vector routing (e.g. RIP [16, 17]) have existed for a long time and are widely used in conventional networks. However, they are not very suitable for use in mobile ad-hoc networks, for a number of reasons:

- They were designed with a *quasi-static topology* in mind. In an ad-hoc network, with a frequently changing network topology, these routing protocols may fail to converge, i.e., to reach a steady state.
- They were designed with a *wired network* in mind. In such a network, links are assumed to be bi-directional. In mobile ad-hoc networks, this is not always the case; differences in wireless networking hardware of nodes or radio signal fluctuations may cause uni-directional links, which can only be traversed in one direction.
- They try to maintain routes to *all reachable destinations*. In mobile ad-hoc networks with a very large number of nodes, such as a large wireless sensor networks, this may become infeasible because of the resulting very large number of routing entries.
- In mobile ad-hoc networks, periodic flooding of routing information is relatively expensive, since all nodes compete for access to the wireless medium (which usually offers a rather limited amount of bandwidth).

Therefore, special routing protocols are needed for ad-hoc networks.

3.3 Classification of ad-hoc routing protocols

Ad-hoc routing protocols are usually classified by the approach they use for maintaining and updating their routing tables. The two main approaches are:

- *Proactive (table-driven) operation:* In this approach, the routing protocol attempts to maintain a routing table with routes to all other nodes in the network. Changes in the network topology are propagated by means of updates throughout the entire network to ensure that all nodes share a consistent view of the network.

  The advantage of this approach is that routes between arbitrary source - destination pairs are readily available, all the time. The disadvantages are that the routing tables will occupy a large amount of space if the network is large, and that the updates may lead to inefficient usage of network resources if they occur too frequently.

- *Reactive (on-demand-driven) operation:* In this approach, routes to a destination are acquired by a *route discovery* process in an on-demand fashion, i.e., a route is not searched for unless it is needed. The acquired route is maintained by a *route maintenance* process until it has been determined that the route is not needed anymore, or has become invalid.

  The advantage of this approach is that unnecessary exchange of route information is avoided, leaving more network resources available for other network traffic. The disadvantage is that route look-ups could take some time. Depending on the application, this may or may not be acceptable.

There also exist hybrid approaches, combining both the proactive and the reactive approach. A more fine-grained classification of ad-hoc routing protocols and a taxonomy for comparing them can be found in [33].

3.4 Properties of ad-hoc routing protocols

As mentioned previously, the characteristics of ad-hoc networks call for routing protocols specifically made for these networks. According to [20], some desirable properties of ad-hoc routing protocols are the following:

- *Distributed operation:* This is essential to mobile ad-hoc networks. Since the mobile ad-hoc network does not rely on any pre-existing infrastructure, its operation should be distributed and decentralized.

- *Loop-freedom:* This is usually desired, although may not be strictly required. Ensuring loop-freedom could help avoiding worst-case scenarios, such as packets looping within the network for arbitrary periods of time.

- *Demand-based (reactive) operation:* Instead of maintaining routing tables with information on routes between all nodes in the network, the routing protocol should find routes as they are needed. This avoids wasting bandwidth of the network.

- *Proactive operation:* This is the opposite of reactive operation. If the overhead of searching routes on an on-demand basis is unacceptable for the application, and the resources of the network (e.g. bandwidth) and its nodes (e.g. energy constraints) allow for it, proactive operation could be desired instead.

- *Security:* It is desirable that the routing protocol has the ability to utilize security techniques to prohibit disruption or modification of the operation of the protocol.

- *Sleep period operation:* Since nodes in a mobile ad-hoc network usually have a limited amount of energy (e.g. battery power), it is desirable that the routing protocol has the ability to cope with situations where nodes may have powered themselves off temporarily to save energy.

- *Support for uni-directional links:* Since there is no guarantee that links are bi-directional in a mobile ad-hoc network, the ability to use separate uni-directional links in both directions to replace a bi-directional link could be of great value, should such a situation occur.

3.5 The IETF MANET working group

Specifications for many existing ad-hoc routing protocols have been developed by the IETF (Internet Engineering Task Force) working group MANET [1], a working group focusing on mobile ad-hoc networks. Their near-term goal is to standardize one or more intra-domain unicast routing protocols, and currently, they have published drafts for the following nine ad-hoc routing protocols:

- AODV (Ad hoc On-demand Distance Vector) [7]
- DSR (Dynamic Source Routing) [23]
- ZRP (Zone Routing Protocol) [24]
- OLSR (Optimized Link State Routing) [25]
- LANMAR (Landmark Routing Protocol) [26]
- FSR (Fisheye State Routing) [27]
- IERP (Interzone Routing Protocol) [28]
- IARP (Intrazone Routing Protocol) [29]
- TBRPF (Topology Broadcast based on Reverse-Path Forwarding) [30]

In addition to developing routing protocol specifications, the MANET working group also serves as a meeting place and forum for discussions on mobile ad-hoc networking issues in general. As such, it has become an extremely valuable resource for researchers and developers in this area.

3.6 Examples of ad-hoc routing protocols

In this section, some ad-hoc routing protocols are presented. Focus is put on how they work and their suitability given different network and mobility conditions, as this is important for their deployment in an ad-hoc network.

3.6.1 DSDV

Although perhaps somewhat outdated, DSDV has influenced many other ad-hoc routing protocols. It is included here as a reference.

Description

DSDV (Destination-Sequenced Distance-Vector) [22] is a proactive ad-hoc routing protocol that is based on the distributed Bellman-Ford routing algorithm [19, pp. 290-292]. The classical count-to-infinity problem of Bellman-Ford is avoided by using sequence numbers, allowing nodes to distinguish between stale and new routes and ensuring loop-freedom.

Operation

Since DSDV is a proactive routing protocol, each node maintains a routing table with all other destinations in the network listed along with the next hop and the number of required hops to reach each destination. Routing table updates in DSDV are distributed by two different types of update packets:

- *Full dump:* This type of update packet contains all routing information available at a node. As a consequence, it may require several NPDUs (Network Protocol Data Units) to be transferred if the routing table is large. Full dump packets are transmitted infrequently if the node only experiences occasional movement.

- *Incremental:* This type of update packet contains only the information that has changed since the latest full dump was sent out by the node. Hence, incremental packets only consume a fraction of the network resources compared to a full dump.

Broadcasts of route information contain the destination address, the number of hops required to reach the destination (the cost metric), the highest sequence number known of that destination and another sequence number, unique to each broadcast. Nodes receiving such a broadcast will update their routing tables if the destination sequence number is greater than the existing one, or if it equals the existing one but the number of hops to reach the destination is smaller. New routes will immediately be advertised to a node's neighbors, and updated routes will cause an advertisement to be scheduled for transmission within a certain settling time. For its operation, DSDV requires the values of three parameters to be set:

- The *incremental update period*
- The *full update period*
- The *settling time* for routes

The values of these parameters largely determine the performance of DSDV, and must be chosen carefully. Research, such as in [2] and [3], indicates that DSDV experiences low throughput and fails to converge as node mobility increases. Therefore, the DSDV protocol is probably best used in ad-hoc networks where node mobility is low or moderate.

3.6.2 OLSR

Description

OLSR (Optimized Link State Routing) [25] is another proactive ad-hoc routing protocol. It is an optimization of the classical link state algorithm [19, pp. 292-301] tailored to mobile wireless LANs. The key concept of OLSR is that of *multipoint relays (MPRs)*. Multipoint relays are selected nodes that broadcast messages during the flooding process. This approach reduces the number of control messages transmitted over the network. In addition, the control messages of OLSR contain less information than those of the classical link state algorithm, limiting the amount of overhead even further.

OLSR provides optimal routes in terms of number of hops and is supposed to be particularly suitable for large and dense networks. It works independently from other protocols, and makes no assumptions about the underlying link layer.

Operation

Each node selects a set of multipoint relays (MPRs) among its neighbors, such that the set covers all nodes that are two hops away in terms of radio range. Also, each node keeps information about the set of neighbors which have selected it as one of their MPRs. This information is obtained from periodic HELLO messages. (Periodic HELLO messages are used for neighbor sensing in OLSR, since the protocol does not make any assumptions about the underlying link layer.)

Routes are established through the selected MPRs of a node. This means that a path to a destination is a series of hops through MPRs, from the source to the destination. Since OLSR is a proactive protocol, routes are immediately available when needed.

Changes in connectivity cause link state information to be broadcasted by a node. This information is then re-broadcasted by the other nodes in that node's MPR set only, reducing the number of messages required to flood the information throughout the network. Also, the link state information broadcasted by a node contains only the state of links to nodes in its MPR set, not all its neighbors. This is sufficient, since each two-hop neighbor of a node is a one-hop neighbor of some node in the MPR set.

Summary

OLSR is an attractive proactive routing protocol suitable for large and dense ad-hoc networks. One of its benefits is that it does not make any assumptions about the underlying link layer, allowing it to be used in a variety of configurations. Neighbor sensing is performed by periodic beaconing rather than link layer feedback. If route acquisition time is important to the application, the ad-hoc network is large and/or dense and traffic is exchanged between a large number of nodes, OLSR could be a suitable choice.

3.6.3 AODV

Description

AODV (Ad hoc On-demand Distance Vector) [7] is a reactive ad-hoc routing protocol utilizing destination sequence numbers to ensure loop-freedom at all times and to avoid the count-to-infinity problem associated with classical distance-vector protocols. It offers low overhead, quick adaptation to dynamic link conditions and low processing and memory overhead.

Message types

AODV defines three different message types for routing protocol control packets:

- Route Request (RREQ)
- Route Reply (RREP)
- Route Error (RERR)

Route discovery

Since AODV is a reactive protocol, routes to destinations are acquired in an on-demand manner. When a node needs a route to a destination, it broadcasts a RREQ message. As this message is spread throughout the network, each node that receives it sets up a *reverse route*, i.e., a route towards the requesting node. As soon as the RREQ message reaches a node with a fresh enough route to the specified destination, or the destination itself, a RREP unicast message is sent back to the requesting node. Intermediate nodes use the reverse routes created earlier for forwarding RREP messages.

If intermediate nodes reply to all transmissions of a given RREQ message, the destination node being searched for never learns about a route back to the requesting node, as it never receives the RREQ. This could cause the destination node to initiate a route discovery of its own, if it needs to communicate with the requesting node (e.g. to reply to a TCP connection request). To solve this, the originator of a RREQ message should set a *gratuitous RREP* flag in the RREQ if it believes that this situation is likely to occur. An intermediate node receiving a RREQ with this flag set and replying with a RREP must also unicast a *gratuitous RREP* to the destination node. This allows the destination node to learn a route back to the requesting node.

Routing table information

Each routing table entry maintained by AODV contains the following fields:

- Destination IP Address
- Destination Sequence Number
- Valid Destination Sequence Number
- Interface
- Hop Count (number of hops needed to reach the destination)

- Next Hop
- List of Precursors (described below)
- Lifetime (expiration or deletion time of the route)
- Routing Flags
- State (valid or invalid)

Route maintenance

Nodes monitor the link status of the next hop in *active routes*, i.e., routes whose entries are marked as valid in the routing table. This can be done by observing periodic broadcasts of HELLO messages (beacons) from other nodes, or any suitable link layer notifications, such as those provided by IEEE 802.11 [34]. When a link failure is detected, a list of unreachable destinations is put into a RERR message and sent out to neighboring nodes, called *precursors*, that are likely to use the current node as their next hop towards those destinations. For this purpose, nodes maintain a precursor list for each routing table entry. Finally, routes are only kept as long as they are needed. If a route is not used for a certain period of time, its corresponding entry in the routing table is invalidated and subsequently deleted.

Summary

AODV is currently one of the most popular ad-hoc routing protocols and has enjoyed numerous reviews (e.g. in [2], [3] and [35]). These indicate that AODV performs very well both during high mobility and high network traffic load, making it one of the most interesting candidates among today's ad-hoc routing protocols. Several independent AODV implementations exist, such as AODV-UU [6] and Mad-hoc AODV [38].

3.6.4 DSR

Description

DSR (Dynamic Source Routing) [23] is a reactive ad-hoc routing protocol based on *source routing*. Source routing means that each packet contains in its header an ordered list of addresses through which the packet should pass on its way to the destination – this source route is created by the node that originates the packet. The usage of source routing trivially allows routing of packets to be loop-free, avoids the need for keeping up-to-date routing information in intermediate nodes and allows nodes that overhear packets containing source routes to cache this information for their own future use.

Route discovery

A node that wishes to send a packet to a destination checks in its *route cache* if it has a route available. If no route is found in the route cache, route discovery is initiated. The node initiating the route discovery broadcasts a *Route Request* packet, containing its own address, the destination (target) address and a unique request identification. Each Route Request also contains an initially empty list of nodes through which this particular Route Request packet has passed.

If a node receives such a Route Request and discovers that it is the target of the request, it responds with a *Route Reply* back to the initiator of the Route Request. The Route Reply contains an copy of the accumulated route record from the Route Request. If a node receiving a Route Request instead discovers that it is *not* the target of the request, it appends its own address to the route record list of the Route Request, and re-broadcasts the Route Request (with the same request identification).

Route Replies are sent back to the Route Request initiator using a route found in the route cache of the replying node. If no such route exists, the replying node should itself initiate a route discovery to find a route back to the originator of the original Route Request (and piggyback its Route Reply, to avoid possible infinite recursion of route discoveries). However, if the replying node uses a MAC protocol that requires bi-directional links, such as IEEE 802.11 [34], the node replying to the initiator's Route Request should instead reverse the route found in the Route Request, and use that route for its Route Reply. This ensures that the discovered route is bi-directional, and eliminates the need for an additional route discovery. Finally, when the initiator of a Route Request receives the Route Reply, it caches the route in its route cache and uses it for sending subsequent packets to that destination.

Route maintenance

If a route in the route cache is found to be broken – by link-layer notifications, lack of passive acknowledgements, lack of network-layer acknowledgements or reception of packets containing *Route Error* information – the node should remove the broken route from its route cache and send a Route Error to each node that has sent a packet routed over that link since an acknowledgement was last received. In addition, the node should notify the original senders of any affected pending packets by sending a Route Error to them, and try to salvage the pending packets. Salvaging can be done by examining the node's route cache for additional routes to the same destination and replacing the source route with a valid one. If no such alternative routes exist, the pending packets are discarded.

Summary

The many caching features found in DSR and its techniques for detecting and utilizing uni-directional links (if the underlying link layer can cope with those) makes it attractive for many ad-hoc networking configurations. However, the source routing approach comes at the cost of increased overhead, since each packet must carry the complete path to its destination in its packet header. Simulation studies such as [2] and [3] indicate that DSR works very well when network traffic loads are moderate, and that high mobility does not pose any particular problems. The number of control packets sent by DSR is generally much lower than for e.g. AODV with HELLO messages, but the byte overhead is larger because of the source routes contained in packets.

3.6.5 TORA

Description

TORA (Temporally-Ordered Routing Algorithm) [31] is a reactive ad-hoc routing protocol based on link reversal algorithms. It offers distributed operation, i.e., routers only need to maintain information about adjacent routers, and prevents long-lived loops. It minimizes communication overhead by localizing the algorithmic reaction to network topology changes, and also offers proactive operation as an option.

For each destination, TORA builds and maintains a *directed acyclic graph (DAG)* which is used for routing packets to that destination. By associating *heights* with each node in the network and adjusting these as necessary, packets always flow downstream on their way to the destination. If a link breaks, the heights of nodes can be adjusted by link reversal algorithms to allow the direction of the link to be reversed.

Requirements

For its operation, TORA requires IMEP (Internet MANET Encapsulation Protocol) [32], a protocol designed specifically to support the operation of mobile ad-hoc network routing protocols. The IMEP features of specific interest to TORA are reliable, ordered broadcasts for its distributed operation, link and network layer address resolution and mapping, security authentication and (preferrably) link status sensing to avoid beaconing for node presence indication.

Route creation

In reactive mode, routes are created as needed. The two packet types used for this are *Query (QRY)* packets and *Update (UPD)* packets. A node initiates the route creation process by sending a QRY packet to its neighbors, containing the identifier of the destination for which a route is requested. The QRY packet is propagated by neighboring routers until it is received by a router that has a trusted route to the destination, which will reply with a UPD packet indicating its height. On its way back to the node that initiated the route discovery, the UPD packet will cause routers to adjust their heights to be larger than the height specified in the UPD packet. Each router forwarding such a UPD packet will include their new height in the UPD packet, allowing the requested route to be created as a downstream route from the requesting node to the destination.

In proactive mode, the node initiating the route discovery instead sends an *OPT (Optimization)* packet that is processed and forwarded by neighboring routers. The OPT packet allows routers to change their mode of operation to proactive operation, but otherwise has the same purpose as a QRY packet.

Route maintenance

Maintenance of routes is performed by reacting to topological changes in the network, such that routes to the destination are re-established within a finite amount of time. When a node detects a link failure, it adjusts its height to a local maximum with respect to the failed link. It then broadcasts a UPD packet to its neighbors. However, a route failure is not propagated until a node loses its last downstream link.

If network partitioning is detected by a node, all links in that partition are marked as undirected to erase invalid routes. This is performed by sending *Clear (CLR)* packets to all neighbors. After such a partitioning, nodes will have to re-initiate the route discovery process when they require a route to a destination.

Summary

TORA uses a rather unique *link reversal* approach for solving the problem of routing in ad-hoc networks. However, its underlying requirements, i.e., the usage of IMEP, has proven to be a considerable source of overhead. Simulations in [2] indicate that this amount of overhead may grow so large that TORA effectively suffers a congestive collapse when the number of nodes reaches 30; contention of the wireless medium results in collisions which in turn make the situation even worse, fooling the IMEP protocol to erroneously believe that links are breaking. Therefore, TORA is probably best suited for small or moderately large ad-hoc networks.

3.7 Conclusions

From the examples of ad-hoc routing protocols presented in this chapter, it should be evident that the selection of an ad-hoc routing protocol for use in an ad-hoc network requires careful thought. Parameters such as network size, mobility and traffic load all have an impact on the suitability of each protocol. This, together with the spontaneous nature of ad-hoc networks, easily turns the selection of a routing protocol into a complicated task. Possibly, the ongoing work of the IETF MANET working group [1] – to promote a few of the existing ad-hoc routing protocol specifications to experimental RFCs – could be of help, but it is too early to tell.

Chapter 4

AODV-UU

This chapter provides a technical overview of AODV-UU. The main focus is put on its software modules and packet handling, as this is crucial for its operation and for the porting of AODV-UU to the ns-2 network simulator described in Chapter 6.

4.1 Introduction

AODV-UU [6] is a Linux implementation of the AODV (Ad hoc On-demand Distance Vector) [7] routing protocol, developed at Uppsala University, Sweden. It runs as a user-space daemon, maintaining the kernel routing table. AODV-UU was written in the C programming language and has been released under the GNU General Public License (GPL).

AODV-UU implements all mandatory and most optional features of AODV. One of the main goals of AODV-UU was to supply an AODV implementation that complied with the latest draft – not older versions – and this goal is upheld by continuous software development. New users of AODV-UU will appreciate its easy installation, stability and ease of use.

The system requirements of AODV-UU are rather modest. A recent Linux distribution with a version 2.4.x kernel along with a wireless network card suffices (it is also possible to use a wired networking setup). In addition, AODV-UU can be cross-compiled for the ARM platform so that it can be used on many popular PDAs, e.g. the COMPAQ iPAQ and the Sharp Zaurus. The version of AODV-UU described in this chapter is version 0.5, which complies with version 10 of the AODV draft. The complete source code is available from the AODV-UU homepage [6].

4.2 Installation and usage

Installation of AODV-UU is very straightforward. Compilation is handled by the UNIX `make` utility and a corresponding Makefile for AODV-UU, resulting in a user-space routing daemon (`aodvd`) and a kernel module (kaodv.o). The Makefile also offers a target to install the kernel module on the system. When installed, the kernel module is automatically loaded by the `modprobe` module loading system as soon as it is needed. To run AODV-UU, one executes the routing daemon (and optionally detaches it from the console). From there, the operation of AODV-UU should be more or less transparent.

During execution, the AODV-UU routing daemon will log relevant events to a logfile if logging has been enabled. Also, if routing table logging has been enabled, the routing table will be periodically dumped to a routing table logfile. These two logfiles are useful for analyzing the behavior of AODV-UU, e.g. for post-run analysis of ad-hoc networking experiments, but can also help gaining understanding of AODV operation in general.

4.3 Configuration

AODV-UU offers many options for adjusting its operation. These are supplied as parameters on the command line to the `aodvd` routing daemon. The following options are available:

- **Daemon mode** (-d, --daemon): Allows detaching of the routing agent from the console, i.e., transparent execution in the background.
- **Force gratuitous** (-g, --force-gratuitous): Forces the gratuitous flag to be set on all RREQs. Gratuitous RREQs are described in Section 3.6.3.
- **Help** (-h, --help): Displays help information.
- **Interface** (-i, --interface): Specifies which network interfaces that AODV-UU should be attached to. The default is the first wireless network interface.
- **HELLO jittering** (-j, --hello-jitter): Disables jittering of HELLO messages.
- **Logging** (-l, --log): Enables logging to an AODV-UU logfile.
- **Routing table logging** (-r N, --log-rt-table N): Logs the contents of the routing table to a routing table logfile every N seconds.
- **N HELLOs** (-n N, --n-hellos N): Requires N HELLO messages to be received from a node before it is treated as a neighbor.
- **Uni-directional hack** (-u, --unidir-hack): Enables detection and avoidance of uni-directional links. *This is an experimental feature*.
- **Gateway mode** (-w, --gateway-mode): Enables Internet gateway support. *This is an experimental feature*.
- **Disabling of expanding ring search** (-x, --no-expanding-ring): Disables the expanding ring search for RREQs, which is normally used for limiting the dissemination of RREQs in the network.
- **No wait-on-reboot** (-D, --no-worb): Disables the 15-second wait-on-reboot delay at startup.
- **Version information** (-V, --version): Displays version and copyright information.

Those features of AODV-UU that are marked as experimental are not guaranteed to work, and may disappear in any future release. Therefore, they should be used with caution.

4.4 Interaction with IP

Since AODV is a reactive protocol, route acquisition is done on demand. This requires the routing protocol implementation to be able to intercept requests for destinations to which a valid route does not exist. Furthermore, the timeout-based removal of stale routing table entries requires support for monitoring of packets at each host.

Early AODV implementations such as Mad-hoc AODV [38] and the implementation described by Larsson and Hedman in [35] were unable to intercept and temporarily delay packets for which a route did not exist, causing initial connection attempts to a previously “unknown” node to fail. The temporary work-around was to require the user to manually generate some arbitrary initial traffic to that node for route establishment to take place. This hindered transparent operation of connection-oriented protocols such as TCP, where initial packets are vital for connection setup. Since then, packet handling support in Linux has been vastly improved. Most notably, a software framework called *Netfilter* has been developed, allowing very flexible packet handling to be performed. AODV-UU uses Netfilter for all its packet processing and modification needs.

4.4.1 The Netfi lter framework

Netfilter [36] is a Linux kernel framework for mangling packets. For each network protocol, certain *hooks* are defined. These hooks correspond to well-defined places in the protocol stack and allow custom packet mangling code to be inserted in form of kernel modules. Figure 4.1 illustrates this for the IP protocol.

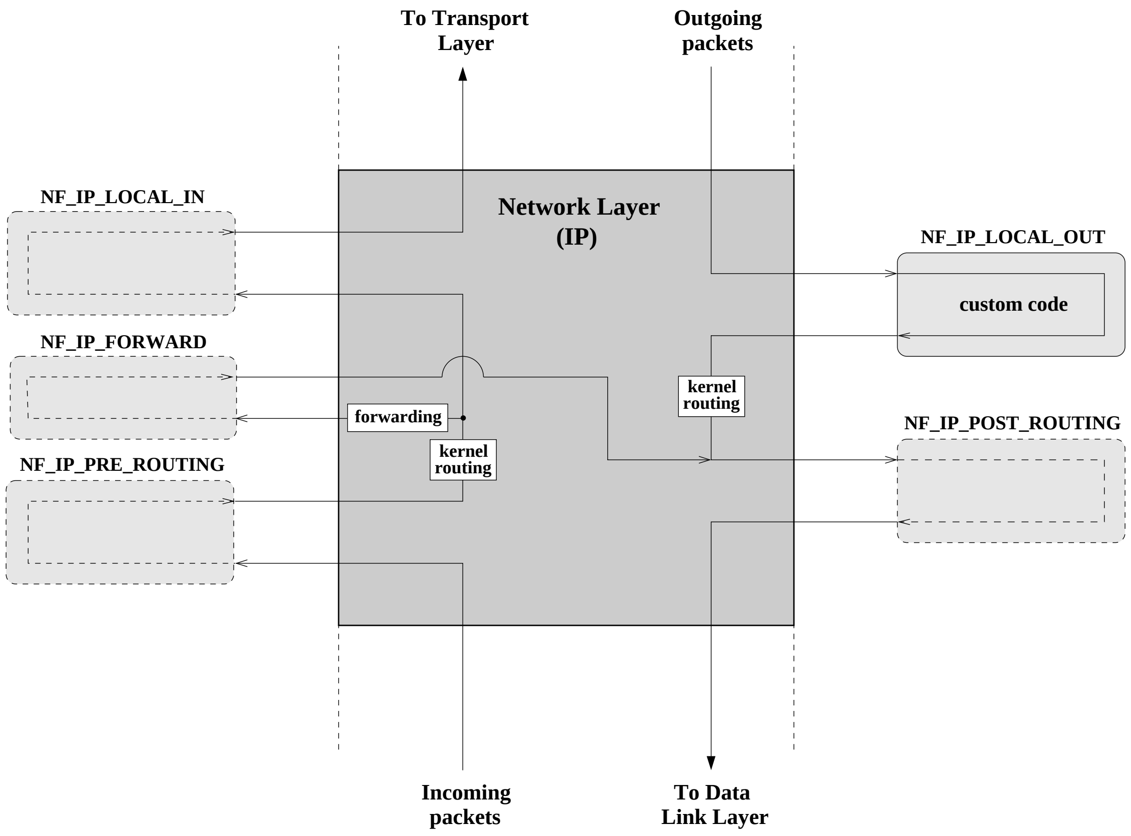


Figure 4.1: Netfilter hooks for IP. Packets delivered on these hooks can be captured and modified by custom code segments (kernel modules).

Each packet arriving at such a hook is delivered to the code segments that have registered themselves on that hook. This allows packets to be altered, dropped or re-routed by these custom code segments. When a packet has been processed (and possibly modified), a *verdict* should be returned to Netfilter. This verdict instructs Netfilter to perform some packet-related action, e.g. to drop the packet or to let it continue its traversal through the protocol stack.

Netfilter also offers the ability to process packets in user-space. By returning a special queue verdict, packets are queued in kernel space and information about the packet is sent to user-space over a *netlink socket*. Queued packets remain queued until the user-space application, at the other end of the socket, returns verdicts indicating the desired actions for these packets.

In AODV-UU, a kernel module component constantly listens to both inbound and outbound packets by registering itself on the appropriate Netfilter hooks. Packets are queued as needed to allow user-space packet processing to be performed by the AODV-UU routing daemon. This allows the on-demand route acquisition and operation of AODV to be realized. Finally, the Netfilter approach allows the implementation to stay independent of kernel modifications. This is a big advantage in terms of software maintenance.

4.4.2 Performance

The user-space packet processing comes at a certain performance penalty, but this has not been a major concern during the development of AODV-UU. Instead, the developers have aimed for stability and completeness of features. Current experience estimates the additional delay caused by user-space processing to be roughly one millisecond for a one-hop round-trip packet.

