Porting AODV-UU into NS-Miracle

Erik Andersson, Emil Ljungdahl
Karlstad University

Abstract. The NS-Miracle[1] (Multi InteRfAce Cross Layer Extension for NS) framework lacks support for AODV (Ad-hoc On-demand Distance Vector) routing. Research projects at Karlstad University need this feature for simulation of wireless Ad-hoc networks. The Network Simulator (NS-2)[2] has support for AODV routing, but NS-2 does not meet the requirements in simulation with multihomed nodes. AODV-UU[3] is a draft compliant Linux implementation of the AODV protocol from Uppsala University. This implementation also includes a port for NS-2. This project has resulted in a port of AODV-UU for NS-2 into NS-Miracle compatible modules. We have also extended the original code to support multiple interfaces. Different simulation scenarios have been created and analyzed to verify functionality of the port. However, simulation tests that cover more technologies and topologies are still needed. Features such as automatic interface discovery and AP-gateway mode are proposals for future work.

1 INTRODUCTION

In order to establish and maintain connectivity between nodes in an Ad-hoc network, a routing protocol needs to be in place. The routing protocol controls how nodes should route packets in the network. Routing protocols in multihop environments can be divided in three major categories, proactive, reactive and hybrid. One example of a proactive protocol is DSDV (Destination-Sequenced Distance Vector)[4] which determines routes prior any data traffic requests. AODV (Ad-hoc On-demand Distance Vector)[5] is a reactive routing protocol that only maintain routes when needed. A hybrid protocol uses both proactive and reactive techniques.

Our main objective for this project is to port AODV-UU[3] for the Network Simulator (NS-2)[2] into NS-Miracle (Multi InteRfAce Cross Layer Extension for NS)[1] compatible modules. We will also try to extend the original code to better support multihoming. In this report we will give a brief presentation of the AODV protocol and the AODV-UU implementation. We will also cover some basic information about NS-2 and the NS-Miracle framework. Given this background information we will describe the porting process and test results. At the end we will discuss open issues and future work.
2 MOTIVATION

The NS-Miracle framework lacks support for AODV routing. Research projects at Karlstad University need this feature for simulation of wireless Ad-hoc networks. NS-2 has support for AODV routing, but does not meet the requirements in simulation with multihomed nodes.

3 AODV

3.1 DESCRIPTION

AODV (Ad hoc On-demand Distance Vector)[5] is a routing protocol used in ad-hoc networks by mobile nodes. It is a reactive protocol, which means that no routes are discovered until they are needed. This ensures quick adaption to dynamic link conditions and moving nodes. As only active routes are maintained the AODV protocol uses a small amount of resources. To avoid routing loops at all time it uses destination sequence numbers.

3.2 ROUTE DISCOVERY

When data is to be sent to a new node in the mesh, AODV first needs to find a route to this node. AODV broadcasts a RREQ-packet which includes information about the requesting host, the requested host and sequence numbers. To avoid routing loops, a receiving node first checks that it has not received the RREQ before. A reverse route to the requesting node is then set up. If the node does not know about the requested node, it will increment a “hop count field” and re-broadcast the request. If the node already know a route to the requested node (or it is the requested node), it will send a route reply message (RREP-message) as unicast back to the previous node in the request chain.

After a while when intermediate nodes know a route to the destination, there is a great chance that the destination node never receives any RREQ-packets, and thus cannot set up a reverse route, which will cause the destination node to send a RREQ-packet before it is able to send a reply message. To avoid this, there is a gratuitous RREP flag defined in a RREQ-packet. If this flag is set, an intermediate node which replies to a RREQ-message also needs to send a RREP-message with route information about the requesting node to the requested node.

3.3 ROUTE MAINTENANCE

Routes are only kept as long as they are needed. If a route is not used for a certain period of time it is marked as invalid and subsequently removed. To monitor the link availability of routes two techniques are available. The first method is to observing periodically broadcasted beacons, called HELLO-messages. The second method AODV supports is link layer feedback, such as the one defined for IEEE 802.11 [6].
4 THE NETWORK SIMULATOR (NS-2)

This section will give a brief overview of NS-2. Parts that are interesting for the porting work will be described in more detail in section 4.4.

4.1 BACKGROUND

NS-2 is a discrete event simulator that supports simulations of many different network protocols and techniques. Simulations can be conducted over wired LANs and WANs, and in wireless ad-hoc networks. NS development began in 1989 as a variant of the REAL network simulator[7]. In 1995 the project was supported by DARPA, Xerox PARC, UCB, and USC/ISI. Due to its open source nature the project has grown through contributions from network researchers around the world.