4.5 Software modules

AODV-UU consists of a variety of software modules, i.e., C source code files (.c) with corresponding header files (.h). Also, some header files constitute placeholders for definitions (constants and macros) only. In the following subsections, the software modules are divided into kernel-related and non-kernel-related modules, and each module is described individually. The intention is to summarize the functionality contained in the source code, as an initial step towards the porting of AODV-UU to the ns-2 network simulator described in Chapter 6.

4.5.1 Non-kernel-related modules

aodv_hello.{c, h}

This module contains HELLO message functionality. It offers generation and sending of HELLO messages, scheduling of HELLO message generation, processing of HELLO messages, and forced processing of a message as if it were a HELLO message (to extract neighbor information). It should be noted that HELLO messages in fact are RREP messages with their fields set to special values. Neighbor set extraction is performed through reading special *AODV message extensions*, i.e., additional information contained within an AODV message.

aodv_rerr.{c, h}

This module contains RERR message functionality. It defines a datatype for RERR messages and unreachable destinations, and offers RERR message creation, addition of unreachable destinations to RERR messages and processing of RERR messages.

aodv_rrep.{c, h}

The aodv_rrep module contains RREP message functionality. It defines the datatypes for RREP and RREP-ACK messages. The latter are used for acknowledgment of RREP messages, if the link over which the RREP was sent may be unreliable or uni-directional. The module offers creation and processing of messages of these two message types.

aodv_rreq.{c, h}

This module contains RREQ message functionality. It defines the datatype for RREQ messages, offers creation and processing of RREQ messages, and allows route discoveries to be performed. It also offers buffering of received RREQs and blacklisting of RREQs from certain nodes. The blacklisting is used for ignoring RREQs from nodes that have previously failed to receive or acknowledge RREPs correctly, e.g. due to a uni-directional link.

aodv_socket.{c, h}

This module contains the socket functionality of AODV-UU, responsible for handling AODV control messages. It maintains two separate message buffers; a *receive buffer* and a *send buffer*, used for storing an incoming or outgoing message until it has been processed or sent out. Each buffer holds only one (1) message at a time. This is not a problem, since packets are handled one by one, and hence safely can be discarded from the buffers after message processing or sending has completed.

The module offers creation of an AODV message (initialization of the send buffer), queueing of an AODV message (copying of message contents to the send buffer) and processing of a received packet. The packet processing function checks the type field of the packet, converts it to the appropriate message type and calls the correct handler function, e.g. `rrep_process()` of the aodv_rrep module if the message is a RREP message.

aodv_timeout.{c, h}

This module contains handler functions for timeouts, i.e., functions that are called when certain pre-determined events occur. The handler functions receive a pointer to the affected object, e.g. a routing table entry, as a parameter. Handler functions are defined for the following events:

- *Route deletion timeout:* This timeout occurs when a route has not been used for a certain period of time. This deletes the route from the internal routing table of AODV-UU.
- *Route discovery timeout:* This timeout occurs when route discovery was requested, but a route was not found within a certain amount of time. If the *expanding ring search* option is enabled, this results in a new route discovery with a larger TTL value than in the previous attempt. Otherwise the packet is dropped, an ICMP *Destination Host Unreachable* message is sent to the application and the destination is removed from the list of destinations for which AODV-UU is seeking routes.
- *Route expiry timeout:* This timeout occurs when the lifetime of a route has expired. This marks the route as down, creates a RERR message regarding unreachable destinations and sends the RERR message to affected nodes.
- *HELLO timeout:* This timeout occurs when HELLO messages from a node stop being received. This timeout occurs on an individual basis for each affected route, and is treated as a link failure, i.e., the route expiry timeout handler function is called.
- *RREQ record timeout:* This timeout occurs when a RREQ message has become old enough that subsequent RREQs from the affected node with a certain RREQ ID should not be considered as duplicates (i.e., being discarded if received) anymore.
- *RREQ blacklist timeout:* This timeout occurs when RREQs from a certain node should not be ignored anymore. Nodes end up in the blacklist by repeatedly sending RREQs for a certain destination, even though RREPs have been sent back to the requesting node. (The cause for this could e.g. be uni-directional links.) The RREQ blacklist timeout handler function removes a node from this blacklist so that normal operation is resumed.
- *RREP-ACK timeout:* If RREPs are sent over an unreliable or uni-directional link, a RREP-ACK can be requested by the sending node. If no RREP-ACK is received within a certain time period, this timeout occurs. The result is that the node that failed to reply with a RREP-ACK is added to the RREQ blacklist, described earlier.
- *Wait-on-reboot timeout:* This timeout occurs when the 15-second wait-on-reboot phase at startup has elapsed. It resumes active operation of the node, i.e., re-enables transmission of RREP messages.

debug.{c, h}

This module contains the logging functionality of AODV-UU. It offers logging of general events to a logfile as well as routing table logging to a routing table logfile. It also contains functions for converting IP addresses and AODV message flags to strings, for the purpose of printing and logging.

Logging of all general events is performed by a special logging function which examines the severity of the log message, checks the current log settings, and writes log messages to the appropriate logfiles. By default, log messages are also written to the console.

Routing table logging is performed by periodically dumping the contents of the routing table to a routing table logfile. Finally, debug-purpose logging is done through a special DEBUG macro in the source code. Such logging will only be effective if AODV-UU has been compiled with the DEBUG option set.

defs.h

This header file contains macros and datatypes used throughout AODV-UU, and hence, almost all the other modules include it. A brief listing of the contents can be found below.

- *AODV-UU version number:* Used by the main program (main.c) for displaying the program version number.
- *Logfile and routing logfile paths:* By default, these are set to /var/log/aodvd.log and /var/log/aodvd_rt.log.
- *A definition of infinity, and infinity checking:* Used for marking a route as down, by setting its corresponding hop count to this value. An infinity checking macro allows for easy testing of values against this infinity value.
- *Maximum number of interfaces:* The maximum number of network interfaces supported by AODV-UU.
- *A datatype for host information:* The `host_info` datatype contains information about a host, i.e., its latest used sequence number, latest time of broadcast, RREQ ID, Internet gateway mode setting, number of network interfaces attached to the host and an array of corresponding network devices.
- *A datatype for network devices:* The `dev_info` datatype contains information about a network device, i.e., its status, socket, index number (for addressing into the array of network devices), name, IP address, netmask and broadcast address.
- *Macros for retrieving network device information:* The macros DEV_IFINDEX(ifindex) and DEV_NR(n) are useful when network device information is needed.

  DEV_IFINDEX(ifindex) allows for retrieval of the network device information based on an interface index, as defined by the operating system. It translates this interface index into a network device number, and retrieves the network device information from the host information of the current host.

  DEV_NR(n) instead directly retrieves network device information from the host information of the current host; it indexes into the network device array and retrieves the network device information from there.
- *AODV message types:* The different message types used by AODV-UU (AODV_HELLO, AODV_RREQ, AODV_RREP, AODV_RERR and AODV_RREP_ACK) are associated with unique message type numbers.
- *The AODV message type:* This is the message type for AODV messages in AODV-UU. It contains a type field followed by space for type-specific data, i.e., enough space to hold any type of AODV message.

  All AODV messages in AODV-UU, such as RREQs, RREPs and RERRs, will be type casted to this general message type before they are sent out. Similarly, since the type field indicates the type of the message, received AODV messages can be type casted back to their corresponding “specialized” type and processed correctly.
- *Macros to access AODV extensions:* Macros used for accessing AODV message extensions, i.e., extra information included with AODV messages, such as neighbor set information.
- *A type definition for callback functions:* This type definition specifies that callback functions used for socket communication are arbitrary functions taking an integer as an argument (used for passing file descriptors to message handling functions).

icmp.{c, h}

This module contains functionality for sending an ICMP *Destination Host Unreachable* message to the application if route lookup fails for some destination.

main.c

This is the main program of AODV-UU. It handles initialization of the other modules, kernel module loading and host initialization, and uses `select()` to wait for incoming messages on sockets.

packet_input.{c, h}

This module handles communication with the AODV-UU kernel module, kaodv. All packets arriving from the kernel module, both incoming and outgoing packets, will be processed by the `packet_input()` function. However, AODV control messages will be ignored, since these are handled separately by socket communication in the aodv_socket module.

Route discovery and packet buffering is performed as needed. If a packet is destined for another node, and an active route to that node is available, the packet is forwarded using the next hop information from the routing table.

packet_queue.{c, h}

This module handles queueing (buffering) of packets. Each packet from Netfilter is associated with a unique packet ID, and these IDs are queued in a FIFO queue. Hence, the module performs a light-weight queueing of packets. As soon as AODV-UU has decided on an action for a packet, that packet can be removed from the queue. The packet ID together with a verdict is then returned through the libipq module to Netfilter, which carries out the requested action. The functionality of the packet_queue module allows individual packets to be added, sent or dropped, or all packets in the queue to be destroyed (dropped).

params.h

This header file contains the following constants, defined by the AODV draft [7]:

- *ACTIVE_ROUTE_TIMEOUT:* The lifetime of an active route.
- *TTL_START:* Initial TTL value to be used for RREQs.
- *DELETE_PERIOD:* Time to wait after expiry of a routing table entry before it is expunged, i.e., deleted.
- *ALLOWED_HELLO_LOSS:* Maximum loss of anticipated HELLO messages before the link to that node is considered to be broken.
- *BLACKLIST_TIMEOUT:* Timeout for nodes that are part of this node's RREQ "blacklist".
- *HELLO_INTERVAL:* Interval between broadcasts of HELLO messages.
- *LOCAL_ADD_TTL:* Used in the calculation of TTL values for RREQs during local repair.
- *MAX_REPAIR_TTL:* Maximum number of hops to a destination for which local repair is to be performed.
- *MY_ROUTE_TIMEOUT:* Time period for which a route, advertised in a RREP message, is valid.
- *NET_DIAMETER:* The maximum diameter of the network. This value will be used as an upper bound for RREQ TTL values.
- *NEXT_HOP_WAIT:* Time to wait for transmission by a next-hop node when attempting to utilize passive acknowledgements (i.e., overhearing of network traffic).
- *NODE_TRAVERSAL_TIME:* Conservative, estimated average of the one-hop traversal time for packets.
- *NET_TRAVERSAL_TIME:* Estimated time that it takes for a packet to traverse the network.

- *PATH_TRAVERSAL_TIME:* Time used for buffering RREQs and waiting for RREPs.
- *RREQ_RETRIES:* Maximum number of RREQ transmission retries to find a route.
- *TTL_INCREMENT:* Incrementation of the TTL value in each pass of *expanding ring search*.
- *TTL_THRESHOLD:* When the TTL value for RREQs in *expanding ring search* has passed this value, it should be set to NET_DIAMETER (instead of continuing the stepwise incrementation).
- *K:* This constant should be set according to characteristics of the underlying link layer. A node is assumed to invariably receive at least one out of K subsequent HELLO messages from a neighbor if the link is working and the neighbor is sending no other traffic.

routing_table.{c, h}

This module contains routing table functionality. It defines the datatypes for routing table entries and precursors, both containing pointers to another element of the same type. That way, entries can form a linked list. Each routing table entry contains the following information:

- Destination IP Address
- Destination Sequence Number
- Interface (network interface index)
- Hop Count
- Last Hop Count
- Next Hop
- A list of precursors
- Lifetime
- Routing Flags *(forward route, reverse route, neighbor, uni-directional* and *local repair)*
- A timer associated with the entry
- RREP-ACK timer for the destination
- HELLO timer
- Last HELLO time
- HELLO count
- A hash value (for quickly locating the entry)
- A pointer to subsequent routing table entries

The operations offered by this module are the initialization and cleanup of the routing table, route addition, route modification, route timeout modification, route lookup, active route lookup, route invalidation, route deletion, precursor addition and precursor removal. As routes are added, updated, invalidated or deleted, the kernel routing table of the system is updated accordingly. This is done through calls to kernel routing table functions of the k_route module.

seek_list.{c, h}

This module contains management of the seeking list, i.e., the list of destinations for which AODV-UU is seeking routes. The seeking list is a linked list of entries, each containing the following information:

- The Destination IP address
- The Destination Sequence Number
- Flags (used for resending RREQs)
- Number of RREQs issued
- The TTL value to use for RREQs
- IP data (for generating an ICMP *Destination Host Unreachable* message to the application if route discovery fails)
- A seeking timer
- A pointer to subsequent seeking list entries

Entries may be added to or removed from the seeking list as needed. Seeking list entries can also be searched for, by specifying their destination IP addresses.

timer_queue.{c, h}

This module contains the timer functionality of AODV-UU. It defines a datatype for timers, containing an expiry time, a pointer to a handler function, a pointer to data (used by the handler function), a boolean value indicating timer usage, and a pointer to other timers.

Timers are added to a *timer queue*, represented by a linked list. This list is sorted, with the timer that will expire first at the head of the list. The timer queue can be "aged", i.e., checked for expired timers. During aging, the handler functions of expired timers are called in sequence, with the specified data pointer (e.g. a pointer to a routing table entry) being passed as an argument. This allows the handler functions to be context-aware. Aging of the timer queue also updates the `select()` timeout used by AODV-UU while waiting for incoming messages on sockets. The new timeout of this `select()` timer is taken from the timer at the head of the timer queue.

4.5.2 Kernel-related modules

kaodv.c

This module is the kernel module of AODV-UU. It registers a packet handling function on three Netfilter hooks; NF_IP_PRE_ROUTING (for handling incoming packets prior to routing), NF_IP_LOCAL_OUT (for handling locally generated packets) and NF_IP_POST_ROUTING (for re-routing packets prior to sending them).

Packets arriving on the NF_IP_PRE_ROUTING or NF_IP_LOCAL_OUT hook are queued in user-space, to allow AODV-UU to process them. Packets arriving on the NF_IP_POST_ROUTING hook, i.e., packets that should be sent out by the system, are re-routed to ensure usage of the latest routing information available from the kernel routing table. (Recall that the kernel routing table may have changed as an effect of AODV-UU operation, e.g. route discoveries.)

k_route.{c, h}

This module contains functionality for modifying the kernel routing table of the system. Routes may be added, changed or deleted. The kernel routing table modifications are carried out by `ioctl()` calls.

libipq.{c, h}

This module, developed by the Netfilter core team, contains the functionality for user-space queueing of IP packets. It uses a *netlink socket* to communicate with Netfilter, and allows user-space packet handling callback functions to be called whenever a queued packet "arrives". AODV-UU uses the libipq module for receiving packets and returning verdicts for packets from user-space.

4.6 Packet handling

In this section, the packet handling of AODV-UU is described. Roughly, it distinguishes between data packets and AODV control messages, and handles them separately using different software modules. This is shown in Figure 4.2.

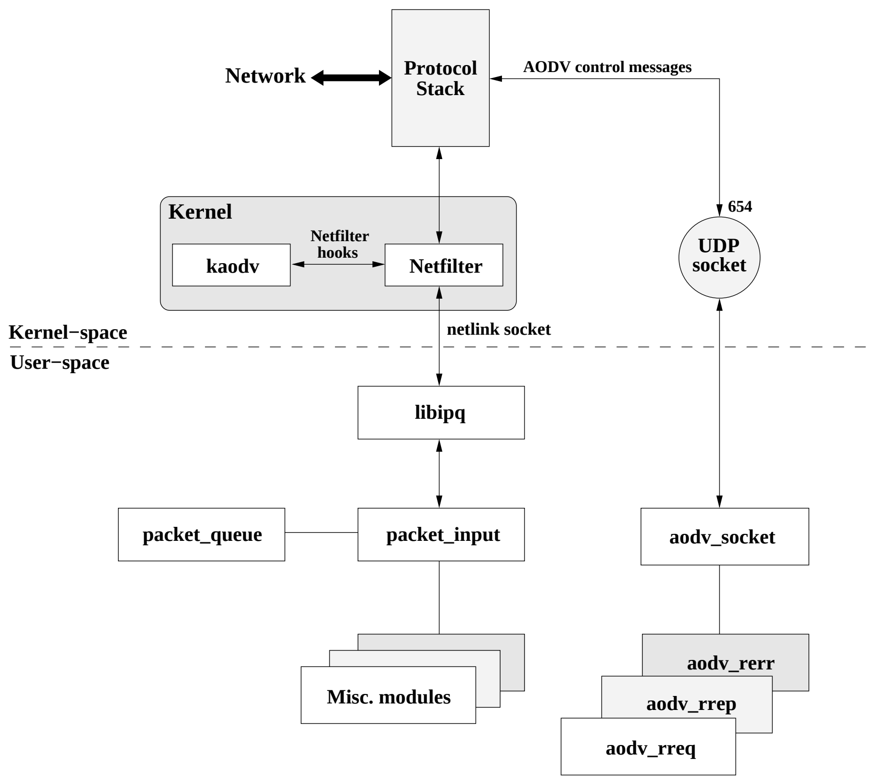


Figure 4.2: Packet handling of AODV-UU. Data packets and AODV control messages are handled separately.

4.6.1 Packet arrival

When a packet traverses the protocol stack, it is caught by the Netfilter hooks that have been set up by the AODV-UU kernel module, kaodv. The `nf_aodv_hook()` function of the kaodv module identifies the packet type, and either tells Netfilter to accept the packet (i.e., to let it through and allow the system to process it on its own) or to queue it (for further processing by AODV-UU in user-space).

Non-IP packets are always accepted, since these packets are of no interest to AODV-UU. Locally generated packets are always queued, since a route may have to be determined for those. Incoming AODV control messages are always accepted, since these eventually should be processed on a separate UDP socket and must be let through in order to be able to arrive there. Also, only packets on AODV-UU-enabled network interfaces should be processed by AODV-UU. Packets on other network interfaces are therefore immediately accepted, and not processed further.

4.6.2 Initial packet processing

Packet processing is performed by the `packet_input()` function of the packet_input module. If the packet is an AODV control message, an accept verdict is returned to the libipq module so that the packet eventually will end up on the AODV control message UDP socket, to be received or sent out, depending on whether the packet is an incoming or outgoing packet. Otherwise, the packet is analyzed further.

4.6.3 Data packet processing

If the destination of the packet (determined by its destination IP address) is the current host, the packet is a broadcast packet, or Internet gateway mode has been enabled and the packet is not a broadcast within the current subnet, the packet is accepted. This means that the packet under these circumstances will be handled as usual by the operating system.

Otherwise, the packet should either be forwarded, queued or dropped. The internal routing table of AODV-UU is used for checking whether an active route to the specified destination exists or not. If such a route exists, the next hop of the packet is set and the packet is forwarded. Otherwise, provided that the packet was generated locally, the unique packet ID provided by the libipq module is used by the packet_queue module for indirectly queueing the packet until AODV-UU has decided on an action, and a route discovery is initiated. If the packet was not generated locally, and no route was found, it is instead dropped and a RERR message is sent to the source of the packet.

4.6.4 AODV control message processing

AODV control messages are received on a UDP socket (on port 654) and processed by the aodv_socket module. The type field of the AODV message is checked, the message is converted to the corresponding specialized message type, and the correct handler function is called in the appropriate module.

4.6.5 AODV control message sending

Each AODV control message generated by AODV-UU is sent out on the AODV control message UDP socket. Such a message will be caught by the Netfilter hook for locally generated packets, NF_IP_LOCAL_OUT, queued by the kaodv module and received by the packet_input module of AODV-UU through libipq. The packet_input module will return an accept verdict to libipq, and the packet will then be caught by the post-routing Netfilter hook, NF_IP_POST_ROUTING. The packet is re-routed to ensure usage of the most recent routing information, and sent out by the system.

4.7 Current status

AODV-UU has successfully been tested together with other AODV implementations during an AODV interop [39] in March 2002, with good results. These results can be attributed to the large amount of dedicated work that has been put into the development of AODV-UU. Currently (September 2002), AODV-UU is undergoing some minor changes to adapt it to the very latest AODV draft (version 11) [7]. A new version of AODV-UU (version 0.6) is expected to appear soon, and hopefully, a technical report on AODV-UU will also be prepared in the near future.

Chapter 5

The ns-2 Network Simulator

5.1 Introduction

The network simulator *ns-2* [8] is an object-oriented, discrete event-driven network simulator developed at UC Berkeley and USC ISI as part of the VINT project [41]. It is a very useful tool for conducting networks simulations involving local and wide area networks, but its functionality has grown during recent years to include wireless networks and ad-hoc networking as well.

The ns-2 network simulator has gained an enormous popularity among participants of the research community, mainly because of its simplicity and modularity. It allows *simulation scripts*, also called *simulation scenarios*, to be easily written in a script-like programming language, OTcl. More complex functionality relies on C++ code that either comes with ns-2 or is supplied by the user. This flexibility makes it easy to enhance the simulation environment as needed, although most common parts are already built-in, such as wired nodes, mobile nodes, links, queues, agents (protocols) and applications. Most network components can be configured in detail, and models for traffic patterns and errors can be applied to a simulation to increase its reality. There even exists an emulation feature, allowing the simulator to interact with a real network.

Simulations in ns-2 can be logged to *trace files*, which include detailed information about packets in the simulation and allow for post-run processing with some analysis tool. It is also possible to let ns-2 generate a special trace file that can be used by *NAM (Network Animator)*, a visualization tool that is part of the ns-2 distribution. This allows simulations to be replayed on screen, which can be useful for complex simulations.

A large amount of documentation exists for ns-2, including a reference manual [40], several tutorials such as [56], and an *ns-users* mailing list [57]. The intention of this chapter is to provide an overview of the ns-2 network simulator and to illustrate the usage of some of its network components – in particular those that are important for the porting of AODV-UU described in Chapter 6. For this reason, the sections on agents and mobile networking have been given a substantial amount of space. Finally, the descriptions of trace file formats in section 5.18.2 complement the almost non-existent documentation on this topic found in the ns-2 manual.

The version of ns-2 described here is version 2.1b9, and the ns-2 distribution used is the ns-allinone distribution, version 2.1b9. The notion ˜ns followed by a path and a filename refers to the corresponding file in the ns-2 source code tree. The complete source code of ns-2 is available from the ns-2 homepage [8].

5.2 Performing simulations

The starting point for performing simulations in ns-2 is to build an OTcl simulation scenario script file that specifies the components to be used and the events that should occur. An example scenario could e.g. set up a network topology consisting of two nodes, connect these two using a 10 Mbps duplex link, set up FTP traffic over TCP, and start and stop this traffic at certain points in time.

5.2.1 Main building blocks

In general, a simulation scenario consists of three main components:

- A network topology
- Connections, traffic and agents (protocols)
- Events and failures

A *network topology* defines the number of nodes and their connectivity, and can either be created manually or with special topology generators such as GT-ITM [42]. *Connections* and *traffic* are set up by traffic generators and agents (protocols) at a node. *Events* and *failures* include connection set-ups/tear-downs, packet flow, packet loss, congestion and mobile node movements.

5.2.2 Simulation scenario contents

The contents of most simulation scenario scripts follow a certain pattern. This is due to the fact that some general setup is common to almost all scenarios. A typical simulation scenario script will include (in order):

- *Creation of a simulator instance.* A simulator instance is needed for any simulation to be performed.
- *Opening of trace files*, both a “normal” trace file and a NAM (Network Animator) trace file.
- *Configuration of nodes*, i.e., setting of parameters that will be used when nodes are created.
- *Creation of nodes and links*, and connection of nodes using these links.
- *Creation of agents and applications*, and attachment (installation) of these.
- *Scheduling of traffic-related events*, e.g. starting or stopping of a traffic generator or agent.
- A *finish procedure* to be called at the end of simulation. This procedure should flush any trace file output and exit the simulator.
- *Scheduling of a call to the finish procedure* at the end of the simulation.
- *Starting of the simulation*, by issuing a `run` command to the simulator instance.

5.2.3 Simulation execution and analysis

A simulation scenario script is executed (i.e., a simulation is performed) by supplying the file name on the command line to the ns-2 simulator. The simulator acts as an OTcl interpreter, interpreting the simulation scenario script line by line. Any error messages or messages generated by the script will be printed to the console. When the simulation has finished, the simulator exits and the command shell prompt returns. No graphical user interface is supplied with ns-2 for performing simulations.

After successfully performing a simulation, the trace files that may have been produced by the simulation scenario script can be analyzed. Depending on the objectives of the simulation, this can either be done with a full-fledged analysis tool such as Trace graph [43] or with simpler, hand-made scripts (usually Perl, `sed` or `awk` scripts) or programs.

5.3 The OTcl/C++ environment

To increase flexibility and efficiency, ns-2 uses *two* programming languages for its operation; C++ and OTcl. C++ is mainly used for event handling and per-packet processing; tasks for which OTcl would become too slow. OTcl is commonly used for simpler routing protocols, general ns-2 code and simulation scenario scripts. The usage of OTcl for simulation scenario scripts allows the user to change parameters of a simulation without having to recompile any source code.

The two programming languages are tied together in the sense that C++ objects can be made available to the OTcl environment (and vice versa) through an OTcl linkage. This linkage creates OTcl objects for C++ objects and allows variables of C++ objects to be shared as well. In addition, it offers access to the OTcl interpreter from C++ code. This makes it possible to implement network components in OTcl, C++ or both. Furthermore, these components can easily be configured from the simulation scenario script because of the OTcl linkage, so the choice of programming language used for the implementation is completely transparent to the user.

5.4 Class hierarchy

To provide a better understanding of the characteristics of network components in ns-2, a partial ns-2 class hierarchy is shown in Figure 5.1.

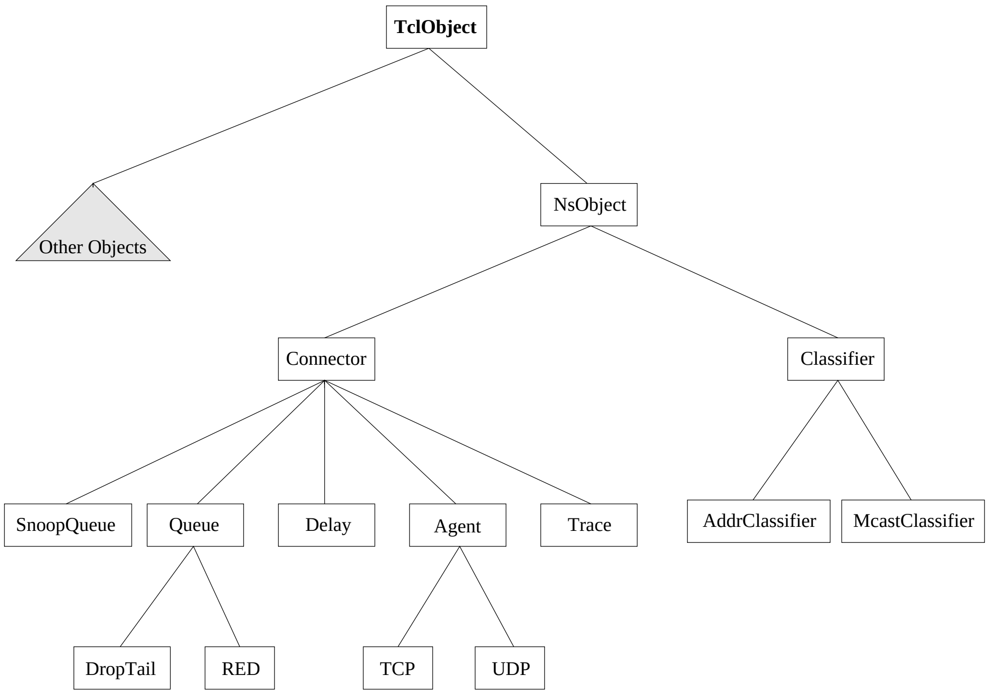


Figure 5.1: Partial ns-2 class hierarchy

The root of the hierarchy is the `TclObject` class, which is the superclass of all library objects in OTcl, i.e., schedulers, network components, timers and other objects. The `NsObject` class forms a superclass for all basic network component objects that handle packets. Basic network components may be *compound* to form more complex network objects such as nodes and links.

The basic network components are divided into two subclasses, `Connector` and `Classifier`, based on the *number of possible output data paths*. Basic network components with only one output data path are placed under the `Connector` class, while switching objects that have multiple possible output data paths instead are placed under the `Classifier` class.

5.5 Event scheduling

Event scheduling in ns-2 is handled by a *discrete event scheduler*, used e.g. by simulation scenario scripts and network components that simulate packet-handling delays or use timers for their operation. There are two different types of event schedulers; *real-time* and *non-real-time* schedulers.

The real-time scheduler `RealTime` is used for *emulation*, i.e., live interaction between the simulator and a real network. For non-real-time purposes, the three schedulers `List`, `Heap` and `Calendar` are available, of which the `Calendar` scheduler is the default. These mainly differ in the way that they store events, yielding different time complexities.

An event scheduler keeps track of the time, an event ID and a handler for each event. When an event should be carried out, the specified handler is called, as shown in Figure 5.2. Handlers in this context are handler methods of C++ objects of the `Handler` class. Events are executed to completion one by one, i.e., no pre-emption is supported. If several events are scheduled to occur at the same time, they are executed in a first scheduled - first dispatched manner.

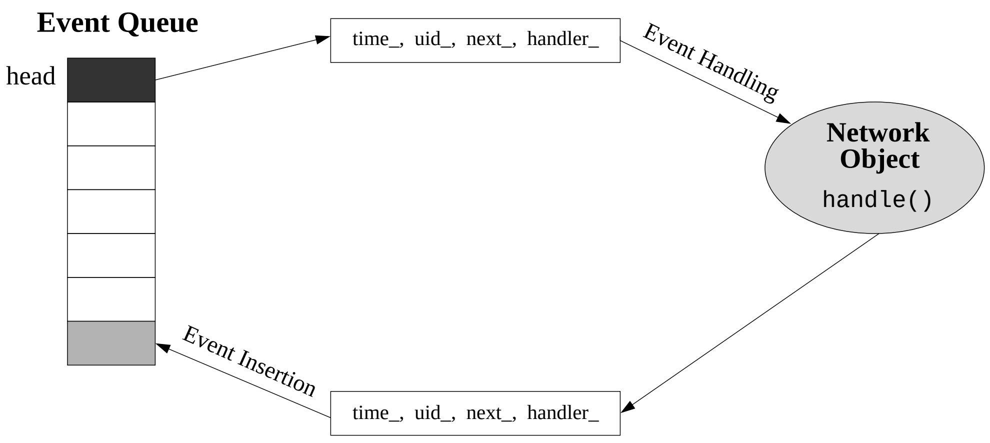
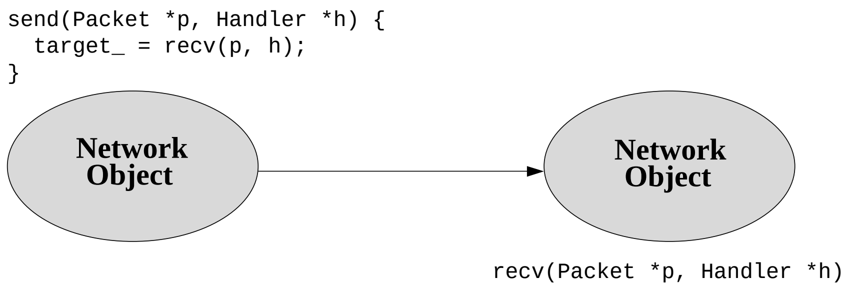


Figure 5.2: Discrete event scheduling in ns-2. The event scheduler calls the handler function of a network component at the specified time of the event.

Event scheduling is indirectly made available to network components of ns-2 through the use of *timers*, described in section 5.12, or through access to an instance of the OTcl interpreter, as mentioned in section 5.3. Simulation scenario scripts may schedule events through the simulator instance by using the `at <time> "<OTcl command>"` command.

5.6 Packet delivery

Packet delivery in ns-2 is built on the concept of network objects (components) interacting with each other. Packets are generated e.g. by agents or traffic generators, and delivered through links from one node to another. In each step of the packet delivery process, a network object sends the packet using a *send* method, which invokes the *recv* (receive) method of another network object. This is shown in Figure 5.3.



```
send(Packet *p, Handler *h) {
  target_ = recv(p, h);
}
```

Figure 5.3: Packet delivery in ns-2. A network object sends a packet to another network object by invoking its receive method.

For packet delivery to take place, a reference to the receiving network object is needed. This, however, is not a problem. All plumbing-work involving network objects is performed in the simulation scenario script, and hence, all references to network objects are known.

5.7 Nodes

Nodes are fundamental in a simulation. They perform processing and forwarding of packets, and are therefore perhaps the most important entities among all network components of ns-2. This section mainly deals with unicast nodes that have a wired connection to the network; wireless nodes are described in section 5.15, and details on multicast nodes can be found in [40].

5.7.1 Node basics

A wired unicast node is a compound object composed of a node entry object and two classifiers, as shown in Figure 5.4. The *node entry* is where packets first arrive. The *address classifier* examines the address field of a packet to determine whether the packet is destined for the current node. Finally, the *port classifier* determines which agent (protocol) at the node that should receive the packet.

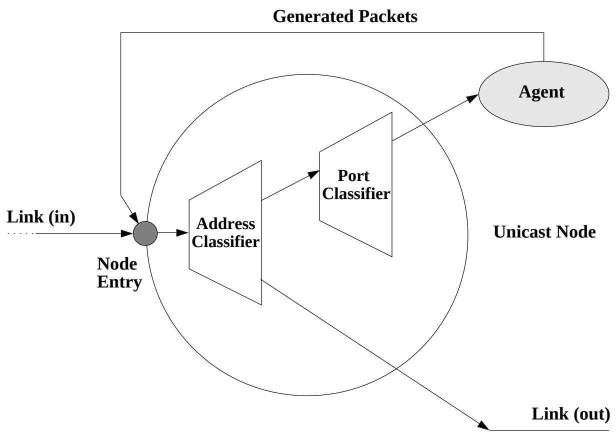


Figure 5.4: Composite construction of a wired unicast node

If the packet is not destined for the current node, the address classifier determines which node that the packet should be forwarded to. This is possible because of routing functionality in the node constantly updating the address classifier. Packets are forwarded to other nodes through links to these nodes.

5.7.2 Node confi guration and creation

Prior to creating nodes, the desired node configuration must be given to the simulator instance. This is done by issuing a `node-config` command to the simulator instance, and passing the configuration options as arguments. Table 5.1 lists some of the available options for node configuration.

If no configuration is made, default values apply, i.e., it is assumed that wired nodes with a flat addressing scheme (where nodes are simply numbered from zero and up) should be used. To illustrate the `node-config` command, an example configuration of a wireless ad-hoc node is shown below:

```
$ns_ node-config -addressType hierarchical \
                 -adhocRouting AODV \
                 -llType LL \
                 -macType Mac/802_11 \
                 -ifqType Queue/DropTail/PriQueue \
                 -ifqLen 50 \
                 -antType Antenna/OmniAntenna \
                 -propType Propagation/TwoRayGround \
                 -phyType Phy/WirelessPhy \
                 -topoInstance $topo \
```

Table 5.1: Selected options for node configuration. A dash (-) indicates that the value is not set by default.

Option	**Available values**	**Default value**
-addressType	flat, hierarchical	flat
-wiredRouting	ON, OFF	OFF
-llType	LL, LL/Sat	-
-macType	Mac/802_11, Mac/Csma/Ca, Mac/Sat, Mac/Sat/UnslottedAloha, Mac/Tdma	-
-ifqType	Queue/DropTail, Queue/DropTail/PriQueue	-
-ifqLen	<length>	-
-phyType	Phy/WirelessPhy, Phy/Sat	-
-adhocRouting	DIFFUSION/RATE, DIFFUSION/PROB, DSDV, DSR, FLOODING, OMNICAST, AODV, TORA	-
-propType	Propagation/TwoRayGround, Propagation/Shadowing, Propagation/FreeSpace	-
-propInstance	Propagation/TwoRayGround, Propagation/Shadowing	-
-antType	Antenna/OmniAntenna	-
-channel	Channel/WirelessChannel, Channel/Sat	-
-topoInstance	<topology file>	-
-mobileIP	ON, OFF	OFF
-rxPower	<value in W>	-
-txPower	<value in W>	-
-idlePower	<value in W>	-
-agentTrace	ON, OFF	OFF
-routerTrace	ON, OFF	OFF
-macTrace	ON, OFF	OFF
-movementTrace	ON, OFF	OFF
-IncomingErrProc	<error model instantiator>	-
-OutgoingErrProc	<error model instantiator>	-

```
                         -channel Channel/WirelessChannel \
                         -agentTrace ON \
                         -routerTrace ON \
                         -macTrace ON \
                         -movementTrace ON
```

After performing node configuration, nodes may be created by issuing a `node` command to the simulator instance. This command creates a node using the current node configuration options. If nodes with different characteristics are needed, the node configuration procedure may be repeated, and more nodes created.

5.7.3 Node addressing

Node addresses in ns-2 consist of two parts; *node IDs* and *port IDs*. Both are 32-bit fields, which together constitute a complete address. Two addressing modes are available that determine how addresses should be interpreted. In *flat addressing*, each node gets assigned a node ID at the time of creation, starting from zero and increasing. This means that the addressing scheme simply becomes a node numbering scheme without any further interpretation of addresses. In *hierarchical addressing,* the node ID is instead divided into different hierarchy levels, with a specific number of bits used for each level. The node addressing mode is configured using the `node-config` command of the simulator instance

5.7.4 Routing

Routing in nodes is performed by *routing modules*, consisting of a *routing agent*, *route logic* and *classifiers*. Routing agents exchange routing packets with neighbors, route logic uses the information gathered by routing agents to perform the computation of a route and classifiers use the computed routing table to carry out the actual packet forwarding.

A routing module initializes its connection to a node by registering itself to the node with a special registration command. During this registration, the routing module tells the node whether it is interested in knowing about route updates and transport agent attachments, and creates and installs classifiers inside the node. Currently, there are six routing modules available. These are listed in Table 5.2.

Table 5.2: Available routing modules

Module name	**Offered functionality**
RtModule/Base	Interface to unicast routing protocols. Provides basic functionality to add/delete routes and attach/detach agents.
RtModule/Mcast	Interface to multicast routing protocols. Establishes multicast classifiers.
RtModule/Hier	Hierarchical routing. A wrapper for managing hierarchical classifiers and route installation.
RtModule/Manual	Manual routing, i.e., manual addition and deletion of routes.
RtModule/VC	Uses a virtual classifier instead of a “vanilla” classifier for forwarding packets.
RtModule/MPLS	Implements MPLS (Multi-Protocol Label Switching) routing.

5.8 Links

A link is a compound object composed of a sequence of *connectors,* as shown in Figure 5.5. Connectors, unlike classifiers, only generate data for one recipient; either the packet is delivered to the *target* (output) of the connector, or it is dropped to a *drop target*.

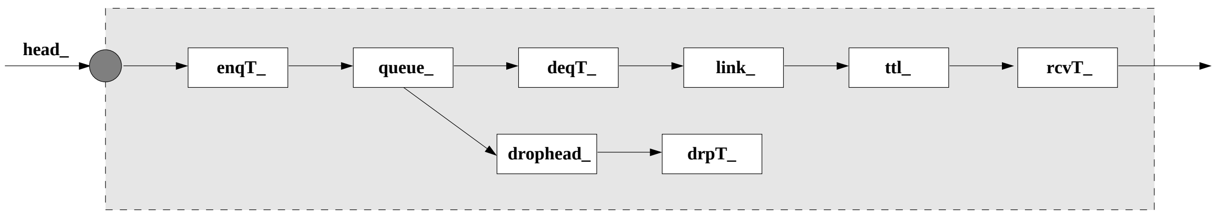


Figure 5.5: Composite construction of a uni-directional link

Links are defined by five instance variables. `head_` is the entry point to the link, pointing to the first object in the link. `queue_` is a reference to the main queue element of the link. Although simple links only contain one queue, more complex links may have multiple queues. `link_` is a reference to the element that actually models the link, giving it its delay and bandwidth characteristics, and `ttl_` is a reference to the element that manipulates the TTL field in every packet. Finally, `drophead_` is a reference to an object that is the head of a queue of elements that process link drops.

Links also allow packets to be monitored more closely. By using the `trace-all` command of the simulator instance (see section 5.18.1), trace elements `enqT_` and `deqT_` are added to track when a packet is enqueued or dequeued from queue_. Also, `drpT_` allows dropped packets to be traced, and `rcvT_` allows packets to be traced at the end of a link.

Creation and configuration of links is provided by the simulator instance. By issuing a `simplex-link` command to the simulator instance, a simplex link is created between two nodes with certain bandwidth, delay and queue type characteristics. Similarly, duplex links are created with the `duplex-link` command. To model connectivity changes, links offer a dynamic mode that is selected by issuing a `dynamic` command. After this command has been issued to a link, the status of that link can be changed by issuing `up` and `down` commands. In dynamic mode, links also offer an `up?` command for checking the status of a link.

The cost for a packet to traverse a link, which defaults to 1, can be set using the `cost` command. This value is used by the route logic of routing modules when calculating routes. Other useful commands for links include `errormodule`, which inserts an error model before the queue element of a link to model errors in packets, and `insert-linkloss`, which inserts an error model after the queue element of a link to model link loss. Error models and their installation is described further in section 5.14.

5.9 Queues

Queues are objects that hold or drop packets, and are used e.g. in links and interface queues in wireless networking. Currently, ns-2 includes support for drop-tail (FIFO) queueing, RED buffer management, class-based queueing and several variants of fair queueing. Some of the available queue types are listed below, along with their respective OTcl class names.

- `Queue/DropTail` queues implement simple *FIFO queues*. A subclass of this queue type, `PriQueue`, allows packets to be prioritized based on their packet type.
- `Queue/FQ` queues implement *fair queueing*.
- `Queue/SFQ` queues implement *stochastic fair queueing*.
- `Queue/DRR` queues implement *deficit round robin scheduling*, and support multiple flows.
- `Queue/RED` queues implement *random early-detection gateways*, which can be configured to either mark or drop packets.
- `Queue/CBQ` queues implement *class-based queueing*, in which packets can be associated with traffic classes based on their IP header flow IDs.

5.10 Packets

Packets are the fundamental unit of exchange between objects in a simulation. They are built up of *packet headers*, corresponding to different protocols that may be used, and *packet data*. Access to the different packet headers and the data portion of a packet is made available through access methods. This is shown in Figure 5.6. New protocols may add their own packet header types to the available ones, and unused packet headers may be turned off to save memory during simulations.

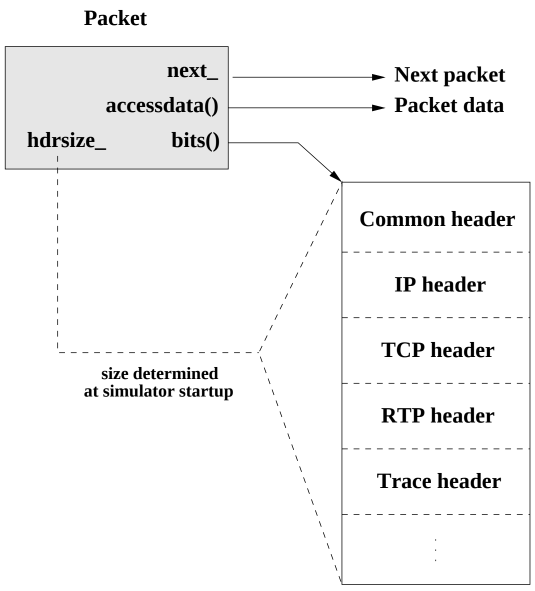


Figure 5.6: Packet contents. Each packet is built of packet headers and packet data.

5.10.1 Packet types and headers

Packet types are defined in ˜ns/common/packet.h. An enumeration of packet types is provided by the `packet_t` enumeration, and a `p_info` class provides the mapping to packet type names:

```
enum packet_t {
  PT_TCP,
  ...
  PT_NTYPE
};

class p_info {
public:
  p_info() {
    name_[PT_TCP] = "tcp";
    ...
    name_[PT_NTYPE]= "undefined";
  }
}
```

At the beginning of a simulation, the available packet types are reviewed by the `PacketHeaderManager` OTcl class to assign each packet type a unique offset in packets. This offset is stored in a static `offset_` variable of each packet header class, and is used by the `access()` methods of these classes to provide a pointer into the packet where that packet header begins. Usually, packet header classes also supply access macros to simplify packet header access even further.

Unused packet headers may selectively be disabled by issuing a `remove-packet-header` command in OTcl *before* any simulator instance is created. For large simulations with lots of traffic, this could save a considerable amount of resources (most notably, memory).

5.10.2 The common header

The only mandatory packet header in ns-2 is the *common header*, `hdr_cmn`, mainly used for tracing packets and measuring other quantities during a simulation. The most important fields of this header are:

- `ptype_`, the packet type.
- `uid_`, a unique packet ID.
- `size_`, the simulated packet size.
- `ts_`, the timestamp of the packet. This field is used for queue delay measurements.
- `error_`, an error flag indicating whether the packet is corrupted or not.
- `errbitcnt_`, the number of corrupted bits in the packet.
- `direction_`, the direction of the packet during network stack traversal.
- `prev_hop_`, the address of the previous hop. Used for hop-by-hop routing in wireless simulations.
- `next_hop_`, the address of the next hop. Used for hop-by-hop routing in wireless simulations.
- `num_forwards_`, the number of times that a packet has been forwarded (used in wireless simulations).
- `iface_`, the label of the receiving interface.

The `ptype_` field is used for packet type identification in trace logs, making them easier to read. The `uid_` field is used by the scheduler for scheduling packet arrivals. The `size_` field specifies the size of a simulated packet in bytes. It is used for computing the time it takes for a packet to be delivered over a link, and should therefore be set to be the sum of the sizes of application data, IP-, transport- and application-level headers. The `error_` and `errbitcnt_` fields are used for indicating the degree of packet corruption. The `direction_` and `next_hop_` fields are particularly important in wireless simulations. Finally, the `iface_` field is used by the simulator when performing computations of multicast distribution trees, and indicates on which link a packet was received.

5.10.3 Packet allocation and de-allocation

Packets are allocated on demand by calling the `alloc()` method of the `Packet` class. This reserves memory for a new packet, and returns a pointer to the packet. However, when packets are de-allocated using the `free()` method of the `Packet` class, they are not removed from memory but instead stored on a private list of free packets, to be reused when subsequent packet allocations are requested. This avoids memory fragmentation.

5.11 Agents

Agents represent endpoints where packets are generated or consumed, and are used for implementing protocols at various layers. *Routing agents* and *traffic sinks* are some examples of agents that are frequently used in simulations. The OTcl `class Agent` and the C++ `Agent` class together implement agents in ns-2. The source code can be found in ˜ns/common/agent.{cc, h} and ˜ns/tcl/lib/ns-agent.tcl. Here, we will consider C++ agents only, since OTcl packet handling generally is not recommended (due to performance issues).

5.11.1 Agent state information

Each agent holds a certain amount of *state information*, mainly to be able to assign default values to packet fields when generating packets, and to identify itself. Table 5.3 shows the state information kept for each agent.

Table 5.3: Agent state information

Variable	**Description**
agent_addr_	Address of this agent
agent_port_	Port number of this agent
dst_addr_	Address of destination agent
dst_port_	Port number of destination agent
type_	Packet type
size_	Packet size (in bytes)
ttl_	Default IP TTL value
fid_	IP flow identifier
prio_	IP priority field
flags_	Packet flags

5.11.2 Agent packet methods

Agents offer two methods for generating packets, `Packet *allocpkt()` and `Packet *allocpkt(int)`. These methods allocate space for a new packet, and assign default values to its packet fields using the state information of the agent. The latter method also makes room for a data payload.

Packets may be sent out with the `void send(Packet *p, Handler *h)` method, which hands them to the default target of the agent. Usually, this is the node entry of the node that the agent is attached to. Packet reception is available through a `void recv(Packet *p, Handler *h)` method, which will receive all packets destined for the agent. The handler argument is usually ignored by agents.

5.11.3 Agent attachment

Agents are attached to nodes by issuing an `attach-agent <node> <agent>` command to the simulator instance. This installs an agent in the port classifier of the node, using the agent's designated port number, `agent_port_`. After attaching an agent to a node, the agent will receive all packets destined for it, i.e., all packets whose port number matches the port number of the agent. The delivery of packets to agents is carried out by the port classifier of a node.

Routing agents are somewhat special, since they require more steps to be installed in a node. For instance, routing agents should usually be installed as the default target of the address classifier of a node to perform forwarding of packets. Such installation details are taken care of by existing OTcl support code (e.g. during node creation), and are not described here.

5.11.4 Connecting agents

To allow two agents attached to different nodes to communicate, the simulator instance offers a `connect <src> <dst>` command for connecting them with each other. This command sets the destination address and destination port of one agent to be the address and port of the other agent (and vice versa). This arranges for communication between the two agents. However, the nodes to which the agents are attached need also be connected to each other, e.g. by a duplex link.

5.11.5 Application attachment

Agents allow *applications* to be attached on top of them. For this purpose, ns-2 offers a simple API for interaction between agents and applications. However, this API lacks support for passing data between applications; it only offers *notifications* of received data and completed data transmission to the application. Hence, the application API is only used by relatively simple applications, such as traffic generators, FTP applications and Telnet applications. This API shortcoming has also resulted in agents partially taking over the role of applications. That is, when custom packet processing functionality is to be implemented in a node, that functionality is often put into an agent.

5.11.6 Creating a new agent

When creating a new agent, e.g. a routing agent, a number of steps have to be followed in order to make the agent available to the ns-2 environment. The following steps are fundamental:

- The inheritance structure of the agent should be decided on. The `Agent` class forms a foundation for agents, but multiple inheritance may be desired, depending on the functionality of the agent.
- The `void recv(Packet *p, Handler *h)` procedure for packet reception should be defined.
- Any necessary timer classes should be defined if the agent utilizes timers for its operation. The `void timeout(int tno)` method of the `Agent` class may be overridden to provide a custom timeout method; it is empty and unused by default. More information on timers can be found in section 5.12.
- OTcl linkage functions should be defined to allow agent usage from the OTcl environment of ns-2.
- The source code of the agent must be incorporated into the source code tree of ns-2, and ns-2 recompiled for the changes to take effect.

OTcl linkage

Linkage of the agent to the OTcl environment is performed in three steps. First, a mapping between the OTcl name space and the name of the agent class must be established. This is done by creating a static `TclClass` class instance whose constructor registers the name of the agent class with OTcl, and allows for creation of the agent:

```
static class ExampleAgentClass : public TclClass {
public:
  ExampleAgentClass() : TclClass("Agent/Example") {}
  TclObject *create(int argc, const char*const* argv) {
    return (new ExampleAgent());
  }
} class_example;
```

This allows the agent to be instantiated from OTcl. Next, binding of variables should be performed to allow these to be accessed from the OTcl environment. This is done by using the `bind()` method:

```
ExampleAgent::ExampleAgent() : Agent(PT_EXAMPLE) {
  bind("packetSize_", &size_);
}
```

In this example, the `size_` variable of the `ExampleAgent` class is bound so that it becomes accessible from OTcl as `packetSize_`. Values of bound variables are automatically kept consistent between the C++ and OTcl environment. The argument to the constructor of the `Agent` base class is the packet type that should used by the agent when generating packets.

Finally, the agent should be capable of receiving commands from the OTcl environment, e.g. to start or stop the agent or to modify its behavior in other ways. This is done by overriding the `Agent::command()` method:

```
int ExampleAgent::command(int argc, const char*const* argv)
{
  if (argc == 2) {
    if (strcmp(argv[1], "start") == 0) {
      startAgent();
      return TCL_OK;
    }
  }

  // Unknown command
  return (Agent::command(argc, argv));
}
```

Here, three things should be noted:

- The number of arguments passed to the `command()` method is one more than the number of arguments passed to the agent instance by the OTcl simulation scenario script. If the agent is given a `start` command by issuing `$agent start` from OTcl, the `command()` method of the agent will receive not one but *two* arguments, of which the first one is “cmd”. This must be considered when checking the argument count and extracting parameters from the argument array.
- A special return value, `TCL_OK` or `TCL_ERROR`, must be returned by the `command()` method. This provides the OTcl interpreter with information on the status of the command. If `TCL_OK` is returned, execution of the OTcl simulation scenario script continues as usual, but if `TCL_ERROR` is returned, the simulator will exit with an error message pointing out where things went wrong.
- If the agent cannot handle the command given to it through the `command()` method, it should pass the argument count and arguments on to the `Agent` base class for processing. The agent should return the return value of the `Agent::command()` method.

Compilation

For the agent to become available to ns-2, its source code must be integrated into the ns-2 source code tree, and ns-2 recompiled. This is preferrably done by modifying the Makefile.in file of ns-2, which serves as a template for the `configure` tool of the system when it creates a Makefile for compilation on that specific system. The target object (.o) files of the agent should be added among the other compilation prerequisites, so that the agent is compiled together with the rest of the ns-2 source code.

Finally, `configure` should be run from the ns-2 directory with Makefile.in in it to produce a new Makefile. Running `make` should then recompile the necessary parts of the ns-2 source code tree, and include support for the newly created agent.

5.12 Timers

Timers are used by agents and other objects in ns-2 for keeping track of delays and performing periodic executions of program code. They may be implemented in either C++ or OTcl, and rely on the scheduler of ns-2 for their notion of time. In this section, we will consider C++ timers only. The source code can be found in ˜ns/common/timer-handler.{cc, h}.

For C++ timers, the abstract base class `TimerHandler` should be used. A `void expire(Event *e)` method must be defined by subclasses, containing the code to be performed when the timer expires. Additional tweaking of timer behavior is available by overriding the `void handle(Event *e)` method, which normally consumes the timer event, calls `expire()`, and sets the status of the timer appropriately. Timers of the `TimerHandler` class offer the following functionality through public member functions:

- `void sched(double delay)` schedules the timer to expire `delay` seconds in the future.
- `void resched(double delay)` reschedules the timer to expire `delay` seconds in the future. In contrast to the `sched()` method, the timer may be pending.
- `void cancel()` cancels the timer (if it is pending).
- `int status()` returns the current status of the timer, i.e., one of TIMER_IDLE, TIMER_PENDING and TIMER_HANDLING.

Since timers are frequently used by agents, it is very common to see constructors of timer classes taking a pointer to an agent as an argument. This pointer can be stored by the timer and later dereferenced to gain access to methods of that agent class. Similarly, timer classes are often made friends of an agent class to allow them to access protected methods which otherwise would be inaccessible.

5.13 Traffic generators

Traffic generators are special applications that generate simulated network traffic. They should be attached to a transport agent, e.g. a TCP agent, for sending the generated traffic. Currently, four traffic generators are available:

- `Application/Traffic/Exponential` generates bursts of traffic, with burst and idle times of exponential distributions.
- `Application/Traffic/Pareto` generates bursts of traffic, with burst and idle times of pareto distributions.
- `Application/Traffic/CBR` generates constant bit-rate traffic, allowing the bit rate and packet size to be configured.
- `Application/Traffic/Trace` is used for generating traffic based on trace files from existing simulations. This traffic generator selects a random starting place in the trace file, from which it begins to generate traffic.

5.14 Error models

Error models are *connectors*, used for introducing link-level packet errors or packet loss into a simulation. Errors are modeled by setting the `error_` flag of packets, and packet loss is modeled by dropping packets to a drop target. Some of the most common error models are described below. The source code can be found in ˜ns/queue/errmodel.{cc, h}.