4.2 DESIGN

The NS-2 core is written in the object-oriented language C++. It is important that the core is capable of manipulate bytes, packet headers and deal with computation logic in an effective way. A compiling language like C++ meets these requirements. In contrast to fast computation logic, the creation and modifications of simulation scenarios need to be flexible. Given these premises the interpreting object-oriented language OTcl[8] was chosen as a frontend to the core. The class hierarchies created in the two programming languages are closely related to each other. Objects created in the C++ environment can be made available in the OTcl environment and vice versa. This is done by OTcl linkage. The OTcl linkage also enable the OTcl interpreter to be invoked from C++ code.

4.3 SIMULATION

To run a simulation in NS-2 you first need to create a test scenario written in OTcl. The test scenario usually consists of nodes, links and agents. Nodes (fixed or mobile) are connected through links (wired or wireless) and agents (protocols) are attached to the nodes. Scheduling of events instructs the simulator to trigger events at certain points in time during the simulation. One example is that nodeA sends a HTTP GET request to nodeB. The output of the simulation is written to a tracefile. The tracefile could then be analyzed to determine simulation results.

4.4 ENTITIES OF INTEREST

NODES A node is composed of several objects. The default unicast node (see figure 1) contains the following:

- an address
- a list of neighbors
- a node entry
– an address classifier
– a port classifier
– a list of agents
– a node type identifier
– a routing module

Fig. 1. A NS-2 unicast node

The address is monotonically increasing by 1 (starting with zero) for every node that is added to the simulation scenario. NS-2 addresses consist of a node id and a port id. Nodes can be addressed in two different ways, flat (one level node ids) or hierarchical (multilevel node ids).

The node entry points to the first element that will handle arriving packets (the address classifier). The address classifier is responsible to route packets. Routing facilities in the routing module helps the address classifier to determine the correct route. If the incoming packet is destined for the current node, the packet will be forwarded to the port classifier. Otherwise the address classifier tries to route the packet to the correct link. The port classifier determines which agent that should receive the packet.

Prior any node creation the simulator needs to be instructed how to create nodes. This is done through the node configuration interface. If no configuration is given the default one will be used.

AGENTS Agents act as endpoints in the simulation environment. This is where packets are constructed or consumed. Agents are used when implementing various network protocols in NS-2. Examples of agents are Routing agents and traffic
sinks. The agent functionality (class Agent) is written in both OTcl and C++. To be able to assign various fields to a constructed packet, each agent holds internal state information. The C++ class Agent holds the following information:

- addr_ (node address)
- dst_ (where packet should be sent)
- size_ (packet size in bytes)
- type_ (type of packet)
- fid_ (IP flow identifier)
- prio_ (IP priority field)
- flags_ (packet flags)
- defttl_ (default IP ttl value)

In order to make two agents (different endpoints) communicate, they need to be connected to each other. This means that the originating agent sets the destination address and destination port to the corresponding agent.

It is possible to attach applications to agents, but due to limitations in the API this is not a common used feature. The API between agents and applications might be enough for simple services such as FTP and traffic generators. Application functionality is usually implemented as an agent.

**QUEUES** A queue is added to the simulation to enable a location where packets can be held or dropped. The packet scheduling facility decides which packets that should be serviced or dropped. NS-2 supports different types of queuing mechanisms including drop-tail (FIFO) queuing, RED buffer management and variants of Fair Queuing (e.g. Stochastic Fair Queuing). The priority queue (PriQueue) is based on the drop-tail implementation and allows for instance routing traffic to be prioritized.

A queue has two different states, blocked or unblocked. A queue is blocked when dequeued packets are in transit between the queue and its downstream neighbor. The queue will remain blocked as long as it has at least one packet to send and the downstream link is busy. The queue becomes unblocked only when a downstream neighbor invokes the resume function.

5 NS-MIRACLE

This section will give a brief introduction to NS-Miracle (Multi InteRfAce Cross Layer Extension for NS)[1] and explain some of the key concepts of NS-Miracle in more detail.

5.1 INTRODUCTION

NS-Miracle is a set of libraries that enhance functionality provided by NS-2. It is currently developed by the SIGNET Lab at the University of Padova[9].