- *ErrorModel:* This is the base error model. It allows the unit of error to be changed to *bits* or *packets*, and the random variable for error probability to be specified. If a drop target has been set, it will receive all corrupted packets. Otherwise corrupted packets are passed on to the target of the error model, allowing errors to be handled at the receiving node.
- *Multi-state error model:* This error model uses state transitions for changing between different error models. Such transitions occur according to a transition state model matrix, specifying the probabilities of switching from one error model to another.

- *Two-state error model:* This error model introduces errors (or not), depending on the current state of the error model.
- *List error model:* This error model drops packets or bytes based on their sequence numbers.
- *Periodic error model:* This error model switches between errors and no errors periodically.
- *SelectErrorModel:* This error model drops packets of a certain type when the unique ID of a packet has a value with a certain offset from a cycle value.

5.14.1 Usage in wired networks

To use an error model in a wired network, it has to be inserted into a simple link. Since simple links are compound objects, error models may be inserted at several different places:

- *Before the queue of the link.* This can be done either with the `errormodule <errmod>` method of a simple link, which inserts the error model right before the queue of the link and sets the drop target of the error model to be the drop target of the link, or with the `lossmodel <errmod> <src> <dst>` method of the simulator instance, which inserts the error model into the simple link between the specified source and destination.
- *After the queue of a link, but before the delay element.* This can be done either with the `insert-linkloss <args>` method of a simple link or the `link-lossmodel <errmod> <src> <dst>` method of the simulator instance. These two methods work just like the `errormodule` and `lossmodel` methods described above, except for the placement of the error model.
- *After the delay element of a link.* This can be done with the `install-error` method of the `Link` class. This currently does not produce any trace, but is included for future extensions.

5.14.2 Usage in wireless networks

In wireless networks (described in section 5.15), error models can be applied both to incoming and outgoing network traffic on a wireless channel. The error models are placed between the network interface and the MAC layer of the network stack, and allow incoming and outgoing packets to be corrupted independently. Installation of error models in a wireless node is performed through the `node-config` command of the simulator instance, as described in section 5.7.2. More specifically, the parameters `-IncomingErrProc` and `-OutgoingErrProc` to the `node-config` command should specify OTcl methods that return error model instances, as shown below.

```
proc ExampleErrProc{} {
  set err [new ErrorModel]
  $err set rate_ 0.01
  $err drop-target [new Agent/Null]
  return $err
}

$ns_ node-config -IncomingErrProc ExampleErrProc \
                 -OutgoingErrProc ExampleErrProc

set node1_ [$ns_ node]
set node2_ [$ns_ node]
...
```

In this example, a standard `ErrorModel` instance is created for both the incoming and outgoing wireless channel of each node, with a packet error rate of one percent. Corrupted packets will be sent to a *null agent* that discards them, instead of to the MAC layer or network interface (depending on the packet direction).

5.15 Mobile networking

Mobile networking in ns-2 is based on the mobility extensions [44] developed by the CMU Monarch group [45]. These extensions introduce the notion of *mobile nodes* connected to *wireless channels*, and allow for simulation of wireless networks and ad-hoc networks.

5.15.1 Mobile nodes

A mobile node is a node with extra functionality to adapt it to mobile networking. Figure 5.7 shows the schematics of a mobile node, with an additional agent attached (for packet generation and processing).

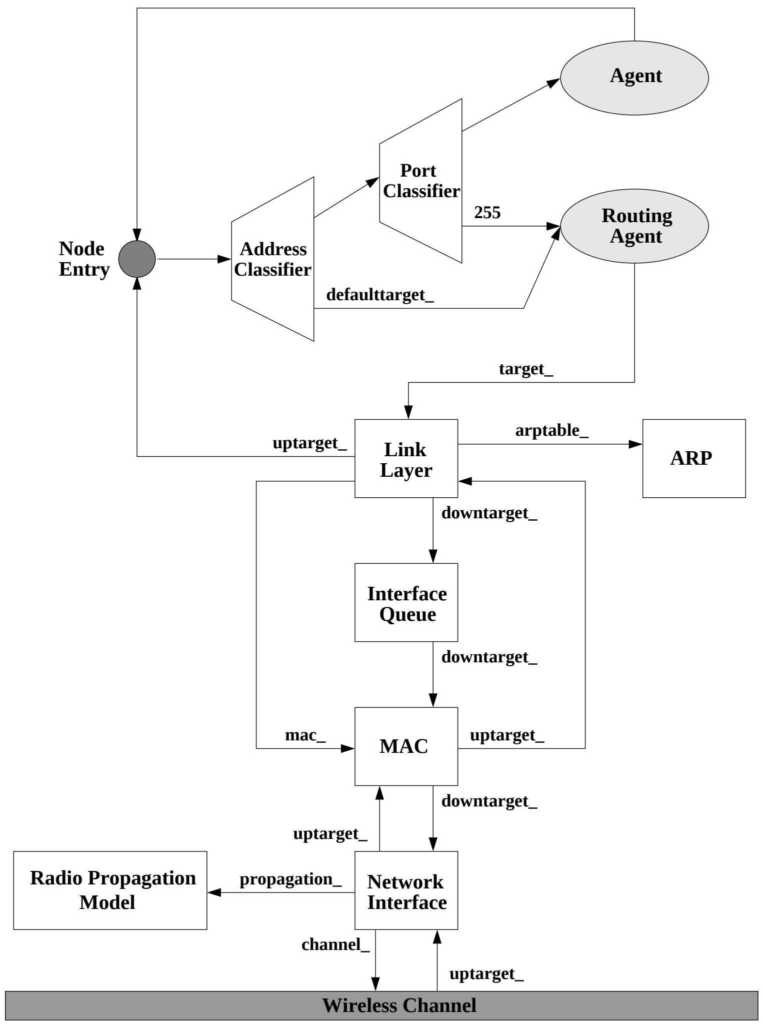


Figure 5.7: Schematics of a mobile node in ns-2

The most important difference between wired nodes and mobile nodes is that mobile nodes are connected to *wireless channels* for their communication, whereas wired nodes are connected by links. Also, mobile nodes may be *moved* within a topography, as opposed to wired nodes which remain stationary. The mobile node itself is a compound object, built from the following parts:

- An *address classifier* used for handing packets to the port classifier or routing agent. The default target of the address classifier is often the routing agent, to allow for packet forwarding.
- A *port classifier*, used for handing packets to agents attached to the mobile node.

- A *routing agent* for routing table management and packet forwarding. The routing agent should set the `next_hop_` field of packets to indicate their next-hop destination.
- A *link layer* responsible for converting network addresses to hardware addresses (with the help of an ARP module, see below) and preparing packets to be put onto a wireless channel.
- An *ARP module* that resolves network addresses to hardware (MAC) addresses.
- An *interface queue*, used for storing packets that should be sent out.
- A *MAC layer* for managing access to the wireless channel.
- A *network interface* that sends and receives packets over the wireless channel.
- A *radio propagation model* determining the signal strength of received packets, and hence, whether a packet can be received by a network interface or not.
- A *wireless channel* over which packets are distributed.

5.15.2 Packet transmission

Packets are transmitted by a mobile node in the following way. The packet is first generated by an *agent* or a *traffic source* at the mobile node. Each generated packet is delivered to the *entry* of the mobile node, i.e., the *address classifier*. This classifier determines if the packet was destined for the current node, or if it should be forwarded. Packets that should be forwarded are handed to the *routing agent* of the mobile node through a *default target* of the address classifier. The routing agent processes the packet, fills in the `next_hop` field of the packet, and passes it down to the *link layer*.

The link layer translates the destination address into a hardware (MAC) address, by letting an *ARP* module generate ARP requests, process ARP replies and fill in the MAC header of the packet. The packet is then handed down to the *interface queue*, which holds packets that should be transmitted. The length of the interface queue can be specified, and depending on the queue type used, routing protocol control packets can be prioritized.

The *MAC layer* retrieves packets from the interface queue when appropriate, i.e., when the wireless channel is free to use. Packets are handed to the *network interface*, which puts the packet onto the wireless channel. Copies of the packet are then delivered to all network interfaces attached to the channel, at the time when a packet would start arriving in a physical system (i.e., based on the speed of light and the distance between nodes). However, this does not necessarily mean that the packet can be correctly received by all of them; *radio propagation models* determine this at the time of packet reception.

5.15.3 Packet reception

When a packet is received by a network interface attached to a wireless channel, the network interface consults a *radio propagation model* for determining whether the packet can be correctly received or not. The settings of the network interface and the selected radio propagation model affect this (see sections 5.16 and 5.17).

If a packet is correctly received, it is handed to the MAC layer by the network interface. The MAC layer hands the packet to the link layer, which in turn hands the packet to the entry of the mobile node, i.e., the address classifier. The address classifier checks the destination address of the packet to see if it matches the address of the current node. If it does, the packet is handed to the port classifier. Otherwise, the packet is handed to a default target – usually the routing agent – for further processing and possible forwarding. The port classifier hands the packet to an agent attached to the mobile node, based on the port number contained in the packet. This completes the packet delivery.

5.15.4 Simulation scenario setup

The setup of a simulation scenario with mobile nodes differs somewhat from a simulation scenario with wired nodes. The following steps should be performed, in addition to those of section 5.2.2, to create a simulation with mobile nodes:

- Wireless tracing should be enabled for NAM (Network Animator) to track movements of mobile nodes. This is described in section 5.18.
- A *topography* must be defined, to confine the area in which mobile nodes may move. A flat grid topography with dimensions `x` times `y` metres is created with the following code:

  ```
  set topo [new Topography]
  $topo load_flatgrid $x $y
  ```

  The topography instance should be passed to the mobile node with the `-topoInstance` option of the `node-config` command (see section 5.7.2).

  Optionally, a third argument may be passed to the `load_flatgrid` command; the resolution of the grid. The default resolution is 1.
- A *GOD (General Operations Director) object* must be created. It annotates trace logs with information about the optimal number of hops from a source to a destination, used for statistics and routing protocol evaluations. A GOD object is created with the following code (`nn` is the total number of nodes in the simulation, and must be specified):

  ```
  set god_ [create-god $nn]
  ```
- If desired, the `Queue/DropTail/PriQueue` queue (commonly used as an interface queue) can be set to prioritize routing protocol control packets. This is done by changing the value of its `Prefer_ Routing_Protocols` variable:

  ```
  Queue/DropTail/PriQueue set Prefer_Routing_Protocols 1
  ```
- The physical characteristics of the network interface should be set, along with configuration of the MAC layer to suit the simulation. This is decribed in section 5.17.

5.15.5 Mobile node confi guration and creation

Mobile nodes are configured and created as usual by issuing the commands `node-config` and `node` to the simulator instance. However, after creating a mobile node, one has to disable the *random motion* feature of the node to prevent it from performing random movements on its own:

```
set mobilenode_ [$ns_ node]
$mobilenode_ random-motion 0
```

5.15.6 Mobile node movements

Mobile nodes may be moved within the topography of the simulation, and keep track of their current positions. Currently, movements are only possible in two dimensions (X and Y), although node positions and movements include a third (Z) value, which is ignored. This limitation implies that flat grid topologies currently are the only meaningful topologies in ns-2 simulations.

Initial positioning

The initial position of a mobile node is specified by setting its X_, Y_ and Z_ instance variables:

```
$mobilenode_ set X_ 10.0
$mobilenode_ set Y_ 20.0
$mobilenode_ set Z_ 0.0
```

For NAM (Network Animator) to be able to correctly place and draw nodes in a visualization of a simulation, all nodes must be passed to the simulator instance through an `initial_node_pos <node> <size>` command, where `node` is a mobile node and `size` is the desired node size in NAM:

```
$ns_ initial_node_pos $mobilenode1_ 10
$ns_ initial_node_pos $mobilenode2_ 10
...
```

Movement commands

Mobile node movements are performed through the `setdest` command, which takes the desired `x` and `y` coordinates and a `velocity` (measured in m/s) as arguments. If one wishes to move the mobile node to these coordinates in a certain time, the velocity for doing so has to be calculated manually.

```
$mobilenode_ setdest <x> <y> <velocity>
```

Random motion

Random motion can be used by issuing the `random-motion 1` command to a mobile node. This command will cause the mobile node to perform motions according to a *random-waypoint model* [2] as soon as the node is given a `start` command. In such a model, the node performs movements in random directions with pauses in between.

Movement pattern files

It is also possible to move mobile nodes according to a *movement pattern file*. Such a file contains OTcl code with `setdest` movement commands scheduled at appropriate times, and may be generated either manually or with the `setdest` utility of the ns-2 distribution.

A movement pattern file can easily be included in a simulation scenario script by asking the OTcl interpreter to *source* it, using the `source "movement.tcl"` command. This includes the specified file as if its contents had been inserted at that line, and allows for easy switching between different movement patterns while keeping the main parts of a simulation scenario script unchanged.

5.15.7 Routing in mobile networking

Routing in mobile nodes is very different from routing in wired nodes. In a wired node, a routing module maintains a routing table, which is used by route logic to perform route computations (see section 5.7.4). These computations are then used by the classifiers of a node to find a wired link to forward packets on.

In mobile nodes, the *routing agent* takes on all the responsibility for routing and forwarding. No routing table is kept in the mobile node itself; instead, the routing agent has to maintain such a routing table internally. No separate route logic is available to perform route computations; this has to be performed by the routing agent. Also, computed routes are not installed by modifying an address classifier. Instead, the address classifier usually has the routing agent set as its default target, and the routing agent itself has to perform the packet forwarding. Finally, this forwarding does not occur over wired links, but over a wireless channel. It is the responsibility of the routing agent to fill in the `next_hop_` field of each packet that should be forwarded, before handing it down to the link layer of the mobile node.

Creating a routing agent is not very different from creating any other agent; it is simply attached to a special port number (RT_PORT, i.e., 255) in the port classifier of a mobile node. However, one has to make sure that the requirements of the routing agent (and the protocol that it implements) are met. Although most routing agents are placed as the default target of the address classifier in a node, some routing algorithms may require a different placement.

Currently, ns-2 includes routing agent implementations of four common routing protocols for wireless simulations; DSDV, AODV, DSR and TORA. However, the continuous updating of routing protocol specifications (and lack of up-to-date implementations) have made some of these routing agents rather obsolete. For instance, the AODV routing agent supposedly adheres to version 6 of the AODV draft, which is several years old (and definitely outdated).

5.15.8 Miscellaneous features

In addition to wireless ad-hoc networking, the mobile networking portions of ns-2 also offer support for *wired-cum-wireless scenarios*, where mobile nodes communicate with *base stations* which act as gateways to a wired network. This is the common infrastructure seen in most wireless LANs today. Furthermore, ns-2 includes support for *Mobile IP* [46], where mobile hosts communicate with their home network through *home agents* and *foreign agents*. Both these additions were not part of the CMU Monarch mobility extensions to ns-2, but were added later. Further documentation on wired-cum-wireless scenarios and Mobile IP can be found in [40].

5.16 Radio propagation models

Radio propagation models are used for calculating the received signal power of packets in wireless simulations. Based on the received signal power and *thresholds*, different actions may be taken for a packet. Either the signal is considered too weak, or the packet is marked as containing errors and thrown away by the MAC layer, or it is received correctly. Currently, there are three radio propagation models implemented in ns-2; the *free space model*, the *two-ray ground reflection model* and the *shadowing model*. Originally, all these models come from the domains of radio engineering and physics.

5.16.1 Free space model

The free space model [47], `Propagation/FreeSpace`, assumes a single, clear line-of-sight path between the transmitting and receiving nodes. It uses the formula in Equation 5.1 to calculate the received signal power at a distance d from the transmitter.

$$P_r(d) = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2 d^2 L} \tag{5.1}$$

In this formula, P_t is the transmitted signal power, G_t is the antenna gain of the transmitter, G_r is the antenna gain of the receiver, L $(L \geq 1)$ is the system loss and λ is the wavelength. In ns-2 simulations, it is very common to select $G_t = G_r = 1$ and $L = 1$.

Usage

Mobile nodes can be configured to use the free space model by issuing the `node-config` command of the simulator instance with the `-propType` option set as follows (prior to node creation):

```
$ns_ node-config -propType Propagation/FreeSpace
```

5.16.2 Two-ray ground reflection model

The two-ray ground reflection model, `Propagation/TwoRayGround`, considers not only a single line-of-sight path between nodes but also *ground reflection*. This model has shown [48] to give more accurate predictions of the received power at long distances than the free space model. It uses the formula in Equation 5.2 to calculate the received signal power at a distance d from the transmitter.

$$P_r(d) = \frac{P_t G_t G_r h_t^2 h_r^2}{d^4 L} \tag{5.2}$$

In this formula, P_t is the transmitted signal power, G_t is the antenna gain of the transmitter, G_r is the antenna gain of the receiver, L $(L \geq 1)$ is the system loss, h_t is the height of the transmitting antenna and h_r is the height of the receiving antenna. However, at short distances d, the two-ray ground reflection model does not give particularly good results due to oscillations caused by the combinations of the two rays. Therefore, the two-ray ground reflection model in ns-2 calculates a *crossover distance*, d_c, for automatically switching between the free space model and the two-ray ground reflection model. The formula for the crossover distance is shown in Equation 5.3 (λ is the wavelength).

$$d_c = (4\pi h_t h_r)/\lambda \tag{5.3}$$

When $d < d_c$, the free space model (Equation 5.1) is used. When $d \geq d_c$, the two-ray ground reflection model (Equation 5.2) is used instead, since both models give the same results for $d = d_c$.

Usage

Mobile nodes can be configured to use the two-ray ground reflection model by issuing the `node-config` command of the simulator instance with the `-propType` option set as follows (prior to node creation):

```
$ns_ node-config -propType Propagation/TwoRayGround
```

5.16.3 Shadowing model

The shadowing model [48], `Propagation/Shadowing`, attempts to more realistically model multi-path propagation effects, i.e., *fading*. This model has two parts; a *path loss model*, which predicts the mean received signal power at the distance d from the transmitter, and a *log-normal random variable*, which models probabilistic communication between nodes at the edge of the radio range. The formula for the shadowing model is shown in Equation 5.4.

$$\left[\frac{P_r(d)}{P_r(d_0)}\right]_{dB} = -10\beta \log\left(\frac{d}{d_0}\right) + X_{dB} \tag{5.4}$$

In this *log-normal shadowing model* formula, $P_r(d_0)$ is the received signal power at the reference distance d_0, $P_r(d)$ is the received signal power at the distance d and X_{dB} is a Gaussian random variable with zero mean and standard deviation σ_{dB}. β is called the *path loss exponent* and σ_{dB} the *shadowing deviation*. These two values need to be set to determine the characteristics of the shadowing model. Some typical values are shown in Tables 5.4 and 5.5.

Table 5.4: Typical path loss exponent (β) values

Environment	β
Outdoor, free space	2
Outdoor, shadowed urban area	2.7 - 5
Indoor, line-of-sight	1.6 - 1.8
Indoor, obstructed area	4 - 6

Table 5.5: Typical shadowing deviation (σ_{dB}) values

Environment	σ_{dB} (dB)
Outdoor	4 - 12
Office, hard partition	7
Office, soft partition	9.6
Factory building, line-of-sight	3 - 6
Factory building, obstructed	6.8

Usage

Mobile nodes can be configured to use the shadowing model by setting its parameters and then issuing the `node-config` command of the simulator instance as follows (prior to node creation):

```
Propagation/Shadowing set pathLossExp_ 1.8  ;# path loss exponent
Propagation/Shadowing set std_db_ 4.0       ;# shadowing deviation (dB)
Propagation/Shadowing set dist0_ 1.0        ;# reference distance (m)
Propagation/Shadowing set seed_ 0           ;# RNG seed

$ns_ node-config -propType Propagation/Shadowing
```

This uses the specified seed value for the random number generator. If one wishes to use another seeding method, this has to be specified, and the propagation instance passed to the `node-config` command:

```
set prop [new Propagation/Shadowing]
$prop set pathLossExp_ 1.8
$prop set std_db_ 4.0
$prop set dist0_ 1.0
$prop seed <seed-type> 0             ;# seeding method (and value)

$ns_ node-config -propInstance $prop  ;# pass propagation instance
```

The (`seed-type`) seeding method parameter should be one of `predef`, `raw` and `heuristic`. A *pre-defined seed* provides a set of known good seeds, while a *raw seed* uses the extra parameter for specifying the seed value. These two seeding methods yield a deterministic behavior. Finally, a *heuristic seed* uses the current time and a counter to generate a seed, and hence, yields a non-deterministic behavior.

5.17 Communication range adjustment

Adjustment of the communication range in wireless networking is done in two steps. First, the characteristics of the network interface have to be set. This is typically done using technical specifications of a wireless network interface as a reference. Second, a *receive threshold* has to be calculated, based on the characteristics of the network interface and the selected radio propagation model.

5.17.1 Network interface characteristics

The parameters of the `Phy/WirelessPhy` network interface that need to be set are the following:

- `Pt_`, the *transmission power* in W (Watts).
- `freq_`, the *frequency* used for wireless communication.
- `CPThresh_`, the *capture threshold*. This value determines how much stronger a radio signal must be than one currently being received for capture to occur. It is measured in dB.
- `CSThresh_`, the *carrier sense threshold*. This value determines the minimal received power in W needed for the network interface to detect a transmission from another node.
- `L_`, the *system loss*. Normally set to 1.0.
- `RXThresh_`, the *receive threshold*. This value determines the minimal received power in W to be able to receive a packet. Calculation of this value is described below.
- `bandwidth_`, the *bandwidth* of the network interface. This does not affect the communication range, but should be set appropriately anyway.

Since signal powers are commonly measured in dBm (dB compared to 1 mW), the conversion formulas of Equations 5.5 and 5.6, from [49], could be useful during calculation of these values.

$$mW = 10^{(dBm/10)} \tag{5.5}$$

$$dBm = 10\log_{10}(mW/1) = 10\log_{10}(mW) \tag{5.6}$$

5.17.2 Receive threshold calculation

The *receive threshold* of the network interface is calculated by supplying its parameters, the radio propagation model and possibly parameters of the radio propagation model to a `threshold` utility that is part of the ns-2 distribution. This utility has the following syntax:

```
threshold -m <propagation-model> [options] <distance>
```

The `propagation-model` should be `FreeSpace`, `TwoRayGround` or `Shadowing`. The communication range, `distance`, is measured in metres. The additional `options` depend on which radio propagation model that is used. The available options are listed below.

- `-Pt <transmit-power>` sets the *transmission power*, corresponding to `Pt_` of the network interface.
- `-fr <frequency>` sets the *frequency*, corresponding to `freq_` of the network interface.
- `-Gt <transmit-antenna-gain>` sets the gain of the transmitting antenna.
- `-Gr <receive-antenna-gain>` sets the gain of the receiving antenna.
- `-L <system-loss>` sets the *system loss*, corresponding to `L_` of the network interface.
- `-ht <transmit-height>` sets the height of the transmitting antenna.
- `-hr <receive-height>` sets the height of the receiving antenna.
- `-pl <path-loss-exponent>` sets the *path loss exponent* β (for the shadowing model).

- `-std <shadowing-deviation>` sets the *shadowing deviation* σ_{dB} (for the shadowing model).
- `-d0 <reference-dist>` sets the *reference distance* d_0 (for the shadowing model).

It is important to check the output of the `threshold` utility to verify that all values are correct, since it assigns values for an ancient Lucent WaveLAN wireless network interface as default values. Finally, the `RXThresh_` value given by the `threshold` utility should be set in the simulation scenario script along with the other settings of the network interface. A complete example is given below, where also the data rates of the MAC layer are configured. The radio propagation model used for calculating the receive threshold in this example is the two-ray ground reflection model, and the values shown yield a communication range of 22.5 metres.

```
Phy/WirelessPhy set Pt_ 0.031622777    ;# Tx power (W), 15 dBm

Phy/WirelessPhy set bandwidth_ 11Mb    ;# 11 Mbps bandwidth
Mac/802_11 set dataRate_ 11Mb          ;# 11 Mbps for data
Mac/802_11 set basicRate_ 1Mb          ;# 1 Mbps for broadcasts

Phy/WirelessPhy set freq_ 2.472e9      ;# Europe, Channel 13, 2.472 GHz
Phy/WirelessPhy set CPThresh_ 10.0     ;# Capture threshold (dB)

Phy/WirelessPhy set CSThresh_ \
                    5.011872e-12       ;# Carrier sense threshold (W),
                                       ;# receiver sensitivity -83 dBm

Phy/WirelessPhy set L_ 1.0             ;# System loss

Phy/WirelessPhy set RXThresh_ \
                    5.82587e-09        ;# Receive threshold (W), from
                                       ;# threshold utility (22.5 m)
                                       ;# given the other parameters
```

5.18 Trace files

Trace files or *trace logs* are an important part of a simulation, since they provide information on the events that occured. Currently, ns-2 offers three different trace formats; an *old trace format* (very commonly used), a *new trace format* and a *tagged trace format*. Also, a special *NAM trace format* is used by NAM for its visualization of simulations.

5.18.1 Trace confi guration

Tracing is configured by issuing commands to the simulator instance. The following commands are available:

- `use-newtrace` selects the new trace format.
- `use-taggedtrace` selects the tagged trace format.
- `trace-all $fd` specifies that tracing should be performed to a trace file referenced by the `fd` file descriptor.
- `namtrace-all $namfd` specifies that tracing should be performed to a NAM trace file referenced by the `namfd` file descriptor.

- `namtrace-all-wireless $namfd <x> <y>` is similar to `namtrace-all`, but annotates the NAM trace with information about the size of the topography (its `x` and `y` dimensions).
- `flush-trace` flushes all open traces to disk. This command should be used at the end of a simulation, before exiting the simulator.

Also, the configuration of nodes heavily affects the amount of logged information. The `node-config` command of the simulator instance allows agent, router, MAC layer and movement tracing to be turned on or off individually (see section 5.7.2). An example usage of these commands is shown below for a wireless simulation. Both a normal trace and a NAM trace is opened, all tracing is enabled, and at the end of the simulation, the traces are flushed to disk before the simulator is halted.

```
set val(x)    50             ;# X dimension of the topography
set val(y)    15             ;# Y dimension of the topography
set val(stop) 200.0          ;# simulation time

set tracefd  [open $val(tr) w]
set namtrace [open $val(nam) w]

$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $val(x) $val(y)

$ns_ node-config -agentTrace ON \
                 -routerTrace ON \
                 -macTrace ON \
                 -movementTrace ON

...

proc finish {} {
    global ns_ tracefd namtrace
    $ns_ flush-trace
    close $tracefd
    close $namtrace
    $ns_ halt
}

$ns_ at $val(stop) "finish"
```

5.18.2 Trace formats

Old trace format

An example of the old trace format is shown below. This format has the advantage that it is relatively easy to read, because of its formatting. Fields are grouped together to provide information from the different parts of a packet, such as the MAC header and the IP header. A description of the contents is given in Table 5.6.

```
s 100.004910095 _1_ RTR  --- 24 AODV 40 [0 0 0 0] -------
 [1:255 0:255 255 0] [0x2 0 [1 0] 20000.000000] (REPLY)
```

Table 5.6: Trace log contents (old trace format)

Column	Contents
1	Event type. r = received, s = sent, f = forwarded, D = dropped.
2	Time of the event.
3	Node ID.
4	Type of tracing object. RTR = router, MAC = MAC layer, IFQ = interface queue, AGT = agent.

5	Reason. END = end of simulation, COL = collision,
	DUP = duplicate, ERR = error, RET = retry count exceeded
	STA = invalid state, BSY = busy, DST = invalid destination,
	NRTE = no route, LOOP = routing loop, TTL = TTL reached zero,
	IFQ = queue full, TOUT = timeout (packet expired),
	CBK = MAC callback, SAL = salvage, ARP = dropped by ARP,
	FIL = filtered, OUT = dropped by base station.

6	UID (Unique ID) of the packet.
7	Packet type.
8	Packet size.
9-1 9-2 9-3 9-4	Expected time to send data. Destination MAC address. Source MAC address. Type (800 = IP, 806 = ARP).
10-1 10-2 10-3 10-4 10-5 10-6	IP source address. Source port number. IP destination address. Destination port number. TTL value. Next hop address.
11 and up	Packet-type specific trace information.

New trace format

An example of the new trace format is shown below. This trace format is not as easy to read as the old trace format, because of the (often) lengthy lines consisting of tag - value pairs. It is however well suited for parsing by trace log analysis utilities. A description of the most common contents is given in Table 5.7, and examples of application tags are given in Table 5.8. New tags will be introduced as new applications are added to ns-2 (if those applications choose to support the new trace format).

```
s -t 100.004910095 -Hs 1 -Hd 0 -Ni 1 -Nx 25.05 -Ny 20.05 -Nz 0.00
-Ne -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.255 -Id 0.255
-It AODV -Il 40 -If 0 -Ii 24 -Iv 255 -P aodv -Pt 0x2 -Ph 0 -Pd 1
-Pds 0 -Pl 20000.000000 -Pc REPLY
```

Tagged trace format

This trace format was recently added to ns-2, and currently lacks documentation. It is similar to the new trace format, using tags and values, but the tag names must be determined by each object to be traced, making the coordination of tag names for this trace format more difficult. As a result, tag clashes are likely to occur. Anyone wishing to use this trace format should therefore investigate the existing tagged trace formats specified by the ns-2 tracing source code carefully.

Table 5.7: Trace log contents (new trace format)

Column	**Contents**
1	Event type. r = received, s = sent, f = forwarded, D = dropped.
Tag	**Description**
-t -t *	Time. Global setting.
-Ni -Nx -Ny -Nz -Ne -Nl	Node ID. X coordinate of node. Y coordinate of node. Z coordinate of node. Node energy level. Trace level (such as AGT, RTR or MAC).

-Nw	Reason. END = end of simulation, COL = collision,
	DUP = duplicate, ERR = error, RET = retry count exceeded
	STA = invalid state, BSY = busy, DST = invalid destination,
	NRTE = no route, LOOP = routing loop, TTL = TTL reached zero,
	IFQ = queue full, TOUT = timeout (packet expired),
	CBK = MAC callback, SAL = salvage, ARP = dropped by ARP,
	FIL = filtered, OUT = dropped by base station.

-Is -Id -It -Il -If -Ii -Iv	IP: Source address.port number IP: Destination address.port number IP: Packet type. IP: Packet size. IP: Flow ID. IP: UID (Unique ID) of the packet. IP: TTL value.
-Hs -Hd	Node ID of this node. Node ID of next hop.
-Ma -Md -Ms -Mt	MAC: Duration. MAC: Destination MAC address. MAC: Source MAC address. Type (800 = IP, 806 = ARP).

Table 5.8: Example application-level trace log contents (new trace format)

Tag	**Description**
-P arp -Po -P arp -Pm -P arp -Ps -P arp -Pa -P arp -Pd	ARP: ARP request/reply. ARP: Source MAC address. ARP: Source address. ARP: Destination MAC address. ARP: Destination address.
-P cbr -Pi -P cbr -Pf -P cbr -Po	CBR: Sequence number. CBR: Packet forwarding count. CBR: Optimal number of forwards.
-P tcp -Ps -P tcp -Pa -P tcp -Pf -P tcp -Po	TCP: Sequence number. TCP: Acknowledgement number. TCP: Packet forwarding count. TCP: Optimal number of forwards.

NAM trace format

The NAM trace format is used by NAM (Network Animator) to visualize simulations. It is a tagged format, resembling the new trace format, and includes support for initialization, node, link, queue, agent and variable events. However, it is far too extensive to be described here. Full details on the NAM trace format can instead be found in [40].

5.19 Problems encountered with ns-2

Following the in-depth review of ns-2 and its network components in the previous sections, some comments should be made regarding problems that one may encounter while using the ns-2 simulation environment. These comments are based on observations during the course of this master's thesis project, as well as several independent reports from individuals on the *ns-users* mailing list.

Perhaps the most frequent criticism against the physical layer and MAC layer in wireless simulations is the difficulty of setting the wireless bandwidth correctly. Many reports indicate that even though the bandwidth and data rate is set to 11 Mbps, the actual throughput is limited to values far below this figure. Apparently, this issue has not been investigated thoroughly enough for a solution to be available.

Another important topic, also related to wireless simulations, is the behavior of wireless broadcasts versus unicasts. The ns-2 model of the 802.11 MAC layer offers separate transmission rate settings for broadcast and unicast network traffic (the *basic rate* and *data rate*, respectively), although this has no effect on the radio range of the transmission; it merely affects the time required for completing the transmission. This is also mentioned in Chapter 7, where, for this reason, some simulation results differ from real-world results.

Finally, a problem related to post-run analysis and debugging should be noted. It is not uncommon for the ns-2 model of the 802.11 MAC layer to drop packets without specifying any reason for doing so. Needless to say, this can make post-run analysis and interpretation of trace logs very cumbersome. Hopefully this will be rectified in a future release of ns-2.

5.20 Summary

The ns-2 network simulator is the result of many years of hard work from a large number of contributors. Its modularity and flexibility has made it one of the most popular network simulators to use for research, and the support for mobile networking has contributed to its widespread usage for performing wireless and ad-hoc networking simulations. It is continuously updated with approximately one major release each year and minor versions in between, although some of its network components (most notably, routing agents) have become rather outdated due to lack of recent implementations. Since ns-2 still is under development, it should be treated as a valuable tool for conducting network simulations rather than *the absolute truth*. That, however, applies to all network simulators.

Chapter 6

Porting AODV-UU to ns-2

6.1 Introduction

This chapter describes the porting of AODV-UU [6] to the ns-2 network simulator [8]. The problems associated with this conversion are pointed out, a number of conversion approaches are reviewed, and finally, the actual conversion process is described. This conversion process is characterized by the ambition to use *the same source code* for the ported version of AODV-UU as for the conventional Linux version, to the extent this is possible.

6.2 Conversion overview

Since routing in mobile nodes in ns-2 is performed by *routing agents*, the goal of the porting process is to create such a routing agent based on the AODV-UU source code. Each node will instantiate this routing agent, and use it for ad-hoc routing in a wireless networking scenario.

Using the same source code for the ported version as for the conventional version requires a substantial amount of care; changes to the original source code should not affect AODV-UU during normal compilation. This calls for an ability to switch between the two versions, depending on whether the compilation should result in a Linux *routing daemon* or an ns-2 *routing agent*.

6.3 Initial decisions

Considering that AODV-UU was written in C, which is a subset of C++, and that network objects in ns-2 performing any kind of packet processing should be implemented in C++ rather than OTcl (according to the ns-2 manual [40]), the decision was made that the porting process should result in a C++ routing agent for ns-2. It was also decided that the existing AODV routing agent of ns-2 should serve as a template during the initial construction of the AODV-UU routing agent.

6.4 Similar projects

During the initial phase of this project, similar projects were searched for. There exist many routing agents for ns-2, e.g. the ones that come as part of the ns-2 distribution (see section 5.15.7), but none of these are the result of porting a C implementation to C++; they were implemented in C++ from the beginning. The closest match was a C++ TORA implementation [50], with support code for running it in ns-2. However, that implementation was also written in C++ from the beginning.

Because of this, focus was put on locating general material on C to C++ conversions. Several useful books and documents on this topic were found, e.g. [51], [52], [53] and [54]. These were used during the porting process.

6.5 Areas of concern

In this section, different areas of concern associated with the porting process are described. This is the main analysis part of the porting process. The issues described here were found by careful examination of the AODV-UU source code [6], the ns-2 source code [8] and the ns-2 manual [40]. It is assumed that the reader is acquainted with AODV-UU (described in Chapter 4) and ns-2 (described in Chapter 5).

6.5.1 Environmental differences

The execution environments of an AODV-UU routing daemon running in Linux and a routing agent in ns-2 are very different. In Linux, a single instance of the AODV-UU routing daemon is executed on each host, and uses a physical wireless network interface for its communication.

In ns-2, routing agents are instantiated for each node in the simulation. All routing agent instances execute in the same environment, i.e., they all run on the same host, and hence must be able to co-exist without interfering with each other. Furthermore, routing agents do not have access to a "real" physical wireless network interface.

6.5.2 Network interfaces

The AODV-UU routing daemon retrieves information about network interfaces during its initialization in the main module. For each network interface, the IP address, name, netmask and broadcast address is retrieved. No such information is kept for network interfaces (e.g. the `Phy/WirelessPhy` network interface) in ns-2. Instead, all addressing information is kept in each node, and packets carry all the information needed for transmission. Furthermore, support for multiple network interfaces and multiple wireless channels is still somewhat experimental in ns-2, and most routing agents do not support such operation.

6.5.3 Packet handling

Packet reception

The packet handling of the AODV-UU routing daemon can be divided into two parts; handling of AODV routing protocol control packets and handling of data packets. AODV control packets are received by the `aodv_socket_read()` function of the aodv_socket module, whereas data packets are received by the `packet_input()` function of the packet_input module.

In ns-2, routing agents receive all incoming packets through a single `recv()` (receive) method. Packet types and contents have to be analyzed by this method to determine how packets should be processed. There are no sockets available, but agents achieve similar functionality by attaching themselves to the port classifier of a node, specifying the port number of interest. It should be noted that routing agents always are attached to port number 255 (RT_PORT) of a node, and hence, the port number for exchanging routing protocol control packets is 255 rather than any other port number (e.g. 654 in the case of AODV).

Packet types

In ns-2, packet types have to be created manually for specific purposes. Many common packet types are built-in, but they may not be as easy to use as in the real world. For instance, UDP packet processing would require the usage of a special `Agent/UDP` transport agent, which makes it more difficult for other agents to utilize UDP as part of their operation. A real-world application (such as the AODV-UU routing daemon) could use UDP sockets for the same purpose, without experiencing problems related to its position in the network stack.

Packet direction

The kernel module of AODV-UU uses Netfilter hooks for determining whether a packet is an incoming packet or an outgoing packet, and whether it should be routed or passed on to the system for forwarding. In ns-2, the direction of a packet can be determined by analyzing the `direction_` field of its common header. This field is set to DOWN, UP or NONE depending on the direction of the packet. DOWN indicates that the packet is an outgoing packet, and UP that it is an incoming packet.

Packet transmission

The AODV-UU routing daemon sends outgoing packets through sockets, which are then caught by the hook for locally generated packets, NF_IP_LOCAL_OUT, processed by AODV-UU, caught by the post-routing hook, NF_IP_POST_ROUTING, and finally forwarded by the system.

In ns-2, routing agents instead hand all outgoing packets to their default target (the link layer) after filling in the `next_hop_` field of the common header. As a sidenote, this makes routing agents differ somewhat from other agents, whose default target instead is the node entry to allow generated packets to arrive at the routing agent for forwarding.

6.5.4 Kernel interaction

AODV-UU uses kernel interaction for updating the kernel routing table (the k_route module), sending ICMP messages to an application (the icmp module) and receiving packets on Netfilter hooks (the kaodv module). Obviously, no kernel interaction should be part of the execution of an ns-2 routing agent; the only environment offered to the routing agent is the ns-2 environment. The routing table is kept internal to the routing agent, ICMP messages are not passed to applications (because of the very limited application API upcalls provided by ns-2) and reception of packets is entirely performed through the `recv()` method of the routing agent.

6.5.5 Variables

The variables in the AODV-UU source code affect each routing daemon process separately. In ns-2, any variables exposed to the C++ environment will be globally available. Therefore, the instantiation of routing agents in ns-2 requires variables to be local to each routing agent instance.

Global variables

Global variables are used in AODV-UU for sharing global settings, such as configuration options, between all modules. It is a convenient way of sharing simple data, to avoid passing around information data structures everywhere. However, global variables of AODV-UU modules must not be global in a routing agent implementation. If they were, all routing agents would share the same variables, interfering with each other. Instead, these variables must be local to each routing agent instance.

Static variables

Static variables offer hiding of information and functions from other modules, and are important for a modularized software development in C. However, this information hiding may be unnecessary in an ns-2 routing agent, depending on whether a modularized approach is desired *within* the routing agent or not.

AODV-UU uses static variables e.g. for hiding functions that should only be called from within a certain module and for hiding variables associated with a single module, such as the seeking list of the seek_list module and the send and receive buffers of the aodv_socket module.

6.5.6 Timers

The AODV-UU routing daemon uses the timer of `select()` for scheduling the checking of its own timer queue in the timer_queue module. Handler functions are executed as timers expire, and the timeout of the `select()` timer is updated when events have been carried out.

In ns-2, timers of the `TimerHandler` class are available for agents and other network objects to use, thereby allowing similar functionality to be achieved. A timer in ns-2 could replace the `select()` timer and examine the timer queue of the timer_queue module upon expiration. It should be noted, however, that such a timer would have to exist for each routing agent instance, since packet handling of different routing agent instances should be separate.

6.5.7 Logging

The logging portions in the debug module of AODV-UU pose yet another problem. In ns-2, these logging features have to be modified to allow for individual logging of events for each routing agent instance. Such a modification would have to use a sensible logfile naming scheme to avoid ambiguities.

6.5.8 Non-applicable features

AODV-UU has a number of features that are not relevant or directly applicable to simulations, for one reason or another. These features are listed below.

- *Internet gateway support*. This feature implies that the host is connected to the Internet, allowing itself to function as a gateway for other nodes in the ad-hoc network. Such a feature is normally handled by base stations in a wireless LAN, which are connected to a wired network.

  However, since no Internet access is available in ns-2 simulations, this feature does not apply. Also, the experimental support for multiple wireless channels and network interfaces in ns-2 would currently make an implementation of such a feature cumbersome.

- *Specifying network interfaces* to attach to. This does not apply to routing agents in ns-2, since their only means of communication is through a link layer, not through network interfaces. A routing agent does not know the exact layout of the network stack.

- *Daemon mode*. The daemon mode option of AODV-UU refers to detaching the agent from the console during startup by closing the stdin, stdout and stderr streams and executing the daemon in the background. This does not apply to routing agents in ns-2, since it is the ns-2 simulator itself that possibly should be detached from the console. Network components execute as part of the ns-2 environment, not as separate processes.

- *Version information* and *online help*. This is not very useful in ns-2 simulations. The version of the AODV-UU routing agent, if considered important, should be made known to the user prior to installation. Similarly, the user should read the documentation accompanying the AODV-UU routing agent prior to running it, as with any other software.

6.6 Different approaches of conversion

Given the task of porting AODV-UU to run in the ns-2 network simulator, a number of conversion approaches are possible. Each approach determines how the original source code should be handled, and the results of a conversion. The approaches that have been considered are described in the following subsections, along with their respective advantages and disadvantages. The choice of a conversion approach heavily depends on the existing source code and the environment in which it should be executed. Therefore, the considerations described in section 6.5 should be kept in mind when such a choice is made.

Finally, it should be noted that the approaches described here by no means are the only possible ones. Any programmer with good skills in analyzing source code, and perhaps some imagination, could create hybrid approaches to suit the conversion task at hand.

6.6.1 Monolithic class approach

In a *monolithic class approach*, all source code is placed in a single class. The resulting class need not have any relationship with other classes; it could exist on its own as a stand-alone class.

The main advantage of this approach is its simplicity. It is very easy to incorporate existing source code into a single class, since there is no doubt about where to place the code. However, there are disadvantages as well. *Name clashes* of identifiers could cause problems, unless the original source code utilizes a decent scheme for naming functions and variables. Also, any *structure* prior to the conversion is lost, since all parts are collapsed to form the monolithic class. Therefore, this approach may be unsuitable for well-structured source code with a large number of similar identifiers.

6.6.2 Object-oriented approach

In an *object-oriented approach*, the intention is to arrange the original source code to form classes, of which objects may be instantiated.

The suitability of this approach depends on the layout of the original source code and the ambition of the programmer. If the original source code lacks a modular structure, the conversion task could become tedious and very time-consuming. On the other hand, if the original source code is divided in separate modules, each with a clearly assigned task, these modules could form a good basis for the construction of corresponding classes. However, the programmer may still consider the task of converting a non-object-oriented program into an object-oriented one – by preserving the original source code – an inappropriate way of obtaining the desired results. It may be better to apply an object-oriented analysis to the original problem, and write an object-oriented version of the program from scratch.

6.6.3 Daemon approach

In a *daemon approach*, programs that execute as separate processes can be used as-is, provided that an appropriate *interface* is written and connected to the application wishing to use these processes.

The advantage of this approach is that the original source code may remain unchanged. The problem instead lies in the ability to establish communication with the interface software. For instance, AODV-UU routing daemons could be attached to virtual network interfaces, and software between these virtual network interfaces and the ns-2 network simulator could convert data as needed. Hence, the complexity of modifying the original source code has moved to the interface software and its associated communication.

An important aspect (and a disadvantage) of the daemon approach is *resources*. It may not be possible to execute many daemons in parallel because of resource constraints. For instance, a large number of routing daemons running concurrently would not only consume a considerable amount of memory; they would also increase the workload of the machine on which they are executed. Furthermore, the daemon approach may be unsuitable because of constraints of the application utilizing the daemon processes. For instance, the application may not be able to keep up with the amount of data generated by all daemon processes in real-time. Finally, the real-time aspect itself may be a big disadvantage even though the application *is* able to cope with the data generated by the daemon processes, if the fastest possible execution is desired.

6.7 The porting process

In this section, one of the conversion approaches is selected and the task of porting AODV-UU to run in the ns-2 network simulator is described in detail. The resulting source code of this conversion, i.e., the source code of AODV-UU version 0.5, is available from the AODV-UU homepage [6].

6.7.1 Choice of conversion approach

For the porting of AODV-UU to run in the ns-2 network simulator, the *monolithic class approach* was chosen. The reasons for this choice were the following:

- The contents of the source code does not justify creating classes for instantiation of objects. Entries of linked lists, such as routing table entries and seeking list entries, are the only dynamically allocated objects that would be useful to instantiate. Most other objects exist in only one instance.
- The simplicity of the monolithic class approach. During the initial phase of the project, it was uncertain whether a conversion was possible, and to what extent the original source code would have to be modified.
- The layout of routing agents in ns-2. Most routing agents contain only one (or very few) modules to be compiled with ns-2.
- A daemon approach would require usage of the real-time scheduler of ns-2. This scheduler has proven to work poorly, complaining about system time running backwards and crashing the simulator. These problems have been reported by several ns-2 users on the *ns-users* mailing list [57] as well. Furthermore, usage of the real-time scheduler does not utilize the full processing capability of the simulator and the computer. Time-consuming simulation scenarios would take a long time to run. Finally, it was not clear how the interface between AODV-UU routing daemons and the simulator should be realized.
- The goal of the project was not to create an object-oriented version of AODV-UU, but to use the existing (non-object-oriented) source code with as little modifications as possible.

6.7.2 Conceptual overview

The ideas proposed for the porting process were the following. The original source code should form a single, monolithic C++ routing agent class, which should be possible to instantiate in the ns-2 environment (both in C++ and OTcl). Parts of the original source code should be *automatically extracted* and put into the correct places of this class by the *preprocessor*. This involves one or more passes of source code extraction for each source code file, using *preprocessor directives* and *macros*. Such preprocessing constructs should be logically defined such that they do *not* affect the original source code during "normal" compilation, i.e., compilation of an AODV-UU routing daemon. Furthermore, the *separate compilation* of each module of AODV-UU should be preserved. The details of the porting process are described in the following subsections.

6.7.3 The AODVUU class

The AODV-UU routing agent class is called `AODVUU`, and its definition is placed in an aodv-uu.h header file. The main methods of this class, offering basic routing agent functionality, are placed in a corresponding aodv-uu.cc source code file. OTcl linkage is performed in aodv-uu.cc by a static `TclClass` instance.

6.7.4 Methods

All function definitions in the AODV-UU source code are prepended with a special `NS_CLASS` macro, which defines these functions to be methods of the `AODVUU` class. Their declarations are automatically placed inside the class by the preprocessor, as described in section 6.7.5. The following are the main methods of the `AODVUU` class:

- `AODVUU(nsaddr_t id)` is the constructor of the `AODVUU` class, and takes the node ID of the node to which the routing agent is attached as an argument. It sets up an AODV-UU packet type by calling the `Agent` base class constructor with the packet type as a parameter. A faked network interface with the name "nsif" is created, and its parameters are initialized using the address of the agent. The constructor also binds configuration variables to the OTcl environment, initializes the data structures of AODV-UU (the timer queue, routing table and packet queue) and calls initialization methods of other modules.
- `~AODVUU()` is the destructor of the `AODVUU` class. It destroys the contents of the routing table, destroys any packets buffered by the packet_input module, and closes the logfiles.

- `void scheduleNextEvent()` checks the next event to occur in the timer queue of the timer_queue module, and schedules an ns-2 timer of the `AODVUU` class to expire at the time of that event. This method should be called as soon as it is possible that the timer queue has changed, e.g. after processing packets. This construct is the equivalent of the `select()` timeout used in the AODV-UU routing daemon.

- `void recv(Packet *p, Handler *)` is the receive method of the AODV-UU routing agent. This method receives all packets that are either destined for the routing agent or should be forwarded. It differs between AODV routing control packets and data packets, and either calls the `recvAODVUU Packet()` method of the aodv_socket module or the `processPacket()` method of the packet_input module. After reception of a packet, the timer queue is rescheduled, since it may have changed as an effect of the packet processing performed by AODV-UU.

- `void packetFailed(Packet *p)` will receive packets returned from the link layer, when the link layer finds out that a packet cannot reach its next-hop destination due to communication range limitations or other events hindering the transmission. The usage of this method depends on whether link layer feedback has been enabled (by defining AODVUU_LL_FEEDBACK during compilation) or not. The purpose of this method is to mark failed routes as down, and to drop the failed packet with the DROP_ RTR_MAC_CALLBACK reason to indicate that the delivery failed due to errors at the link layer.

- `void sendPacket(Packet *p, u_int32_t next_hop, double delay)` schedules the sending of a packet to the link layer using the supplied next-hop information and the desired delay. If link layer feedback has been enabled, information is added to the packet so that it will be possible to call a link layer callback method of the `AODVUU` class, `link_layer_feedback()`, in case a transmission of the packet would fail at the link layer.

- `int startAODVUUAgent()` starts the AODV-UU routing agent. It checks that the routing agent has been initialized properly, sets up the wait-on-reboot timer, schedules the sending of HELLO messages (if link layer feedback is not used), initializes logging, and reschedules the timer queue. This method either returns TCL_OK if the routing agent was started properly, or TCL_ERROR if it failed.

- `void link_layer_feedback(Packet *p, void *arg)` is called when the transmission of a packet fails at the link layer. It calls the `packetFailed()` method of the supplied AODV-UU routing agent instance with the failed packet as a parameter, allowing the failure to be handled.

- `gettimeofday()` overrides the usual `gettimeofday()` functionality offered by <sys/time.h>. It is used by the timer_queue module to schedule timeouts and by the debug module for logging events. The value returned by this method corresponds to the current notion of time of the ns-2 scheduler, with each simulation starting at 00:00:00 (GMT) on January 1, 1970. This allows simulations to be performed without the ns-2 real-time scheduler, since the notion of time presented to the AODV-UU routing agent follows that of the simulator.

- `if_indextoname()` overrides the ususal `if_indextoname()` function offered by <net/if.h>. It is used by logging functions of the debug module for finding the name of a network interface, given its interface index.

6.7.5 Source code extraction

The extraction of source code from the original source code files of AODV-UU for inclusion in the `AODVUU` class is perhaps the most important part of the porting process. It allows the original source code files to remain (almost) unchanged, and the construction of the `AODVUU` class to be easily performed. It should be noted that the objective of this source code extraction is *not* to put all source code into a single file. Rather, source code extraction is used for creating the *declaration* of the `AODVUU` class. The actual methods of this class remain in their original modules, each yielding an object (.o) file during compilation. These object files are then combined during linking to form the complete AODV-UU routing agent.

Macros

The macros used in the ported version of AODV-UU are listed below. `#ifdef MACRO` and `#ifndef MACRO` constructs together with these macros allow code segments to be *selectively* included or excluded by the preprocessor. The reason for "negating" some of the macros is that if none of them are defined, AODV-UU should compile as usual, resulting in an AODV-UU routing daemon.

- `NS_PORT` should be defined during the compilation of an AODV-UU routing agent, i.e., during the "porting" of AODV-UU to ns-2. It is used in .c files for excluding variable initializations (which must instead be performed in the `AODVUU` constructor) and for excluding `#include` directives. In the ported version, all .c files will rely on a single header file, aodv-uu.h, for all method and variable declarations.
- `NS_NO_GLOBALS` is used for indicating that "global" includes and variables, such as standard include files and global variables, should *not* be part of the compilation. This macro is used in .h files for allowing the `AODVUU` class to exclude globals during source code extraction.
- `NS_NO_DECLARATIONS` is used for indicating that method declarations should *not* be part of the compilation. This macro is used in .h files for allowing the `AODVUU` class to exclude method declarations during source code extraction.
- `NS_CLASS` is used for prepending function definitions in the AODV-UU source code with the name of the `AODVUU` class. It expands to `AODVUU::`, and makes the affected functions members of the `AODVUU` class.
- `NS_OUTSIDE_CLASS` is used for calling global functions from within the `AODVUU` class, whose names collide with methods inside the class. For instance, `close()` is such a function. If NS_PORT is defined, this macro expands to the `::` scope operator of C++.
- `NS_STATIC` is used for removing the `static` keyword of function definitions. This is done because such functions, hidden from other modules through the `static` keyword, should *not* be shared by all instances of the `AODVUU` class in the ported version. (They will typically refer non-static data.)
- `AODV_UU` is used in ns-2 source code files for selectively including code segments that only apply to AODV-UU.

Also, some macros are used for other definitions associated with the ported version of AODV-UU:

- `NS_DEV_NR` is the index of the (virtual) network device in the `devs` array of the current host, i.e., the AODV-UU routing agent. It is set to zero, since only one network device is supported.
- `NS_IFINDEX` is the interface index of the (virtual) network interface. It is set to the same value as NS_DEV_NR, since only one network interface is supported.
- `AODV_LOG_PATH_PREFIX` is used for defining the filename prefix of logfiles generated by AODV-UU routing agents. The default is "aodv-uu_".
- `AODV_LOG_PATH_SUFFIX` is used for defining the filename suffix of logfiles generated by AODV-UU routing agents. The default is ".log".
- `AODV_RT_LOG_PATH_PREFIX` is similar to AODV_LOG_PATH_PREFIX, but for routing table logfiles. The default is "aodv-uu_rt".
- `AODV_RT_LOG_PATH_SUFFIX` is similar to AODV_LOG_PATH_SUFFIX, but for routing table logfiles. The default is ".log".