It provides a modular structure of the network stack in NS-2 and allows users to place multiple modules on each layer in the stack (figure 2). This enables for
instance several network interfaces in a node. NS-Miracle also supports backend functionality for cross-layer messages, where modules in the same, or in different layers, can communicate with each other.

NS-Miracle uses dynamic linkage and can therefore be used without recompiling NS-2 when code in NS-Miracle changes. Up to version 2.32 of NS-2 you need to patch NS-2 to support dynamic linkage, this is however included in version 2.33 and above. Dynamic linkage also enables users to develop their own modules without modifying the NS-Miracle framework. The module compiles to a shared object which can be loaded at runtime.

5.2 MODULE

One of the basic building blocks in NS-Miracle is the Module class. A Module represents an entity that can be placed in the stack. This entity can be anything between a Ping-application and a wireless interface at the physical layer. A Module is placed on a specific layer in the stack which is defined in the OTcl simulation setup.

A special type of module is the ChannelModule which connects two Modules in two different nodes.

5.3 SAP

To connect Modules to each other the SAP interface concept (figure 2) from the OSI model[10] is used. This is implemented through the class SAP. Besides being able to forward data the SAP is also a tracing point where dataflows can be logged. It is of course the perfect point for tracing since all communication in a simulation flow through SAPs.
SAPs are used in three different ways. The first place where SAPs are used is to connect modules in neighboring layers with each other. This is where regular data packets travels up and down the stack. This connectivity is manually defined in the OTcl script to be able to define multiple branches in the stack (e.g. multiple IP interfaces connects to one linklayer each).

The second way a SAP is used is to connect a node to a channel. For this purpose a specific SAP class called ChSAP exits, which connects a Module to a ChannelModule.

The last use of SAPs are to connect every Module to the NodeCore. This connection is handled by the ClSAP subclass and is used to send cross layer messages (see 5.4).

5.4 CROSS LAYER MESSAGES

NS-Miracle supports the ability to communicate between Modules with the help of cross layer messages. Every Module is connected to the NodeCore which can be considered as a cross layer message bus. The NodeCore includes all logic to receive and forward cross layer messages.

A cross layer message is derived from the base class ClMessage. By extending ClMessage, Modules are able to pass any kind of data between them. It is however important to note that Modules need explicit knowledge about a specific ClMessage-type to be able to handle it.

Cross layer messages can be sent in synchronous or asynchronous mode. In synchronous mode the sending Module blocks until a reply has been received. This is achieved by letting the receiving node set the reply in the original ClMessage object. A synchronous ClMessage also allows Modules to share variables as the same ClMessage is known to both Modules.

In asynchronous mode the send method returns immediately at the sending module, and the replying modules need to send a new asynchronous message in return. As the sender do not wait for a reply, asynchronous messages can be sent with a specific delay.

The NodeCore can handle two types of destinations for cross layer messages. UNICAST messages are directed to a specific Module, whereas BROADCAST messages are directed to all Modules, or Modules at a specific layer in the stack.

As a complement to the NodeCore bus, cross layer messages can also be sent through the regular SAPs that connect Modules in a stack. This can be useful for communication related to the stack structure, e.g. a mac-layer controlling operation mode of phy-layer.

5.5 PLUG-IN

The PlugIn is a kind of Module that lives outside the stack connecting directly to the NodeCore. In fact it is the base class of Module and only defines the ClSAP connection to the NodeCore. A PlugIn should include functionality that does not fit in a stack, but rather communicate or coordinate Modules on different
sites in the stack. It can also contain functionality specific for the whole node. Example of PlugIn applications are battery drain models and network utilization monitors.

6 AODV-UU

In this section we give a brief introduction to AODV-UU (Ad hoc On-demand Distance Vector implementation from Uppsala University)[3]. We also give a somewhat more complete introduction to the port of AODV-UU for the network simulator NS-2[2].

6.1 DESCRIPTION

AODV-UU[3] is an implementation of the AODV routing protocol for the Linux operating system. It is developed at Uppsala University, with some funding from Ericsson AB. It runs as a user-space daemon, maintaining the kernel routing table. The latest release of AODV-UU is based on RFC3561[5] and is released under the GNU General Public License (GPL). AODV-UU also supports Internet gateway mode, where a gateway node can forward encapsulated packets towards the Internet.

It supports both version 2.4.x and 2.6.x of the linux kernel, and cross-compiling for at least the ARM platform is possible. As AODV-UU includes a Linux kernel-module the programming language of choice is C.