Construction of the AODVUU class

The aodv-uu.h header file of the `AODVUU` class is organized in the following way (items are listed in order):

- A check is made to ensure that NS_PORT has been defined. This indicates that AODV-UU is to be compiled as an ns-2 routing agent.
- Header files needed from ns-2 are included. These allow packet types, timer classes, random generators and trace file support to be used.
- A forward declaration of the `AODVUU` class is made. This is needed to be able to reference the class before it has been completely defined. The modules of AODV-UU need to know about the class during the source code extraction, e.g. for declaring member function pointer variables for timers.
- Global definitions of AODV-UU are included, i.e., params.h and defs.h. These are needed by all parts of AODV-UU, and are therefore included on a global level.
- Global data types, defines and global declarations are selected and extracted from all header (.h) files of AODV-UU. This is done by defining NS_NO_DECLARATIONS, so that no method declarations will be extracted, and by undefining NS_NO_GLOBALS, to ensure that the desired parts of the header files are extracted. `#include` directives perform the actual source code extraction.
- A timer class, `TimerQueueTimer`, is defined. This class is used for instantiating an ns-2 timer, `tqtimer`, which serves as a replacement for the `select()` timer used by the AODV-UU routing daemon. The intention of this timer is to keep track of upcoming events in the timer queue of the timer_ queue module.
- The `AODVUU` class is declared. Its public and protected methods are declared. Method declarations from the header (.h) files of AODV-UU are selected by defining NS_NO_GLOBALS and undefining NS_NO_ DECLARATIONS, and extracted with `#include` directives. For this extraction to work, the macro defined by each header file has to be *undefined* with `#undef AODV_MODULE_H` first, since the header file has already been included once before. This is followed by declarations of member variables that were previously global (taken from the main module), or that were static and placed in some of the other modules. The initialization of these variables is performed in the `AODVUU` constructor.
- The `ifindex2devindex()` method from defs.h is placed below the `AODVUU` class declaration. It is manually placed there because it needs the `AODVUU` class declaration to be able to reference member variables of the class. (Otherwise, an extra pass through defs.h would be needed to extract just one method.)

Notes on source code extraction

Obviously, the chosen approach does not entirely rely on automatic source code extraction for the construction of the `AODVUU` class. Static variables of modules have been manually moved into the class, and their initializations are performed in the constructor. This is a trade-off between automation and the number of passes needed for the preprocessor to extract the source code and put it into the `AODVUU` class.

6.7.6 Packet types and headers

To allow AODV-UU routing agents to communicate with each other, a special `PT_AODVUU` packet type and `PacketHeader/AODVUU` packet header was added to ns-2. This is in contrast with the AODV-UU routing daemon, which uses UDP for its AODV control packets. The reasons for this decision were the following:

- Introduction of a separate packet type allows for *easy tracing* of packets of this type. No existing trace code has to be modified to suit the specific tracing needs of the packet type; instead, custom-made tracing code may be supplied.

- UDP packet processing would complicate the AODV-UU routing agent, since it would require usage or subclassing of the `Agent/UDP` transport agent. Also, the UDP packet type would offer no significant advantages over a custom-made packet type.
- The approach with separate packet types has become a *de facto* standard in ns-2. A majority of all agents implement their own packet types.

The AODV-UU packet type

An AODV-UU packet type, `PT_AODVUU`, was added to be able to reference AODV-UU routing control packets in simulations. It was inserted into the the `packet_t` enumeration in ˜ns/common/packet.h. Also, the name of the packet type was set to "AODVUU" in the constructor of the `p_info` class. To enable the usage of this packet type in OTcl, its name was added to the protocol enumeration in ˜ns/tcl/lib/ns-packet.tcl.

The AODV-UU packet header

The AODV-UU packet header (`PacketHeader/AODVUU`) was based on the existing `AODV_msg` message type of AODV-UU, defined in defs.h. The following additions were made:

- A static inline variable, `offset_`, was added. This variable is required by the `PacketHeaderManager` of ns-2 for storing packet header offsets for each packet header during startup, and can be used for accessing the AODV-UU packet header of packets.
- A static inline method, `offset()`, was added. This method returns the offset mentioned above.
- An *access method*, `AODV_msg *access(const Packet *p)`, was added. This method allows the AODV-UU packet header to be accessed in packets. It is a shorthand for retrieving the packet header offset and accessing the packet using that offset.
- A type definition was made, to allow the AODV_msg struct to be referred to as `hdr_aodvuu`. This is a recommended naming convention of ns-2.
- An access macro, `HDR_AODVUU(p)`, was defined. It is a shorthand for calling the `access()` method of the `hdr_aodvuu` type to retrieve the AODV-UU packet header of a packet `p`.
- The packet header was made available to OTcl by creating a static class, `AODVUUHeaderClass`, and instantiating it as `class_rtProtoAODVUU_hdr` in aodv-uu.cc.

  In the constructor of `AODVVUUHeaderClass`, the `PacketHeaderClass` base class constructor is called both with the desired OTcl class name of the packet header (`PacketHeader/AODVUU`) and the *size* of the packet header. This size is set to be the maximum AODV message size, as defined by AODV_ MSG_MAX_SIZE in aodv_socket.h. This is to reserve space for the largest AODV message allowed (a RERR message with 100 unreachable destinations).

AODV-UU packet header usage

The AODV-UU packet header, once enabled, is part of all ns-2 packets. Since the size of the packet header is fixed, each packet will carry a constant amount of additional data as a consequence of this. This, however, does not mean that the size of AODV-UU packets will appear as a constant value in trace logs. The amount of meaningful information in a packet of the AODV-UU packet type, plus the size of the IP header, is set in the `size_` field of its common header prior to sending the packet.

In the same way, the effective AODV-UU packet size is calculated from the `size_` field of the common header upon packet reception. The size of the IP header is subtracted from the `size_` value, giving the effective AODV-UU packet size. This value can then be used when parsing the information contained in the packet.

It should also be noted that no special conversion of the AODV packet data is performed in the ported version of AODV-UU. The data is stored in network byte order, just like in the conventional version of AODV-UU.

6.7.7 Packet buffering and handling

The internal packet buffering of AODV-UU performed in the packet_queue module had to be modified to buffer the actual *packets* rather than their *packet IDs* provided by Netfilter, since Netfilter is not available in the ns-2 environment. The required changes were applied to the packet_queue module of AODV-UU.

The packet handling of the AODV-UU routing agent is almost identical to that of the AODV-UU routing daemon; the difference is that all packets arrive through the `recv()` method of the routing agent instead of through different sockets. The packet type is analyzed in the `recv()` method, after which the correct handler method (`recvAODVUUPacket()` of the aodv_socket module or `processPacket()` of the packet_input module) is called.

6.7.8 Addressing

The addressing in AODV-UU assumes a 32-bit unsigned integer value, representing the IP address. The ns-2 addressing scheme uses signed 32-bit integers for its node IDs, as described in section 5.7.3. However, this does not pose any specific problems for the AODV-UU routing agent. Nodes are never assigned negative node IDs, so the routing agent can safely type cast its node ID into an IP address. The IP address is stored in the structure for network device information of AODV-UU, and is used e.g. for setting the source address when sending packets.

6.7.9 Timers

The existing timer queue module of AODV-UU, timer_queue, was kept intact in the ported version. To replace the `select()` timer used by the AODV-UU routing daemon, a special timer subclass of the `TimerHandler` class was constructed, `TimerQueueTimer`, and instantiated as `tqtimer` by the AODV-UU routing agent. Furthermore, the `TimerQueueTimer` class was made `friend` of the `AODVUU` class, so that it can access methods of `AODVUU` when the timer expires. (Details on timers are available in section 5.12.)

The `tqtimer` timer is kept synchronized with the timeout of the next upcoming event of the timer queue in the timer_queue module. When it expires, it calls the `timer_age_queue()` method of the timer_queue module. This method checks for any expired timeouts, executes any corresponding events, and returns the new timeout of the timer queue. This value is used for rescheduling the `tqtimer` timer, to once again synchronize it with the timer queue. It should be noted that it is the responsibility of the programmer to make sure that this timer synchronization is performed, by calling the `scheduleNextEvent()` method of the `AODVUU` class whenever it is possible that the timer queue has changed. However, the timer queue can only change during the processing of an AODV message, or as a side-effect of expired timers, so it is rather easy to enforce this policy.

Finally, a comment should be made regarding the calling of timer handler functions. The AODV-UU routing agent required the calls to timer handler functions (and the setting of timer handler functions when creating timers) to be modified. This was because of the instantiation of routing agents; each such handler function is associated with a certain routing agent instance. Therefore, references to timer handler functions were made instance-aware by utilizing the NS_CLASS macro and `this` pointers, pointing to the current routing agent instance. This can be seen in the modified source code of AODV-UU wherever a timer handler method is assigned or called. Details on function pointers in C++ are available in [53].

6.7.10 Logging

The logging portions of the AODV-UU debug module were slightly modified to allow for the instantiation of routing agents. Filename prefixes and suffixes for "general" logfiles and routing table logfiles were added to defs.h. The `log_init()` function of the debug module was modified to construct filenames for logfiles from these prefixes, suffixes and the IP address of the routing agent (i.e., the node). The resulting filenames could e.g. be "aodv-uu_0.0.0.1.log" and "aodv-uu_rt_0.0.0.1.log" for a node with IP address 0.0.0.1 (which is equal to a node ID of 1). This way, each routing agent writes log information to a unique set of logfiles. In addition, the IP address of the agent was added to each line in the "general" logfile to increase clarity.

6.7.11 Tracing

Tracing support for the AODV-UU packet type was added using the tracing source code of the existing AODV routing agent in ns-2 as a template. The tracing source code is available in ˜ns/trace/cmu-trace.{cc, h}. It was decided that the trace format should be equal to that of the existing AODV routing agent, to easily let users switch between the two implementations. The only difference in the tracing of AODV-UU packets is that the packet type field of the ns-2 trace log will read "AODVUU" instead of "AODV". Finally, to perform this tracing, a conversion of the fields in the AODV-UU packet header was needed, since these are stored in network byte order but should be displayed in host byte order.

The AODV-UU routing agent supports the old and the new trace format of ns-2, but not the tagged format. (Details on these formats are available in section 5.18.1.) In the following subsections, examples of the two supported formats are given for each possible AODV message type, and the fields relevant to AODV are explained.

RREP and HELLO messages

```
s 100.004910093 _1_ RTR  --- 24 AODVUU 40 [0 0 0 0] -------
 [1:255 0:255 255 0] [0x2 0 [1 0] 20000.000000] (REPLY)
                      (1)(2)(3)      (5)
                              (4)

s -t 100.004910093 -Hs 1 -Hd 0 -Ni 1 -Nx 25.05 -Ny 20.05 -Nz 0.00
-Ne -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.255 -Id 0.255
-It AODVUU -Il 40 -If 0 -Ii 24 -Iv 255 -P aodvuu -Pt 0x2 -Ph 0 -Pd 1
                                                     (1)    (2)   (3)
-Pds 0 -Pl 20000.000000 -Pc REPLY
    (4)        (5)

s 12.636748053 _0_ RTR  --- 6 AODVUU 40 [0 0 0 0] -------
 [0:255 -1:255 1 0] [0x2 0 [0 5] 2000.000000] (HELLO)
                     (1)(2)(3)     (5)
                             (4)

s -t 12.636748053 -Hs 0 -Hd -2 -Ni 0 -Nx 6.80 -Ny 3.92 -Nz 0.00
-Ne -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.255 -Id -1.255
-It AODVUU -Il 40 -If 0 -Ii 6 -Iv 1 -P aodvuu -Pt 0x2 -Ph 0 -Pd 0
                                                  (1)    (2)   (3)
-Pds 5 -Pl 2000.000000 -Pc HELLO
    (4)       (5)
```

`(1)`: Packet type. 0x2 = RREP.
`(2)`: Hop Count.
`(3)`: Destination IP Address (i.e., the node ID).
`(4)`: Destination Sequence Number.
`(5)`: Lifetime.

It should be noted that since HELLO messages are RREP messages, the packet type `(1)` is 0x2 for HELLO messages as well.

RREQ messages

```
s 127.942238029 _0_ RTR  --- 1 AODVUU 44 [0 0 0 0] -------
 [0:255 -1:255 1 0] [0x1 0 0 [2 0] [0 1]] (REQUEST)
                     (1)(2)  (4)   (6)
                          (3)  (5)   (7)

s -t 127.942238029 -Hs 0 -Hd -2 -Ni 0 -Nx 89.66 -Ny 283.49 -Nz 0.00
-Ne -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.255 -Id -1.255
-It AODVUU -Il 44 -If 0 -Ii 1 -Iv 1 -P aodvuu -Pt 0x1 -Ph 0 -Pb 0 -Pd 2
                                                  (1)    (2)   (3)   (4)
-Pds 0 -Ps 0 -Pss 1 -Pc REQUEST
    (5)   (6)    (7)
```

(1): Packet type. 0x1 = RREQ.
(2): Hop Count.
(3): RREQ ID.
(4): Destination IP Address (i.e., the destination node ID).
(5): Destination Sequence Number.
(6): Originator IP Address (i.e., the originator node ID).
(7): Originator Sequence Number.

RERR messages

```
s 189.612645446 _1_ RTR  --- 6808 AODVUU 32 [0 0 0 0] -------
 [1:255 0:255 1 0] [0x3 1 [2 3] 0.000000] (ERROR)
                    (1)(2)(3)      (5)
                            (4)

s -t 189.612645446 -Hs 1 -Hd 0 -Ni 1 -Nx 221.83 -Ny 80.86 -Nz 0.00
-Ne -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.255 -Id 0.255
-It AODVUU -Il 32 -If 0 -Ii 6808 -Iv 1 -P aodvuu -Pt 0x3 -Ph 1 -Pd 2
                                                     (1)    (2)   (3)
-Pds 3 -Pl 0.000000 -Pc ERROR
    (4)       (5)
```

(1): Packet type. 0x3 = RERR.
(2): DestCount.
(3): Unreachable Destination IP Address 1 (i.e., the node ID).
(4): Unreachable Destination Sequence Number 1.
(5): Lifetime.

It should be noted that a lifetime field is not part of the RERR message. However, this field was logged by the AODV routing agent supplied with ns-2, because it handles several message types equally when logging them. The value of this field was always 0.0 because it was never initialized. For backward compatibility with the AODV trace log format, it is included in AODV-UU trace logs as well (always displaying a value of 0.0).

RREP-ACK messages

Currently, AODV-UU does not utilize RREP-ACK messages since local repair has not been implemented yet. However, tracing support is in place. For RREP-ACKs, a line in the trace log would end in the following way:

```
... [0x4] (RREP-ACK)
     (1)

... -P aodvuu -Pt 0x4 RREP-ACK
                  (1)
```

`(1)`: Packet type. 0x4 = RREP-ACK.

It should be noted that RREP-ACK messages are not logged at all by the AODV routing agent supplied with ns-2, possibly because of the old version of the AODV draft (version 6) that it adheres to.

6.7.12 Excluded features and modules

The non-applicable features of AODV-UU in ns-2 simulations have been reviewed in section 6.5.8. These features were made unavailable in the ported version of AODV-UU by *not* binding the corresponding configuration variables to the OTcl environment, and enforcing their values in the constructor of the `AODVUU` class.

Since the AODV-UU routing agent does not perform any kernel interaction or ICMP message sending, the kaodv, k_route, libipq and icmp modules were not used. Calls to functions of these modules where excluded using `#ifndef NS_PORT` directives in the source code.

6.7.13 Confi guration and usage

Configuration of the AODV-UU routing agent is done by changing the values of its instance variables. The available instance variables are described below.

- `unidir_hack_` determines whether uni-directional links should be detected and avoided.
- `rreq_gratuitous_` determines whether the *gratuitous RREP* flag should be set in RREQs. This flag is described in section 3.6.3.
- `expanding_ring_search_` determines whether *expanding ring search* should be used for RREQs.
- `receive_n_hellos_` requires that a certain number of HELLO messages should be received from a node before it is treated as a neighbor. If used, it should be set to a value greater than or equal to 2.
- `hello_jittering_` determines whether jittering of HELLO messages should be used.
- `wait_on_reboot_` determines whether a 15-second wait-on-reboot delay should be used on startup.
- `log_to_file_` determines whether a "general" logfile should be created.
- `debug_` determines whether the events of the "general" logfile should be printed to the console (stdout).
- `rt_log_interval_` determines the interval between loggings of the internal routing table of the AODV-UU routing agent. The value is specified in milliseconds, and a value of 0 disables routing table logging.

In general, a value of 0 disables an option and a value of 1 enables it. No checking of the values is performed; it is the responsibility of the user to specify sensible values.

Default confi guration

The default configuration of the AODV-UU routing agent is shown in Table 6.1. The setting of these values is performed in ˜ns/tcl/lib/ns-default.tcl. It should be noted that the HELLO jittering and wait-on-reboot settings differ from the default settings of the conventional AODV-UU routing daemon; the reason for this is that the AODV-UU routing agent applies a certain amount of jittering to *all* broadcast packets, and that a wait-on-reboot phase would be purely superfluous in simulations.

Table 6.1: AODV-UU routing agent default configuration

Option	**Value**	**Comment**
unidir_hack_	0	Off
rreq_gratuitous_	0	Gratuitous flag not set
expanding_ring_search_	1	Uses expanding ring search
receive_n_hellos_	0	Off
hello_jittering_	0	No HELLO jittering
wait_on_reboot_	0	No wait-on-reboot
debug_	0	No general logging to stdout
rt_log_interval_	0	No routing table logging
log_to_file_	0	No general logging to logfile

Usage

Changing of the AODV-UU routing agent configuration options should be performed in the simulation scenario script *before* the simulation is started with the `run` command of the simulator instance. This is preferrably done during node creation, as shown below. Also, for the AODV-UU routing agent to be used in a simulation, the `-adhocRouting` option of the `node-config` command of the simulator instance should be set to “AODVUU”.

```
...
$ns_ node-config -adhocRouting AODVUU

for {set i 0} {$i < $val(nn) } {incr i} {
    set node_($i) [$ns_ node]
    $node_($i) random-motion 0         ;# disable random motion
    set r [$node_($i) set ragent_]     ;# get the routing agent
    $r set debug_ 1
    $r set rt_log_interval_ 1000
    $r set log_to_file_ 1
}
...
```

Note how the routing agent is fetched from the mobile node; it is referenced by the `ragent_` instance variable of the node. After the routing agent instance has been retrieved, the values of its instance variables can be changed as desired.

Compile-time confi guration

The AODV-UU routing agent can also be set to either use link layer feedback or to use HELLO messages. This is determined during compilation, and is described in section 6.7.19.

6.7.14 Extending the AODV-UU routing agent

Because of the modular construction of the `AODVUU` class, it is easy to extend the AODV-UU routing agent whenever new modules are added to AODV-UU. The necessary steps are described below. It is assumed that the module to be added is written in C, but some of the steps apply to C++ modules as well.

Module construction

- In its .c file, the module should only include aodv-uu.h if NS_PORT is defined, *not* any other modules. Standard includes are an exception; they are allowed regardless of the NS_PORT definition.
- All functions of the module should be prepended with the NS_CLASS macro, to make them member methods of the `AODVUU` class.
- In its .h file, the module should group standard includes and global datatypes using the NS_NO_GLOBALS macro. Such code segments should be included by the preprocessor if this macro is *not* set.

  Also, the module should group its function declarations using the NS_NO_DECLARATIONS macro. Such declarations should be included by the preprocessor if this macro is *not* set.

Module addition

- The module should be added to either the SRC_NS or the SRC_NS_CPP variable of the AODV-UU Makefile (described in section 6.7.19), depending on whether the module is written in C or C++.
- The header file of the module should be added outside the `AODVUU` class in aodv-uu.h, to extract global data types, defines and global declarations.

  During this extraction, the NS_NO_GLOBALS macro should be undefined, and NS_NO_DECLARATIONS defined.
- The module should be included inside the `AODVUU` class, to extract method declarations. This requires the macro defined by that module (e.g. AODV_MODULE_H) to be undefined with `#undef AODV_MODULE_H` first, since parts of the module were already included on a global level, and hence, the macro was defined.

  During this extraction, the NS_NO_GLOBALS macro should be defined and NS_NO_DECLARATIONS undefined.

Module compilation

No extra steps are necessary for the compilation. Since the module was added to the AODV-UU Makefile, it will automatically become a part of the AODV-UU routing agent in ns-2.

6.7.15 General C vs C++ issues

During the porting process, a number of conversion issues were encountered and had to fixed in the original source code. These issues were merely a result of the slightly fuzzy semantics of the C programming language (compared to C++), and most of them were only warnings. A detailed reference on C vs C++ issues can be found in [51].

- Structs in C++ are classes. As such, their names should appear before the opening curly bracket, and the class definition should end with a semicolon following the closing curly bracket. This was achieved by using `#ifdef NS_PORT` directives, modifying the definition of the struct as needed.

- The C++ compiler complained about several implicit type casts. This was solved by performing explicit type casts, after ensuring that these type casts would not affect the functionality of AODV-UU in an unexpected way. The most common instances of this issue were implicit type casts of `void *` pointers from `malloc()` calls.

- The C++ compiler did not allow the `AODVUU` class name to be part of the timer handler `typedef` in timer_queue.h. To solve this, the complete `typedef` was manually entered for the `handler` field of the `timer` struct.

- Initializations of member variables are not allowed inside the definition of a class. This required the variables that were moved from the different modules into the `AODVUU` class to be initialized in the constructor of the `AODVUU` class instead.

In general, the original AODV-UU source code did not cause very much trouble during the porting process. This can be seen as an indication of good software development and good adherence to the ISO C standard.

6.7.16 Platform portability issues

During the porting process, the platform portability of the AODV-UU routing agent was investigated. AODV-UU relies on a Linux environment for its network-related datatypes, e.g. those defined by <net/in.h> and <net/if.h>. This causes problems when AODV-UU is to be compiled on other platforms, since the definitions contained in these files differ between platforms.

Another large problem is the lack of an endian.h header file on most non-Linux platforms. This file determines how multi-byte values are stored in memory, which depends on the architecture of the machine. A prototype solution for this problem, a simple endian.c program *generating* an endian.h file (which could then be included during compilation), was tried and found to work satisfactorily. The source code of this program is available from [55]. However, it should be noted that this approach does not always work for *cross-compilation*, where compilation is performed on a machine whose architecture is different from that of the target machine.

Attempts were made to compile the AODV-UU routing agent on an UltraSPARC machine running Sun Solaris 8. The endian problem was solved using the endian.c program, but the compilation failed due to problems with the network-related datatypes used by AODV-UU. More research needs to be done in this area for AODV-UU to become portable to other platforms than the Linux platform.

6.7.17 Miscellaneous source code changes

In the previous sections, a number of changes and additions to the existing AODV-UU and ns-2 source code have been described. In addition to those, the following modifications were made:

- A `create-aodvuu-agent` instance procedure, similar to the existing `create-aodv-agent` instance procedure, was added to the `Simulator` OTcl class in ˜ns/tcl/lib/ns-lib.tcl. This procedure creates and installs an AODV-UU routing agent in a node.

- The `PT_AODVUU` packet type was added to the `recv()` method of the `PriQueue` class in ˜ns/queue/priqueue.cc, to allow AODV-UU routing control packets to be prioritized when a `Queue/DropTail/PriQueue` priority queue is used. Also, the `PT_AODVUU` packet type was added to the `prq_assign_queue()` method to classify AODV-UU packets as routing protocol packets.

- "AODVUU" was added as a possible routing agent choice in the `create-wireless-node` instance procedure in ˜ns/tcl/lib/ns-lib.tcl. The choice of AODV-UU as the routing agent results in a call to the `create-aodvuu-agent` instance procedure, described earlier.

- An `init` instance procedure of the `Agent/AODVUU` class in OTcl was added to ˜ns/tcl/lib/ns-agent.tcl. The purpose of this procedure is to supply the routing agent with OTcl commands during initialization.

- The call to `random()` for HELLO message jittering in the aodv_hello module was replaced with a call to the `Random::integer()` method of ns-2, using an `#ifdef NS_PORT` directive. The reason for this is that `random()` is not portable, and therefore, the built-in random generator of ns-2 must be used instead.
- The routing table logging in `print_rt_table()` of the debug module was modified to flush its output after printing each line. Otherwise the line buffer could overflow, resulting in a segmentation fault during large simulations (with a large number of precursors appearing in the routing table).

6.7.18 Locating source code changes

The changes made to the original AODV-UU and ns-2 source code during the porting are easy to locate, because of the macros used for the selective compilation of the AODV-UU routing agent, and the comments provided where changes have been made. In the ns-2 source code, the changes have been marked with "AODV-UU:" comments and the usage of the NS_PORT and AODV_UU macros. In the AODV-UU source code, the changes have been marked with "NS_PORT:" comments and the usage of the NS_PORT macro.

6.7.19 The AODV-UU Makefi le

The AODV-UU Makefile handles the compilation of the AODV-UU routing agent (as well as the compilation of the AODV-UU routing daemon). It is called *from the Makefile of ns-2* with suitable parameters, allowing the compilation to be performed as if the AODV-UU routing agent had been compiled directly from the ns-2 Makefile. It should be noted that the AODV-UU routing agent can *not* be compiled manually using only the AODV-UU Makefile; the compilation needs the aforementioned parameters supplied by the ns-2 Makefile. The reason for this split compilation is that ns-2 assumes all source code to be .cc files, not .c files as is the case with most modules of AODV-UU. By letting the AODV-UU Makefile handle compilation of the AODV-UU routing agent, this problem was circumvented.

Routing agent packaging

The AODV-UU routing agent is packaged as a *library*, libaodvuu, during compilation. For this, the `ar` archiver utility is used. The libaodvuu library contains all object files of AODV-UU (one per module), and can be included during linking of ns-2. This approach was chosen to minimize the changes required to the ns-2 Makefile.

Confi guration options

To configure the AODV-UU routing agent to use link layer feedback instead of HELLO messages, the AODVUU_LL_FEEDBACK macro should be defined. This is done using the EXTRA_NS_DEFS variable in the AODV-UU Makefile.

6.7.20 Integrating AODV-UU with ns-2

In the following subsections, the compilation issues associated with the integration of the AODV-UU routing agent into ns-2 are described.

Makefi le modifi cations

The ns-2 Makefile is generated by the `configure` utility of the system during installation, using Makefile.in as a template. Therefore, the changes needed for compiling ns-2 with AODV-UU support were added to Makefile.in rather than to a Makefile. These changes are listed below.

- An AODV_UU_DIR variable was defined, specifying where the directory with the AODV-UU source code is located, relative to the ns-2 directory, ˜ns/. The value of this variable was set to "aodv-uu", indicating that the AODV-UU source code directory should be ˜ns/aodv-uu.
- The DEFINE variable was changed to include "-DAODV_UU" and "-DNS_PORT", the macros needed for specifying that AODV-UU support should be added in the ns-2 source code and that AODV-UU should be compiled as an ns-2 routing agent.
- The LIB variable was changed to include the AODV-UU directory in the library path, and the libaodvuu library. This ensures that the AODV-UU routing agent is incuded during the linking process.
- A phony aodv-uu target was added, so that the AODV-UU routing agent will be compiled (i.e., the AODV-UU Makefile is called) each time ns-2 is compiled.

  Three variables are passed from the ns-2 Makefile to the AODV-UU Makefile during compilation; NS_DEFS (the defines used by the ns-2 Makefile), OPTS (the compiler options used by the ns-2 Makefile) and NS_INC (specifying the absolute path to the ns-2 directory).

  These are used by the AODV-UU Makefile to compile the AODV-UU routing agent as if it had been compiled directly from the ns-2 Makefile.
- A phony aodv-uu-clean target was added to allow cleaning of compiled files in the AODV-UU directory to be performed from the ns-2 Makefile. This target switches to the AODV-UU directory and issues `make clean` there, i.e., the actual cleaning is performed by the AODV-UU Makefile.

  The aodv-uu-clean target was also added as the first dependency of the existing clean target, so that the AODV-UU directory is cleaned whenever `make clean` is issued in the ns-2 directory.
- The aodv-uu target was added as the first dependency of the ns target, so that AODV-UU is compiled whenever ns-2 is compiled.

Distribution of modifi cations

It was decided that all changes made to the original ns-2 source code during the porting of AODV-UU should be distributed as a *patch*, which can be applied to a fresh copy of ns-2 using the `patch` utility of the system. This eliminates the need for manually replacing or modifying files in the ns-2 source code tree when AODV-UU support is to be installed, e.g. by an end user.

Compilation instructions

Compilation of the AODV-UU routing agent is very straightforward. The required steps are described below.

- A fresh copy of ns-2, version 2.1b9, is needed. It should be unpacked and installed on the target system, e.g. into a ˜ns/ directory.
- A directory containing all the AODV-UU files should be created as "aodv-uu" *below* the ns-2 directory, i.e., as ˜ns/aodv-uu.
- Any desired changes to the AODV-UU Makefile should be made (see section 6.7.19).
- The ns-2 source code tree should be patched, using the patch supplied with AODV-UU:

  ```
  cd ~ns/
  patch -p1 < aodv-uu/ns-2.1b9-aodv-uu-0.5.patch
  ```

  This introduces all necessary changes to the ns-2 source code tree for AODV-UU to be supported.

- Finally, ns-2 should be re-compiled. The following sequence of commands is recommended to ensure that all portions of ns-2 are re-compiled:

```
cd ~ns/
./configure
./make distclean
./configure
./make
```

After successful compilation, the AODV-UU routing agent can be used as described in section 6.7.13.

6.8 Bugs found by simulation

During initial testing of the AODV-UU routing agent, some bugs in AODV-UU were found and corrected. One of the most serious bugs encountered was a *log buffer overflow* in the debug module, causing the simulator to crash with a segmentation fault when a large simulation was conducted and routing table logging had been enabled. It turned out that a large number of precursors appearing in the routing table caused the program to write log data outside its log buffer. This bug was solved by flushing the routing table log after each line.

Simulations also revealed a bug in the processing of AODV messages, where the reception of a message could cause buffered packets to be scheduled for transmission even though an active route to the destination did not exist. This lead to repeated buffering and un-buffering of packets.

Finally, errors related to *RERR processing* were found and corrected by the author of AODV-UU. It was concluded that the simulations helped finding obscure bugs that probably would not have been found otherwise.

6.9 Future work

Some future work remains to be done on the AODV-UU routing agent. The issues listed below have not been (fully) addressed, due to time constraints.

- Support for *local repair* should be added as soon as this feature becomes available for the AODV-UU routing daemon.
- The *tagged trace format* should be supported. Although this format is new and relatively uncommon, its importance may increase in the future.
- Support for using the AODV-UU routing agent in wired and wired-cum-wireless scenarios should be investigated. The primary mission so far has only been to use AODV-UU as an ad-hoc routing agent for wireless mobile nodes.
- The portability of AODV-UU should be extended so that it can be run on other platforms than Linux, e.g. Sun Solaris. This is an important step to increase its usage in the research community.
- Possibly, the configuration parameters of the AODV-UU routing agent should be checked during runtime, e.g. by introducing a special configuration command and enforcing its usage. This would prevent users from supplying incorrect parameters to the routing agent.

6.10 Summary

The porting of AODV-UU to run in the ns-2 network simulator was successfully accomplished. By using the *preprocessor* and *macros* for source code extraction, an AODV-UU routing agent was constructed that uses the same source code as the conventional Linux version, with only minor differences in packet handling. This approach also allows for *easy switching* between compilation of the ported version and the conventional version. The separate compilation of software modules in AODV-UU was preserved by letting the AODV-UU Makefile compile these modules, given certain compilation parameters from the ns-2 Makefile. Finally, the AODV-UU routing agent was integrated into ns-2 by packaging it as a *library*, which is included during linking.

During initial testing in the simulator, bugs were found that probably would not have been possible to find in real-world experiments with AODV-UU. This confirms that simulations and real-world experiments complement each other – in more than one way. Further testing of the AODV-UU routing agent can be found in Chapter 7.

Chapter 7

Testing the Functionality of AODV-UU in ns-2

7.1 Introduction

After porting AODV-UU to run in the ns-2 network simulator, its functionality had to be verified. To do this, several test-runs were performed using different scenarios supplied with ns-2 version 2.1b9 as well as stand-alone scenarios, and the results were compared to the expected results specified by the documentation for each scenario. Documentation for those scenarios that are part of the ns-2 distribution can be found in [56], and the source code of all scenarios and supplemental scripts mentioned in this chapter is available from [55]. The notion ˜ns followed by a path and a filename refers to the corresponding file in the ns-2 source code tree; the complete source code of ns-2 is available from the ns-2 homepage [8].

7.2 General setup

The scenarios were modified to use the AODV-UU routing agent by changing the routing protocol setting to "AODVUU", and the simulation scenario script files were named or re-named to indicate this. AODV-UU version 0.5 was compiled with the option to use HELLO messages. The default run-time configuration options for AODV-UU were used, except for logging which was also enabled. If not otherwise stated, the scenarios use *de facto* wireless settings, i.e., a `WirelessChannel` wireless channel, a `WirelessPhy` wireless network interface with 2 Mbps bandwidth, the two-ray ground reflection radio propagation model, the `802_11` MAC layer (using 2 Mbps as the data rate both for unicast and broadcast packets), a `PriQueue` priority queue of length 50 as an interface queue, and an omni-directional antenna. All these components come with certain default settings, which can be found in ˜ns/tcl/lib/ns-default.tcl.

7.3 simple-wireless-aodv-uu.tcl

7.3.1 Scenario overview

In simple-wireless-aodv-uu.tcl, communication between two mobile nodes is tested by moving them relative to each other so that one node is not always within radio range of the other one. This scenario is based on ˜ns/tcl/ex/simple-wireless.tcl.

7.3.2 Scenario setup

The scenario sets up a 500 x 500 m flat grid and places two mobile nodes out of each other's radio range. One node has a TCP sink agent attached to it, which will receive (and throw away) any incoming TCP traffic. The other node has an FTP agent connected to its TCP agent, simulating FTP traffic. The nodes are connected to each other through the wireless channel.

7.3.3 Scenario description and results

The FTP traffic is started at time 10 seconds. However, at this time the nodes are too far apart for any communication to take place. At time 50 seconds, the node with the TCP sink starts moving towards the FTP source node, and at time 68 seconds, the nodes are close enough to begin exchanging routing control messages. It has been verified that the routing tables of the two nodes are set up correctly, by using the routing table logging option of AODV-UU and analyzing the routing table logfiles.

At time 100 seconds, the TCP traffic starts to flow, and the TCP sink starts moving away from the FTP source node. This causes the link between them to break at time 116 seconds, and subsequent packets are dropped. The link failure is reflected in the nodes' routing tables at time 119 seconds, and the simulation ends at time 150 seconds.

7.3.4 Conclusions

Analysis of the trace log from the simulator as well as the AODV-UU logfiles shows that the scenario works as expected using AODV-UU. The events in the trace log correspond to those of the scenario documentation.

7.4 wireless1-aodv-uu.tcl

7.4.1 Scenario overview

In wireless1-aodv-uu.tcl, three nodes are placed in such a way that two of the nodes need to route their traffic through an intermediate node to reach each other. This scenario is based on ˜ns/ns-tutorial/wireless1.tcl.

7.4.2 Scenario setup

The movement and traffic patterns are loaded from separate files, scen-3-test and cbr-3-test, which are part of the ns-2 distribution. The movement pattern file covers random-waypoint movements over an area of 670 x 670 m, and the traffic pattern file sets up connections between the three nodes during the duration of the simulation (400 seconds). The layout of the scenario is visualized by NAM (Network Animator) in Figure 7.1.

7.4.3 Results and conclusions

By analyzing the AODV-UU routing table logfiles, it was concluded that the two outermost nodes always use the intermediate node to send traffic to each other, and that the intermediate node is the only node ever within radio range of both the other nodes simultaneously.

7.5 The Roaming Node scenario

In order to validate the functionality of AODV-UU in ns-2 further, a scenario called "Roaming Node" was adopted. This scenario was originally used for real-world experiments concerning the "Gray Zone Problem" [58], when using HELLO messages as periodic node announcements with AODV-UU together with wireless communication based on IEEE 802.11b. Briefly, the Gray Zone Problem refers to the fact that HELLO messages, being sent as broadcast packets, may reach a node although it is impossible for data packets to get through (the latter being sent as unicast packets at a higher bit rate). This causes nodes in such gray zones to falsely believe that they have proper connectivity to some of their neighboring nodes, resulting in packet loss when attempts are made to send data packets. The reasons for choosing this specific scenario to test the functionality of AODV-UU in ns-2 were the following:

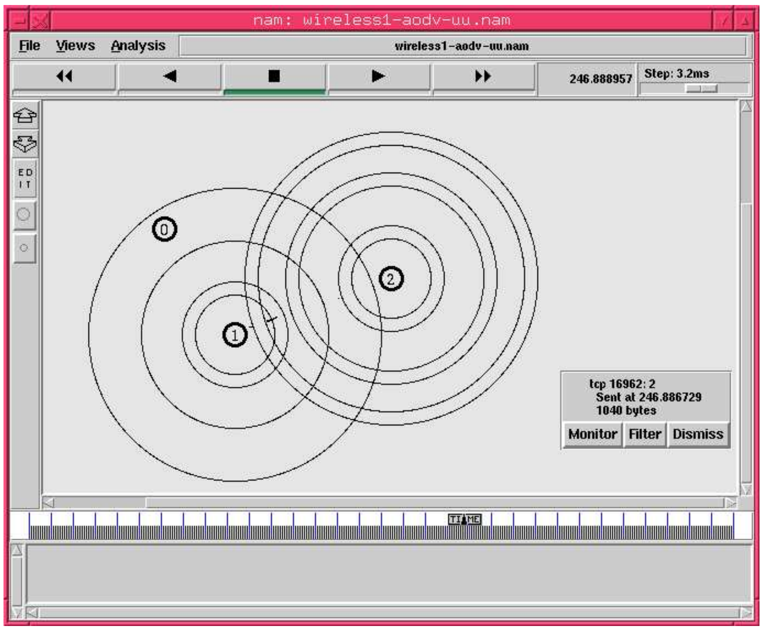


Figure 7.1: Layout of the wireless1-aodv-uu.tcl scenario. All network traffic in this scenario has to pass through an intermediate node.

- The real-world results of the scenario have been carefully documented in [58]. This includes diagrams and detailed descriptions of real-world results using AODV-UU.
- The scenario is small enough to easily be understood, yet complicated enough to test multi-hop routing and observing the behavior of the routing agent when link failures occur.
- It is a good starting point for evaluating if simulation results correspond well to real-world results, and getting to know which problems that may arise when transferring a real-world scenario into one in the simulator.

7.5.1 Scenario overview

The Roaming Node scenario consists of four nodes (denoted GW, C1, C2 and MN), of which three are stationary (GW, C1 and C2) and the fourth one (MN) is a mobile node, "roaming" the environment while sending Ping packets to GW. The GW node sends a Ping reply back to the mobile node for each Ping packet it receives.

The nodes are placed in such a way that each stationary node is within radio range of only its nearest neighbor(s). This will force both the mobile node and the GW node to send their Ping packets through another stationary node in order to reach each other during some parts of the scenario. The layout of the Roaming Node scenario is shown in Figure 7.2.

7.5.2 Scenario setup

Measurements were taken from the real-world environment where the Roaming Node scenario originally took place, and were transferred into coordinates in the simulation scenario script file, roaming-aodv-uu.tcl. Since the flat grid topography in ns-2 uses metres as its unit of measurement, this was easily done.

To model the wireless communication of the Roaming Node scenario more correctly in the simulator, some changes had to be made to the default values of the wireless physical layer. Radio characteristics of the wireless network cards used in the real-world experiments (ORiNOCO 802.11b PC Cards) were taken from [59], and the affected values were set in the simulation scenario script file. Most importantly, the receive threshold for all nodes in the scenario was set to a value corresponding to a radio range of 22.5 m. This is roughly enough for the

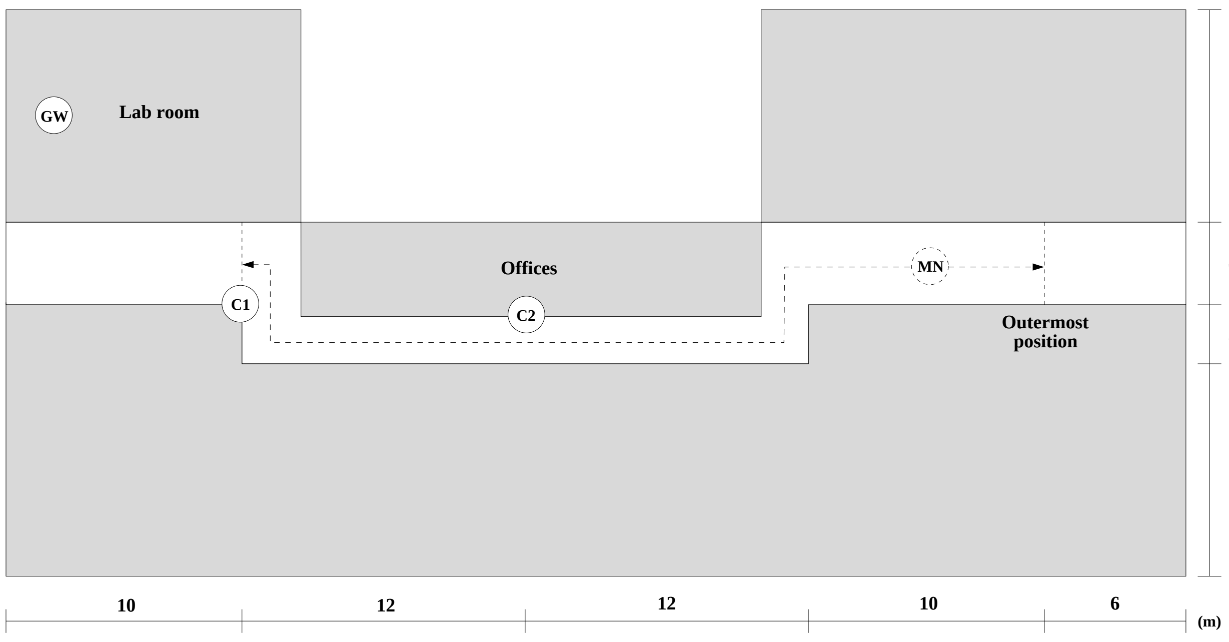


Figure 7.2: Layout of the Roaming Node scenario

mobile node to be able to reach a stationary node at its outermost position during the simulation. Calculation of the receive threshold was made using the `threshold` utility shipped with ns-2. The interface queue was set to prioritize routing protocol control packets.

The existing Ping agents of ns-2 were used to generate and handle the Ping packets at the MN and GW nodes. The repeated creation of Ping packets at the mobile node was made possible by scheduling ns-2 to perform OTcl commands on the Ping agent of the mobile node, instructing it to send a Ping packet at a regular interval. The trace support in ns-2 was modified to perform tracing of Ping packets, as this was not enabled by default. In the real-world experiments, the Ping packets have the Record Route option turned on to allow the route taken to be analyzed. For the simplicity of the Ping agents at the MN and GW nodes, the route was required to be observed manually (i.e., the Ping packets do not contain this information themselves), but the size of the Ping packets was set to be equal to that of the real-world Ping packets with Record Route; 580 bytes including the IP header. Manual tracing of the route taken by a Ping packet could be conducted using the unique ID number of each such packet in the trace log that the simulator generates.

7.5.3 Scenario description and simulation results

The mobile node starts out at the same position as C1, constantly pinging the GW node with Ping packets at a rate of 10 packets/second. It expects to get Ping replies back from the GW node. Since the GW node is close enough to the mobile node, the initial link between them is a direct (one-hop) link.

Between times 40-55 seconds, the mobile node moves from C1 towards C2. When the mobile node comes close to C1, its former link to GW will break because GW is out of range. This occurs at time 52.4 seconds, causing the subsequent Ping packets to be dropped. The mobile node realizes the link failure at time 54.068 seconds (the last HELLO message from GW was received at time 52.019 seconds), and broadcasts a route request (RREQ) message at time 54.1 seconds. It receives a reply from node C1 at time 54.166 seconds, providing it with a route to GW through C1 (2 hops). During the link failure, the mobile node has buffered one Ping packet which is now immediately sent out. The pinging operation can then continue as usual. All in all, the link failure causes 17 Ping packets to be dropped between times 52.43-54.03 seconds.

The mobile node remains at C2 between times 55-95 seconds, and then continues to move towards its outermost position. The movement is performed during times 95-125 seconds. During this movement, the mobile node loses contact with C1, again because of the limited radio range. Packets are first dropped at time

107.43 seconds, and the mobile node detects the link failure at time 109.168 seconds (the last HELLO message from C1 was received at time 107.119 seconds). It broadcasts a route request (RREQ) message at time 109.2 seconds, and receives a route reply (RREP) message from C2 at time 109.304 seconds. This gives the mobile node a route to GW through C2 (3 hops). Two buffered packets are sent out as soon as the new route has been established, after which the pinging operation continues as usual. All in all, the link failure causes 18 Ping packets to be dropped between times 107.43-109.13 seconds.

Between times 125-165 seconds, the mobile node remains at the outermost position, and then starts moving back along the same way towards C2. Between times 195-235 seconds, the mobile node remains at C2. It then moves towards C1 during the time 235-250 seconds, after which it stays at C1 until the scenario ends, at time 290 seconds. On its way back towards C1, the route is optimized along the way as the mobile node receives HELLO messages from the stationary nodes. Specifically, the mobile node receives a HELLO message from GW when it comes within its radio range at time 238 seconds. It then replaces the next hop towards GW with the address of GW (i.e., a direct link).

Finally, it should be noted that no packet loss occurs between times 125-290 seconds, since the mobile node has constant connectivity to GW (either directly or through C2) during this period of time.

7.5.4 Comparisons with real-world results

In the paper discussing the Gray Zone Problem [58], several modifications to AODV-UU are examined together with their effect on reducing gray zones. It is also mentioned that simulations in ns-2 do not suffer from the gray zone problem, since the simulator does not differentiate between broadcast and unicast packets in terms of radio range. The results of the Roaming Node scenario simulation are in most respects very similar to those of the real-world results described in [58]. The mobile node is forced to find a new route towards GW on two occasions during its movements towards the outermost position. During the time between a link failure and the time when the mobile node realizes the link failure, Ping packets are dropped. However, there are differences as well; an analysis of the trace file from the simulator gives the statistics shown in Table 7.1. The corresponding real-world results are shown in Table 7.2.

Table 7.1: Roaming Node Ping statistics (simulation)

Measurement	**Value**
Pings sent by mobile node	2900
Pings received by GW	2865
Ping replies sent by GW	2865
Ping replies received by mobile node	2865
Ping success ratio of mobile node	98.8%

Table 7.2: Roaming Node Ping success ratio (real-world)

Experiment	**Ping success ratio**
Unmodified AODV-UU	91.9%
3-Hellos	98.0%
Neighbor set	97.7%
SNR threshold	99.1%

Clearly, AODV-UU in ns-2 suffers from less packet loss than the unmodified version of AODV-UU in the real-world experiments. This is due to the absence of gray zones in simulations, as mentioned earlier. The effect of this is less packet loss in general during the entire simulation, and no packet loss at all while the mobile node is moving back towards its starting position.

7.6 Other differences between simulations and real-world experiments

Not only phenomena such as the gray zone problem differ between simulations and real-world experiments; the fact that ns-2 is a discrete event-driven simulator has its implications too. Specifically, processing of packets in the simulator does not consume any (virtual) time. Only explicitly requested delays, such as delays imposed by links, do. This means that the AODV-UU routing agent in the simulator will not suffer from processing delays like the real-world implementation does. Therefore, simulations with the intention of measuring routing agent delays might not correspond to real-world results.

7.7 Comparisons with existing AODV implementation

An important (and desired) usage of AODV-UU is to replace the existing AODV routing agent of ns-2, allowing users to benefit from compliance with the latest AODV draft. Therefore, AODV-UU was compared to the existing AODV routing agent of ns-2.

This comparison was automated by a Perl script, scale_test.pl. It creates a movement pattern file with random movements and a traffic pattern file with CBR traffic. A number of simulations are performed using these files, and the resulting packet delivery ratios are averaged. This is done for each routing agent, and for different numbers of nodes. The results are then visualized in a graph, displaying the average packet delivery ratios of the routing agents as a function of the number of nodes.

The comparison performed here used a 800 x 2500 m topography, 200 seconds simulation time, CBR traffic with a rate of 6 packets/second and a packet size of 512 bytes. A maximum of 4 simultaneous connections was set. The movements were random-waypoint movements with 1.0 seconds pause time and a maximum mobility of 20 m/s. Each test was performed 10 times to form the average packet delivery ratios. The numbers of nodes tried were 10, 20, 30, 40 and 50. Two test-runs of the scale_test.pl script were performed, with both routing agents configured to either use link layer feedback or HELLO messages. The results are shown in Figures 7.3 and 7.4.

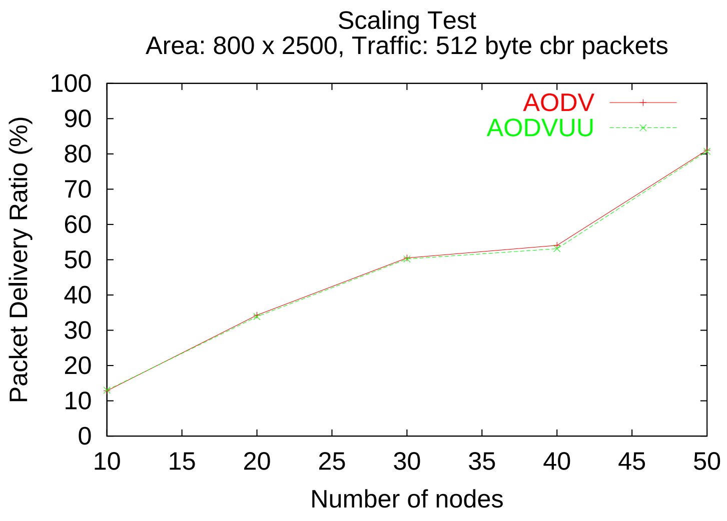


Figure 7.3: Routing agent comparison (link layer feedback)

Both routing agents experience rather low packet delivery ratios due to the size of the topography. Aside from this, Figure 7.3 shows that the AODV-UU routing agent performs on par with the existing AODV routing agent when using link layer feedback. However, the existing AODV routing agent exhibits much worse results when using HELLO messages, as shown in Figure 7.4. The cause of this is failed transmissions at the link layer level, although the reasons for these failed transmissions are not revealed by the trace logs (see section 5.19).

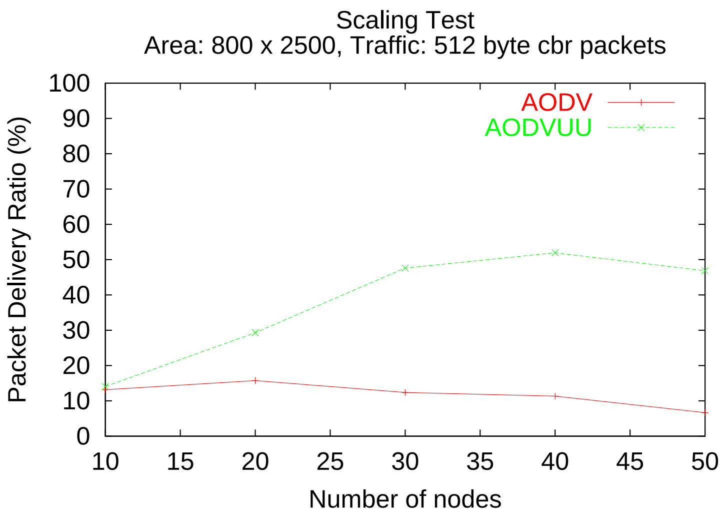


Figure 7.4: Routing agent comparison (HELLO messages)

7.8 Conclusions

The results of all tested scenarios indicate that AODV-UU in ns-2 works as anticipated. It often matches or exceeds the packet delivery performance of the existing AODV routing agent in ns-2, making it a good choice for AODV simulations. Furthermore, it has proven to be possible to transfer a real-world scenario into the simulator and to obtain simulation results with AODV-UU that in most respects are similar to those of real-world experiments. However, one should be aware of that some effects seen in real-world experiments will not show up in simulations, because of limitations in the models of the simulator. One such example is *gray zones*, which could cause packet delivery ratios in simulations to differ from real-world results. This problem is addressed in Chapter 8, where logfiles from real-world experiments are used for minimizing such differences.

Chapter 8

Enabling Trace-based Simulation

8.1 Introduction

In Chapter 6, the porting of the AODV-UU routing protocol implementation to the ns-2 network simulator was described. The ported version of AODV-UU uses the same source code as the conventional version to minimize the differences between simulations and real-world experiments. The tests performed in Chapter 7 indicate that it is possible to perform simulations which resemble real-world experiments, but that the models of the simulator could limit the similarities.

The following sections describe how simulations and real-world experiments can be brought closer to each other by modifying the connectivity of mobile nodes using logfiles from real-world experiments. For this purpose, the APE testbed [9] is used. Its logs from real-world ad-hoc networking experiments are fed through a script to extract connectivity information, and OTcl code is generated to allow the ns-2 network simulator to adjust the connectivity of nodes accordingly.

8.2 The APE testbed

The APE testbed [9, 60] is a Linux-based ad-hoc protocol evaluation testbed, suitable for conducting wireless ad-hoc networking experiments. It uses script-based scenario choreography for giving participating users (i.e., the nodes) movement instructions on screen, thereby allowing scenarios to be repeated with similar results. All network traffic during a scenario is logged, and the logs can be uploaded to a *collect node* when the scenario has ended. The collect node synchronizes the logs and produces a combined logfile containing all information gathered by all nodes during the entire scenario. Post-run analysis scripts allow useful statistics to be extracted, such as packet loss ratio, path optimality and *virtual mobility*, a mobility metric that can be used for acquiring unique "fingerprints" of scenarios. It is also possible to visualize a scenario and the connectivity between nodes as a function of time, with a special *APE-view* utility.

The APE testbed comes with several ad-hoc routing protocols pre-installed (AODV-UU [6], Mad-hoc AODV [38], Kernel AODV, LUNAR [62] and INRIA OLSR [61]), but it is possible to use almost any ad-hoc routing protocol implementation that runs under Linux. Therefore, new protocol implementations are expected to be added to the APE testbed as they become available.

8.3 APE logging

The APE testbed uses *superspy*, a modification of the *spy mode* extensions to the IEEE 802.11 WaveLAN drivers, to log signal levels and signal noise for all packets picked up by the wireless network interface during an APE scenario. This information is saved to a /var/log/superspy.log logfile.

By uploading the logfiles of the nodes to the collect node and running the mk_eth_log.pl script supplied with the APE testbed, an eth.log logfile is generated containing signal-to-noise ratio (SNR) values on a per-packet basis for all nodes in the APE scenario. Another script supplied with the APE testbed, mk_step.pl, is useful for generating average SNR value listings with a certain step length. The resulting logfile, an *eth-step* logfile, contains average SNR values between all nodes in the APE scenario. It could e.g. be used for determining connectivity between nodes, with the step length effectively deciding how fast connectivity changes should be detected.

8.4 Modeling connectivity in ns-2

In wireless ns-2 simulations, a radio propagation model determines the connectivity between nodes. Correct placement and movement of nodes is important for the simulation to resemble a real-world scenario. Still, simulation results may not correspond "close enough" to real-world results, e.g. in terms of packet delivery. The main reasons for this are limitations in the models of the simulator and inaccuracies or errors in simulation scenario scripts, causing inappropriate connectivity or lack of connectivity between nodes. To cure this, a number of actions should be considered:

- Double-checking the parameters of the radio propagation model and re-calculating the *receive threshold* of the wireless network interface.
- Changing the radio propagation model. It may be the case that the radio propagation model is inappropriate for the chosen scenario.
- Visualizing the simulation scenario in NAM (Network Animator) to spot possible node placement and movement errors.