6.2 NS-2 PORT

DESCRIPTION Since late 2002 the AODV-UU package also includes a port[11] for NS-2[2]. The codebase for each of the two versions of AODV-UU are merged and you can decide which one to build when compiling. In NS-2, AODV-UU appears as an Agent for a mobile node and can be used just like any other routing agent. The port uses as much as possible of the original code to ease further maintainability and to affect the functionality of the Linux-version as little as possible.

LIMITATIONS Some of the features implemented in AODV-UU is not applicable for the NS-2 version. In NS-2 you define addressing information (e.g. IP-address and netmask) per node, which make multi-interface nodes non-trivial. Therefore, support for multiple interfaces is left out in the NS-2 port.

IMPLEMENTATION The implementation of the port has to consider a number of issues to not affect the original code. A first difference is that NS-2 agents are written in the programming language C++ while the original AODV-UU code is written in C. However, C-code can be used in C++ as C is a subset
of C++, but C++-code cannot be used in C. Therefore, modifications in the original codebase still needs to conform to C syntax and semantics.

The biggest change in code for the port is in the way packets are handled. The original AODV-UU uses the built in kernel methods like netfilter/iptables[12] to decide what to do with packets, while the NS-2 port needs to use NS-2 methods for passing packets between agents and protocols. The structure of a packet is also quite different between the two versions. A packet in NS-2 is a collection of structs representing headers of a “real packet”.

Another key issue in the implementation is the way an AODV routing agent / daemon is instantiated. In the original Linux implementation of AODV-UU every node in an ad-hoc network runs a single AODV-UU daemon. The same is valid for the NS-2 version. Every node in the simulation runs an AODV-UU agent, but in this case the whole ad-hoc network is part of the same process on a single computer. This changes the way global variables can be used. As AODV-UU agents on different nodes in the simulation should be completely isolated, no global variables can be used.

The original AODV-UU implementation uses a specific interface as the Internet interface when operating in Internet gateway mode. Since NS-2 does not support multiple interfaces, but can attach to both wireless and wired channels, the NS-2 implementation uses the address classifier (section 4.4) to support Internet gateway mode.

The author of the NS-2 port of AODV-UU, Björn Wiberg[11], has chosen a monolithic class approach. This means that all source code is placed in a single class, which can be used as a stand-alone class without dependencies to other classes. This method was used because of its simplicity, all existing code just needs to be wrapped into a class definition, which is easily done with defines in C/C++. However, this approach may result in a non object-oriented structure with class definition and implementation spread out over many files in the codebase.

INTEGRATION The port introduce a new NS-2 packet type (PT_AODVUU) and a new NS-2 packet header (PacketHeader/AODVUU). This packet type and packet header are used instead of the AODV control packets over UDP that the AODV draft defines[5]. This method has become the de facto standard in NS-2, mainly because of easier handling in the node classifiers.

To make NS-2 aware of this new packet type and packet header, a patch needs to be applied to the source. This includes modification of several files in the NS-2 code. Except for the definitions and tracers some queues also need patches to recognize the new packet type and header. The complete patch also includes the OTcl code needed for instantiation and default values.
7 THE PORTING PROCESS

7.1 INTRODUCTION
This introduction will give a brief summary of our procedure to port the AODV-UU code into a NS-Miracle compatible module.

INITIAL CODE REVIEW We started the porting work by doing source code analysis. This gave us basic understanding and the “big picture” of the AODV-UU code. It also helped us to determine the scope of the project.

API INCOMPATIBILITY In order to get the source code compatible with NS-Miracle, we had to adapt the code to use the NS-Miracle API. We identified and changed the most obvious incompatibilities.

STEP BY STEP To find the more subtle changes that were needed, we followed different execution paths and changed code elements step by step.

ADD FEATURES The original AODV-UU code has support for multiple network interfaces. The NS-2 port of AODV-UU does not support multiple interfaces due to the experimental support for multihoming in NS-2. We have re-enabled support for multiple interfaces.

INTEGRATE To be able to build the project and integrate it with the NS-Miracle framework, we created project specific Makefiles and added support for our project into NS-Miracle specific Makefiles. We then tried to build the project and fix build errors.

VERIFY FUNCTIONALITY To verify functionality of the ported code we wrote test cases. We moved from basic tests to more advanced. If the behavior did not meet our expectations we changed or fixed the concerned code parts.

7.2 CODE CHANGES
In this section we present a list of changes that was needed in the original code to complete the port. The list contains both small changes in the API and big conceptual changes.

AGENT TO MODULE The most basic, and possible the most obvious change is the conversion from a NS-2 agent (section 4.4) to a NS-Miracle module (section 5.2). As the concept of agents does not exist in NS-Miracle we had to convert the AODV-UU routing agent into one of the basic entities that exist in...
NS-Miracle. Those entities are Plugin (section 5.5), Module (section 5.2) and ChannelModule (section 5.2).

The original AODV-UU code (section 6.1) use Linux built in routing capability at the network layer and runs the AODV logic as a separate daemon. Mapping this structure to NS-Miracle would result in AODV-UU as a Plugin communicating with an existing routing module. However, NS-Miracle does not include a routing module with enough capabilities to support this kind of routing. The options left is to extend the existing routing module, or simply make the whole AODV-UU package as a Module. The latter one needs less changes, and since the available time for this project was quite limited, we chose this approach.

PACKET TYPES AND HEADERS In NS-2 packet types and packet headers are statically declared in NS-2 header files. Since NS-Miracle links dynamically, new packet types and headers can also be dynamically loaded and are therefore declared together with the modules using them.

AODV-UU adds one new packet type (PT_AODVUU), and one new packet header (PacketHeader / AODVUU). Those declarations were extracted from the patch supplied in the AODV-UU package and added to the source code of the new AODV-UU module.

PACKET HANDLING In NS-2 every object has one or more pointers pointing to targets up and down the stack (e.g. where to send packets). In NS-Miracle this is abstracted with the SAPs (section 5.3) and the methods void sendUp(Packet* p) and void sendDown(Packet* p). We had to change every occurrence of target_¬>recv(packet) with the corresponding NS-Miracle method.

The recv() method is similar in NS-2 and NS-Miracle although there is some difference in the method signature. NS-Miracle does not make use of the function callback pointer specified as the second argument to recv() in NS-2. The default recv() method (the one which needs to be re-implemented in every sub-class of Module) in NS-Miracle is therefore void recv(Packet*) as opposite to void recv(Packet*, Handler*) in NS-2.

ADD MULTIPLE INTERFACE In the NS-2 port of AODV-UU (section 6.2) the support for multiple interfaces was excluded because it was not directly applicable in NS-2. In NS-Miracle the support for multiple interfaces is better, and therefore it is desirable to re-implement the multiple interface feature in the port of AODV-UU for NS-Miracle.

To be able to determine which IP-interfaces that shall be part of the AODV routing, new functionality is needed. This can be solved using cross layer messages (section 5.4), where AODV-UU poll its underlying IP-interface modules, or by explicit define participating interfaces in the simulation configuration (OTcl script). We chose the latter approach since no cross layer messages are defined for
the existing IP-interface class and because of difficulties to differentiate between
Internet gateway interfaces and regular interfaces. In code this means that we
added a new TCL command `add-if`.

AODV-UU also modifies the interface queue which is placed on top of a MAC-
module in the stack. AODV-UU can move packets from one queue to another if
a specific route change interface, or remove packets if a route invalidates. Since
no queue-module is available for NS-Miracle the queues in NS-2 are used for
NS-Miracle as well. This left us with no other possibility than to explicit add
knowledge about queues from the simulation configuration, which is implemented
with the command `if-queue`.

**MODIFY INTERNET GATEWAY** The implementation of *Internet gateway
mode* in AODV-UU for NS-2 relies on the address classifier (section 4.4) to
pass packets to an external wired interface. In NS-Miracle this can be much more
flexible since it is possible to use multiple interfaces. By defining one underly-
ing IP-interface as the gateway interface this is easy accomplished. We added a
new TCL command, `add-gw-if`, that adds an Internet gateway interface, just
as `add-if` did with a regular ad hoc-interface.

Except for the above conceptual changes we also had to add code that set
the *source address* and *ttl* on data-packets that is to be encapsulated (section
6.1). In the NS-2 version of AODV-UU this is done in `Agent`-specific code.

**8 TESTING**

To be able to verify the functionality of the ported code, we have written differ-
ent test scenarios and performed simulations. The more advanced test scenario
consists of several wireless nodes (single and multihomed) and one gateway node.
This section will describe the mentioned test scenario in detail.