If the differences persist, one could attempt to adjust the connectivity between nodes manually. This could e.g. be done by adjusting the communication range of nodes during the simulation, or by introducing link-layer errors to undesired packets.

8.4.1 Communication range adjustment

One way of adjusting connectivity between nodes is to modify their communication range. This is done by changing the settings of the `Phy/WirelessPhy` wireless network interface, as described in section 5.17. If a specific communication range is desired, the value of e.g. the *receive threshold* of the network interface must be calculated using the `threshold` utility of ns-2. If instead a simple on/off connectivity switch suffices, the parameters of the wireless network interface could be set to extreme values to fake connectivity and lack of connectivity, respectively. However, extreme values give no actual guarantees regarding connectivity, and hence, this approach is not very appealing.

8.4.2 Error model usage

A better way to adjust connectivity between nodes would be to use *error models*, as described in section 5.14. Error models allow link-layer errors to be introduced into a simulation, and in the case of wireless simulations, error models can be applied independently to incoming and outgoing packets. The ability of error models to discard entire packets as their unit of error is useful in connectivity modeling, since lack of connectivity means that packets simply should be discarded.

However, an error model alone cannot guarantee connectivity – only lack of connectivity. This is because of the way network interfaces of mobile nodes process packets. A radio propagation model is consulted *after* the error model to determine whether or not an incoming packet should be passed on to the MAC layer. Hence, an error model could model connectivity (and lack of connectivity) only if it is used together with a "generous enough" radio propagation model. The possibility of letting an error model modify the signal strengths of packets is not considered here, since it would render the radio propagation model rather useless.

8.4.3 Existing error models

Two of the existing error models in ns-2 were investigated for the purpose of modeling connectivity changes between nodes. The *ErrorModel* error model offers basic error model functionality, where packets can be dropped with a certain probability. Calls to this error model could be scheduled (changing the error rate to 100% to simulate loss of connectivity), but this would discard all packets, not just packets from a certain node. Hence, this error model is not flexible enough for modeling arbitrary connectivity changes between nodes.

An interesting alternative is the *multi-state* error model, where a matrix of probabilities determines the transitions between different error models. Such a matrix could be constructed *a priori*, based on connectivity information from the APE logfiles. However, the drawback is the resulting large number of additional error models that would have to be constructed. For a simulation with n nodes, the number of required additional error models would be $n^2 - n$, for modeling connectivity between all source - destination pairs (excluding the node itself). Also, one copy of the multi-state error model would be needed for each of the n nodes. Obviously, such an excessive usage of error models would be wasteful.

8.4.4 Conclusions

From the above discussions, it should be clear that neither communication range adjustment nor any of the existing error models in ns-2 are suitable for modeling connectivity changes between mobile nodes. However, a custom-made error model, together with a not too restrictive radio propagation model, would allow connectivity changes between mobile nodes to be modeled properly.

8.5 Proposed solution

The solution proposed for enabling trace-based simulations is the following. A *custom-made error model* should be constructed that allows connectivity to be switched on and off for different packet sources (nodes) in the simulation, on an on-demand and per-source basis. The error model should be inserted between the wireless channel and the network interface of mobile nodes, to *filter incoming packets* before they reach the network interface. Connectivity information should be extracted from the APE testbed logfiles, and be used for generating an *OTcl source code skeleton* that reflects connectivity changes using the custom-made error model. The OTcl source code skeleton should be possible to include in a simulation scenario script. The implementation details of this solution are described in the following sections.

8.6 Related research

Some research related to trace-based simulations can be found in [63], where wireless channel characteristics are collected during real-world experiments and then re-used for performing trace-based *emulations*. An improved two-state error model is constructed and validated through test-runs with a corresponding error model in ns. The results indicate that such a model allows the characteristics of a wireless channel to be captured with good accuracy. Similar discussions and results on this topic can be found e.g. in [64] and [65].

However, it should be noted that these models typically rely on more parameters than just signal strength or signal quality, and that the availability of such additional parameters heavily depends on the data gathering during real-world experiments. In this regard, the APE testbed is somewhat limited, since its main focus is on connectivity between nodes rather than wireless channel characteristics.

8.7 The SourceDrop error model

The proposed error model for modeling connectivity changes is called the *SourceDrop* error model, since it drops packet depending on their source. The OTcl class name of this error model is `ErrorModel/SourceDrop`. The source code, sourcedrop_errmodel.{cc, h}, is available from [55].

8.7.1 Functionality

The SourceDrop error model examines the source MAC addresses of all packets that it receives, and compares them to addresses in a *blocking list*. If a match is found, the packet is discarded. Otherwise, the packet is let through, i.e., is sent to the output of the error model for further processing by the node. Hence, the functionality of this error model resembles that of the `mackill` utility included with the APE testbed [4]. It should be noted that MAC addresses are equivalent to node IDs because of the simple one-to-one address mapping performed by the ARP module of mobile nodes in ns-2.

The blocking list can be dynamically changed during a simulation by issuing the two commands `add <addr>` and `remove <addr>`, which add (block) and remove (unblock) a certain source address. A special `block-all` command can be used to block all incoming packets, and a `reset` command clears the blocking list and turns off the `block-all` option. Unlike other error models, the SourceDrop error model does not allow its unit of error to be set to anything but entire packets.

8.7.2 Technical details

The blocking list is implemented as a linked list of ns-2 address entries of the `nsaddr_t` type. CTS and ACK messages of the `802_11` MAC layer are always let through unless complete blocking of packets has been requested, since their source MAC addresses for some reason always are set to zero rather than the actual source. Installation of the SourceDrop error model in a mobile node is done by following the procedure described in section 5.14. Finally, the error model is intended to be used for incoming packets only, since it examines the *source* MAC address of packets.

8.8 APE logfile processing

As mentioned previously, the APE testbed includes scripts for extracting radio signal quality information gathered during an APE scenario. The mk_step.pl script is of special interest, since it allows average SNR values to be calculated over time intervals whose length can be specified. The results are written to an eth-step logfile, which can then be used for determining connectivity, since a SNR value of zero indicates that a certain node cannot be heard.

For the extraction of connectivity information from the APE logfiles, a special Perl script was constructed. The source code, ethstep2sourcedrop.pl, is available from [55]. This script takes an eth-step logfile as input and processes it line by line. SNR values are stored in a *connectivity matrix* and are updated as the eth-step logfile is read. By comparing each line of SNR values to the current values of the connectivity matrix and a *connectivity threshold*, link failures and re-establishments can be determined. Since each line of the eth-step logfile also contains a timestamp, the times of any such connectivity changes are known.

The ethstep2sourcedrop.pl script produces an OTcl source code skeleton as output, which can be included in an ns-2 simulation scenario script. Special mapping variables are provided that allow the user to easily specify the variable names of the simulator instance and nodes in a simulation. The connectivity changes are carried out by scheduling calls to the SourceDrop error model instance of nodes, and comments are provided for all connectivity events to ease verification. After including the OTcl source code skeleton in a simulation scenario script, the scenario can be executed as usual by ns-2.

8.9 Initial testing

The functionality of the SourceDrop error model was tested by installing the error model in a mobile node that used the AODV-UU routing agent, and selectively blocking packets from other nodes in a simulation scenario script. The logging features of AODV-UU together with ns-2 trace logs verified that the error model worked correctly. The functionality of the ethstep2sourcedrop.pl script was verified by manual inspection of its output and eth-step logs generated by the mk_step.pl script of the APE testbed. It was concluded that the script worked correctly.

8.10 Roaming Node revisited

For investigating the effects of SourceDrop error model usage, the Roaming Node scenario from section 7.5 was once again adopted, since the initial simulation test-run of this scenario indicated a Ping success ratio higher than in the corresponding real-world experiment. It was concluded earlier that this difference mainly was caused by limitations in the models of the simulator.

8.10.1 Simulation scenario setup

The only changes made to the simulation scenario script were the additional installation of the SourceDrop error model for each node and the inclusion (sourcing) of an OTcl skeleton produced by the ethstep2sourcedrop.pl script. The resulting simulation scenario script, roaming-aodv-uu-errormodel.tcl, is available from [55].

Eth-step logs were produced by the mk_step.pl script of the APE testbed using eth.log logfiles from two real-world Roaming Node experiments. For each experiment, three different step lengths were selected; 1.0, 0.5 and 0.1 seconds. This was done to be able to investigate how the step length affects the results, given the fact that network traffic in this scenario consists of 10 Ping packets per second.

8.10.2 Results

The results of the trace-based simulations are compared to the initial simulation (without any error model usage) and the real-world experiment (using an unmodified version of AODV-UU) in Tables 8.1 and 8.2.

Table 8.1: Roaming Node Ping statistics comparison (run 1)

Measurement	**Initial**	**Step 1.0**	**Step 0.5**	**Step 0.1**	**Real**
Pings sent by mobile node	2900	2900	2900	2900	2900
Pings received by GW	2865	2841	2769	2669	$\geq$2665
Ping replies sent by GW	2865	2841	2769	2669	$\geq$2665
Ping replies received by mobile node	2865	2807	2703	2577	2665
Ping success ratio of mobile node	98.8%	96.8%	93.2%	88.9%	91.9%

Table 8.2: Roaming Node Ping statistics comparison (run 2)

Measurement	**Step 1.0**	**Step 0.5**	**Step 0.1**
Pings sent by mobile node	2900	2900	2900
Pings received by GW	2838	2755	2637
Ping replies sent by GW	2838	2755	2637
Ping replies received by mobile node	2804	2712	2615
Ping success ratio of mobile node	96.7%	93.5%	90.2%

These results indicate that it is possible to achieve Ping success ratios similar to those of the real-world experiments by performing trace-based simulations with AODV-UU and the SourceDrop error model. The best results were achieved using a step length of 0.5 seconds, where the difference in Ping success ratios was approximately 1.5%. This is a huge improvement over the results of the initial (non-trace-based) simulation, where the Ping success ratio differed by almost 7%. The asymmetry of links is clearly visible in some of the results, where the packet loss differs on the way to and from the GW node.

8.10.3 Discussion

The step length appears to be critical; a step length too small results in excessive packet loss. The reason for this is that the absence of Ping packets during a large number of small time intervals causes loss of connectivity between nodes, and affects the operation of the RTS/CTS handshake protocol negatively. Hence, the step length must be chosen carefully and with regard to the expected network traffic for the chosen scenario. Judging from the results, the optimal step length for the Roaming Node scenario seems to lie somewhere between 0.1 and 0.5 seconds, although an optimum has not been extensively searched for.

It should also be noted that the application of a single, fixed step length for connectivity modeling in trace-based simulations may not be appropriate for all scenarios. More specifically, in scenarios where network traffic patterns differ temporally and/or spatially (i.e., within the network topology), an adaptive step length could be a better choice, but would also require a more complex connectivity extraction than the one described here.

8.11 Conclusions

By constructing a simple source-based error model for mobile nodes and using it together with AODV-UU and logfiles from real-world experiments, trace-based simulations were performed with good results. More specifically, Ping success ratios were achieved that closely match those of real-world experiments. However, the implicit parameters to this error model – the data extracted from logfiles and the parameters affecting this extraction – require knowledge about the anticipated network traffic and must be chosen carefully. Also, the underlying radio propagation model is of great importance, since it makes the final decision on which packets that should reach the MAC layer of a mobile node.

Finally, error models such as the one devised in this chapter could play an important role in verification of routing agent implementations in ns-2, since they allow connectivity to be explicitly turned off for selected nodes. Such functionality is not available in any existing error models shipped with ns-2.

Chapter 9

Summary and Future Work

In this chapter, the outcome of the master's thesis is summarized, using the proposed goals of section 1.2 as a reference. This is done to verify that the goals have been accomplished, and hence, that the requirements of the original specification have been met. Also, some general requirements for master's theses are reviewed. Finally, notes are made on future work which remains to be done.

9.1 Summary of ns-2 port of AODV-UU

AODV-UU [6] was successfully ported to the ns-2 network simulator [8]. The ported version uses the same source code as the conventional Linux version, with only minor differences in terms of packet handling. This allows simulations to be conducted with the confidence that differences between the two versions are minimal. Test-runs using AODV-UU in ns-2 show that it instead is the models of the simulator that limit the similarities between simulations and real-world experiments. Finally, AODV-UU has proven to be a good replacement for the AODV routing agent supplied with ns-2, achieving good packet delivery performance while adhering to a much more recent version of the AODV draft [7].

9.2 Summary of trace-based simulation

Trace-based simulations using AODV-UU and logfiles from the APE testbed [4] were successfully performed by using a custom-made error model, modeling arbitrary connectivity between mobile nodes, and a script translating connectivity changes in the APE logfiles into error model commands. No radio propagation models were modified during this process, although the custom-made error model could certainly be viewed as an improved version of existing error models in ns-2. Test-runs show that it is possible to achieve packet delivery ratios that come close to real-world results (differing by approximately 1.5%), but that some care is needed to select parameters for the extraction of data from testbed logfiles.

9.3 Comparisons of results

Comparisons of results from simulations and real-world experiments have been performed several times during this master's thesis project. In Chapter 7, results of a *Gray Zone simulation* were compared to real-world results using a simple script to extract Ping statistics (e.g. Ping success ratios) from ns-2 trace files. The same technique was used in Chapter 8 to evaluate the effects of *trace-based simulations*, given logs from two different real-world experiments and varying parameters for log creation. Hence, comparisons between simulation and real-world results have been performed in a systematic way, using simple tools for automating this task.

9.4 General project compliance

The following are some of the requirements for D-level master's theses at Uppsala University, Sweden. The student should apply his/her knowledge in computer science to a realistic problem. A majority of the work should be done independently. The problem should be dealt with in a scientific way, using scientific methods that the student has learned during his/her education, and the results should be analyzed extensively. The problem should be related to scientific literature, and the student should gain new knowledge during the project. Furthermore, the course curriculum specifies that the main goal is to provide training in planning, performing and accounting of a scientific project.

It is my opinion that this master's thesis fulfils all these requirements. Ad-hoc networks, AODV-UU and the ns-2 network simulator were all completely new to me, and required thorough investigation. This provided me with a large amount of valuable knowledge and insights. During the porting process, I applied my knowledge in programming and software engineering to solve the porting problem and to integrate the ported version of AODV-UU into ns-2. All problem solving was preceeded by careful analysis, and the solutions clearly motivated. Almost all work has been carried out independently. Experiments have been described in such detail that they should be easily repeatable, and all simulation scenario scripts and other additional material mentioned in this master's thesis have been made available on a special webpage for download [55]. Where appropriate, references have been provided to related research literature. Finally, the project has provided me with valuable practical experience in problem solving, scientific work and report writing.

9.5 Future work

Some future work remains to be done on the ported version of AODV-UU. Most notably, the platform portability issue should be solved for AODV-UU to become available for other platforms than the Linux platform. Also, the ns-2 OTcl support code of AODV-UU should be integrated into the latest release of ns-2, version 2.1b9a, which was released during this project.

Finally, the possibility of using additional data from real-world experiments for trace-based simulations should be investigated further. Additional parameters such as packet delay and error rates could help making trace-based simulations more realistic, provided that these parameters can be extracted or derived from logfiles of real-world experiments, and that support for them can be implemented in the simulator. This would not only bring simulations closer to real-world experiments, but also to the techniques of trace-based emulation.

Bibliography

[1] *The Official IETF working group Manet webpage*, `http://www.ietf.org/html.charters/manet-charter.html`

[2] J. BROCH, D. A. MALTZ, D. B. JOHNSON, Y. C. HU AND J. JETCHEVA. *A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols*. Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98), October 1998.

[3] P. JOHANSSON, T. LARSSON, N. HEDMAN, B. MIELCZAREK AND M. DEGERMARK. *Scenario-based Performance Analysis of Routing Protocols for Mobile Ad-hoc Networks*. Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99), pp. 195-206, August 1999.

[4] H. LUNDGREN, D. LUNDBERG, J. NIELSEN, E. NORDSTRÖM AND C. TSCHUDIN. *A Large-scale Testbed for Reproducible Ad hoc Protocol Evaluations*. Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC'02), March 2002.

[5] C.-K. TOH, R. CHEN, M. DELWAR AND D. ALLEN. *Experimenting with an Ad Hoc Wireless Network on Campus: Insights and Experiences*. ACM SIGMETRICS Performance Evaluation Review, Volume 28, Issue 3, pp. 21-29, December 2000.

[6] *The AODV-UU webpage*, `http://www.docs.uu.se/scanet/aodv/`

[7] C. PERKINS, E. ROYER AND S. DAS. *Ad hoc On-Demand Distance Vector (AODV) Routing*, Internet Draft, draft-ietf-manet-aodv-11.txt, *work in progress*, June 2002.

[8] *The Network Simulator - ns-2*, `http://www.isi.edu/nsnam/ns/`

[9] *The APE testbed homepage*, `http://apetestbed.sourceforge.net/`

[10] P. KARN. *MACA - A New Channel Access Method for Packet Radio*. Proceedings of the 9th ARRL Computer Networking Conference, London, Ontario, Canada, September 1990.

[11] M. SANCHEZ. *Transmission Power Control in Ad-hoc Networks*, slides, April 2002.

[12] Y-G. HONG, Y-J. KIM, M-T. WANG. *Autoconfiguration of IPv4 Link-Local Addresses in Multilink Networks*, Internet Draft, draft-hong-zeroconf-multilink-ipv4-00.txt, *work in progress*, November 2001.

[13] C-K TOH. *Ad Hoc Mobile Wireless Networks: Protocols and Systems*, December 2001.

[14] D. C. PLUMMER. *An Ethernet Address Resolution Protocol – or – Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*, Request For Comments (RFC) 826, November 1982.

[15] J. MOY. *OSPF Version 2*, Request For Comments (RFC) 2328, April 1998.

[16] C. HEDRICK. *Routing Information Protocol*, Request For Comments (RFC) 1058, June 1988.

[17] G. MALKIN. *RIP Version 2*, Request For Comments (RFC) 2453, November 1998.

[18] R. DROMS. *Dynamic Host Configuration Protocol*, Request For Comments (RFC) 2131, March 1997.

[19] L. L. PETERSON AND B. S. DAVIE. *Computer Networks: A Systems Approach*, 2nd ed., Morgan Kaufmann Publishers, San Francisco, CA, USA, 2000.

[20] S. CORSON AND J. MACKER. *Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations*, Request For Comments (RFC) 2501, January 1999.

[21] L. BUTTYÁN AND J.-P. HUBAUX. *Report on a Working Session on Security in Wireless Ad Hoc Networks*. ACM Mobile Computing and Communications Review (MC^{2}R), Volume 6, Number 4, October 2002.

[22] C. E. PERKINS AND P. BHAGWAT. *Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers*. Proceedings of the SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications, August 1994.

[23] D. B. JOHNSON, D. A. MALTZ, Y-C HU AND J. G. JETCHEVA. *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)*, Internet Draft, draft-ietf-manet-dsr-07.txt, *work in progesss*, February 2002.

[24] Z. J. HAAS, M. R. PEARLMAN AND P. SAMAR. *The Zone Routing Protocol (ZRP) for Ad Hoc Networks*, Internet Draft, draft-ietf-manet-zone-zrp-02.txt, *work in progesss*, July 2002.

[25] T. CLAUSEN, P. JACQUET, A. LAOUITI, P. MINET, P. MUHLETHALER, A. QAYYUM AND L. VIENNOT. *Optimized Link State Routing Protocol*, Internet Draft, draft-ietf-manet-olsr-06.txt, *work in progress*, September 2001.

[26] M. GERLA, X. HONG, L. MA AND G. PEI. *Landmark Routing Protocol (LANMAR) for Large Scale Ad Hoc Networks*, Internet Draft, draft-ietf-manet-lanmar-04.txt, *work in progress*, June 2002.

[27] M. GERLA, X. HONG AND G. PEI. *Fisheye State Routing Protocol (FSR) for Ad Hoc Networks*, Internet Draft, draft-ietf-manet-fsr-03.txt, *work in progress*, June 2002.

[28] Z. J. HAAS, M. R. PEARLMAN AND P. SAMAR. *The Interzone Routing Protocol (IERP) for Ad Hoc Networks*, Internet Draft, draft-ietf-manet-zone-ierp-02.txt, *work in progress*, July 2002.

[29] Z. J. HAAS, M. R. PEARLMAN AND P. SAMAR. *The Intrazone Routing Protocol (IARP) for Ad Hoc Networks*, Internet Draft, draft-ietf-manet-zone-iarp-02.txt, *work in progress*, July 2002.

[30] R. G. OGIER, F. L. TEMPLIN, B. BELLUR AND M. G. LEWIS. *Topology Broadcast Based on Reverse-Path Forwarding (TBRPF)*, Internet Draft, draft-ietf-manet-tbrpf-05.txt, *work in progress*, March 2002, `http://ietf.org/internet-drafts/draft-ietf-manet-tbrpf-05.txt`

[31] V. PARK AND S. CORSON. *Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification*, Internet Draft, draft-ietf-manet-tora-spec-04.txt, *work in progress*, July 2001.

[32] M. S. CORSON, S. PAPADEMETRIOU, P. PAPADOPOULOS, V. PARK AND A. QAYYUM. *An Internet MANET Encapsulation Protocol (IMEP) Specification*, Internet Draft, draft-ietf-manet-imep-spec-02.txt, *work in progress*, August 1999.

[33] L. M. FEENEY. *A Taxonomy for Routing Protocols in Mobile Ad Hoc Networks*, SICS Technical Report T99/07, October 1999.

[34] IEEE 802 LAN/MAN STANDARDS COMMITTEE. *IEEE 802.11, 1999 Edition: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[35] T. Larsson and N. Hedman. *Routing Protocols in Wireless Ad-hoc Networks - A Simulation Study*, Master's Thesis at Luleå University of Technology, Stockholm, 1998.

[36] *The Netfilter/iptables project*, `http://www.netfilter.org/`

[37] J. Postel. *Internet Control Message Protocol*, Request For Comments (RFC) 729, September 1981.

[38] *Mad-hoc AODV technical documentation*, `http://mad-hoc.flyinglinux.net/techdoc.ps`

[39] E. M. Belding-Royer. *Report on the AODV Interop*, UCSB Tech Report 2002-18, June 2002.

[40] K. Fall and K. Varadhan. *The ns Manual (formerly ns Notes and Documentation)*, April 2002, `http://www.isi.edu/nsnam/ns/ns-documentation.html`

[41] *The VINT (Virtual InterNetwork Testbed) Project homepage*, `http://www.isi.edu/nsnam/vint/index.html`

[42] *Georgia Tech Internetwork Topology Models (GT-ITM) homepage*, `http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html`

[43] *Trace graph homepage*, `http://www.geocities.com/tracegraph/`

[44] The CMU Monarch Project. *The CMU Monarch Project's Wireless and Mobility Extensions to ns*, August 1999, `ftp://ftp.monarch.cs.cmu.edu/pub/monarch/wireless-sim/ns-cmu.ps`

[45] *The CMU Monarch Project homepage*, `http://www.monarch.cs.cmu.edu/`

[46] C. Perkins. *IP Mobility Support*, Request For Comments (RFC) 2002, October 1996.

[47] H. T. Friis. *A note on a simple transmission formula*, Proceedings of the IRE, Vol. 34, pp. 254-256, May 1946.

[48] T. S. Rappaport. *Wireless Communications: Principles and Practice*, Prentice Hall, 1996.

[49] Pacific Missile Test Center - Center/Weapons Division. *EW and Radar Systems Engineering Handbook*, Fundamentals: Decibel (dB), October 1999, `http://ewhdbks.mugu.navy.mil/decibel.pdf`

[50] *The UMD TORA/IMEP website*, `http://www.cshcn.umd.edu/tora.shtml`

[51] D. Kalev. *ANSI/ISO C++ Professional Programmer's Handbook*, Ch. 13: C Language Compatibility Issues, June 1999.

[52] M. Kindahl. *Instant C++ for C programmers*, November 2001, `http://www.iar.com/FTP/pub/press/articles/InstantCplusplusforCprogrammers.pdf`

[53] L. Händel. *The Function Pointer Tutorials*, June 2001, `http://www.function-pointer.org/`

[54] M. Cline. *C++ FAQ Lite*, April 2002, `http://www.parashift.com/c++-faq-lite/`

[55] *Porting AODV-UU Implementation to ns-2 and Enabling Trace-based Simulation (file repository)*, `http://user.it.uu.se/~bjwi7937/kurser/exjobb/files/`

[56] Marc Greis and the VINT Group. *Tutorial for the Network Simulator "ns"*, December 2000, `http://www.isi.edu/nsnam/ns/tutorial/index.html`

[57] *Ns-users Mailing List Info Page*, `http://mailman.isi.edu/mailman/listinfo/ns-users`

[58] H. LUNDGREN, E. NORDSTRÖM AND C. TSCHUDIN. *Coping with Communication Gray Zones in IEEE 802.11b based Ad hoc Networks*. The Fifth International Workshop on Wireless Mobile Multimedia (WoWMoM'02), September 2002.

[59] LUCENT TECHNOLOGIES / AGERE SYSTEMS. *User's Guide for ORiNOCO PC Card*, September 2000, `ftp://ftp.orinocowireless.com/pub/docs/ORINOCO/MANUALS/ug_pc.pdf`

[60] E. NORDSTRÖM. *APE - a Large Scale Ad Hoc Network Testbed for Reproducible Performance Tests*, Master's Thesis at Uppsala University, June 2002.

[61] *INRIA OLSR Implementation,* `http://menetou.inria.fr/olsr/`

[62] *The LUNAR Web Page,* `http://www.docs.uu.se/selnet/lunar/`

[63] G. T. NGUYEN, B. NOBLE, R. H. KATZ AND M. SATYANARAYANAN. *A Trace-based Approach for Modeling Wireless Channel Behavior*, 1996.

[64] M. SATYANARAYANAN AND B. NOBLE. *The Role of Trace Modulation in Building Mobile Computing Systems*. Proceedings of the 6th Workshop on Hot Topics in Operating Systems, Cape Cod, MA, USA, May 1997.

[65] B. D. NOBLE, M. SATYANARAYANAN, G. T. NGUYEN AND R. H. KATZ. *Trace-Based Mobile Network Emulation*. Proceedings of ACM SIGCOMM'97, 1997.

Appendix A

Glossary

A number of technical terms and abbreviations are used throughout this document. The listing below is an attempt to explain some of these.

ad-hoc: *"For the particular end or case at hand without consideration of wider application"* (from *Webster's Dictionary*).

API: *Advanced Programmer's Interface*. A software interface available to programmers, e.g. for adding functionality to existing software.

beacon: A control message issued by a node, often periodically, to inform other nodes of its presence.

beaconing: The usage of *beacons*.

broadcast: Sending of a packet to every host on a particular network.

bi-directional link: A link on which network traffic can flow in both directions.

CBR: *Constant Bit Rate*. A traffic type where the amount of traffic is constant per unit of time.

endian: The characteristics of multi-byte value storage in the memory of a machine.

multicast: Sending of a packet to a specified subgroup of network hosts.

network byte order: A standardized way of storing multi-byte values for transmission over a network.

NPDU: *Network Protocol Data Unit*. The unit of data exchange used at the network layer of a protocol stack, i.e., a packet.

PDA: *Personal Digital Assistant*. An electronic handheld device offering useful functionality for everyday use, such as note-taking.

SNR: *Signal-to-Noise Ratio*. The ratio of signal compared to noise.

TTL: *Time To Live;* the lifetime of a packet. Usually measured in number of hops instead of time.

unicast: Sending of a packet to a single destination host.

uni-directional link: A link on which network traffic only can flow in one direction.