**8.1 INTRODUCTION**

The test scenario simulates a wireless ad-hoc network (figure 3) with 802.11
nodes and one wired external node. All of the wireless nodes have AODV routing
capabilities. For details, please refer to the Tcl script `aodv-uu_mesh_plus_wired.tcl`
which can be found in the directory `samples` in the source code. The simulation
environment includes the following entities:

- A shared backbone network
- Three access networks (CH 1, CH 11 and CH 6)
- Three multihomed nodes (B_1, B_2 and B_3) that connect the access and
  backbone networks
- Four singlehomed (1_1, 1_2, 11_1, 6_1) nodes connected to different access
  networks
- One external node (node B_3 is configured as an Internet gateway)
8.2 SCENARIO

Given the previously described simulation setup, this scenario simulates a ping request from node 11_1 to node 6_1. Node 6_1 answers with a ping reply. No routes are established prior to the ping request. The following sequence diagram (figure 4) shows the AODV control traffic that are needed to establish the required routes. The ping request that triggers the route discovery is also shown in the diagram.

8.3 SOME NOTES

The sequence diagram shown in (figure 4) is created from AODV-UU debug information and the simulation trace file. We feel that the AODV routing traffic that was generated during the simulation follows the expected behavior. We have also done tests where node 1_1 ping the external node (and vice versa).

9 OPEN ISSUES

Although the porting process was overall successful, there is still some issues that need to be solved before the AODV-UU module for NS-Miracle “just works”. In this section those issues are listed and described. In some cases a proposed solution is also provided.
9.1 INTERFACE QUEUE NOT RECOGNIZING AODV-UU PACKETS

Since NS-Miracle does not include any interface queue modules, the interface queues from NS-2 are still used. At least one of the queues in NS-2 (Queue/DropTail/PriQueue) are able to prioritize routing specific packets. The packet types recognized as routing packets by the queue are hard-coded in the source code. AODV-UU for NS-2 comes with a patch that add the packet type PT_AODVUU to the source code.

Since NS-2 should be totally unaware of NS-Miracle modules it is not an option to distribute this kind of patch together with the AODV-UU module for NS-Miracle. One option is to write new interface queue modules for NS-Miracle, which however force changes in every NS-Miracle module which use the old NS-2 queues.

9.2 ONLY TESTED FOR 802.11

The new module is only tested together with the 802.11 modules in NS-Miracle. This is mainly because of the limited time frame of this project. There might be reasonable or unreasonable function assumptions for modules in lower layers that is not fulfilled. One example of such an assumption is described in section 9.3.

If unreasonable assumptions are made, the code for AODV-UU should be changed.
9.3 DETECTING ROUTING LOOPS
AODV-UU relies on a hop-counter ($\text{num\_forwards()}$) in the common packet header to detect routing loops. This counter should be increased for every node a packet pass through. The 802.11 MAC module is the only module in NS-Miracle that increments $\text{num\_forwards()}$, and thus the only MAC-module that works correctly with AODV-UU.

The easiest, and probably the most correct solution to this problem is to let AODV-UU keep track of the hop-count independently of other modules, or by detecting routing loops in another way.

10 FUTURE WORK
In this section we propose some new features that we think should be implemented in further versions of AODV-UU for NS-Miracle.

10.1 EXTENDED TRACING
The tracing functionality for the ported AODV-UU module is quite limited. This functionality can be extended to supply the user with more useful trace logs. TCL commands for printing routing-tables etc. should also be implemented.

10.2 ADD AP-MODE
In a real world scenario an AODV-router might work in mixed mode, letting mobile clients without AODV capabilities connect in regular infrastructure mode. This can probably be implemented in AODV-UU by generalize the Internet gateway function to act as a gateway for a list of hosts (the AP clients).

10.3 AUTOMATIC IP-INTERFACE DETECTION
As stated in ADD MULTIPLE INTERFACE in section 7.2 underlying interfaces can be detected automatically via cross layer messages. This assumes however that the IP-interface module is able to handle such cross layer messages, which is not the case at the moment. Another problem to solve is how to distinguish between regular interfaces and an Internet gateway interface.

11 SUMMARY
This project has resulted in an AODV-UU module for NS-Miracle. We feel that the port was successful, although more tests are needed to verify functionality in different scenarios. Compared to the NS-2 version of AODV-UU we have re-enabled multiple interface support. We have also identified some features that are subject for further implementation.

The project was finished within the given time frame. We hope that our work can be useful in future research projects at Karlstad University.
References