Performance Improvements to the AODV Routing Protocol and Multiple Hop Wireless Routes

A thesis presented in partial fulfilment of the requirements for the degree of

Masters of Engineering

in

Computer Systems Engineering

at Massey University, Palmerston North,

New Zealand.

Matthew Kersley Sinclair

2005

Abstract

This research focused on improving the performance of the Ad-hoc On-demand Distance Vector (AODV) routing protocol over multiple hop routes. The two specific areas that this research addressed were the dramatic decrease in throughput over multiple hop IEEE 802.11 wireless routes and the problems caused by the use of hello messages by AODV implementations to detect broken routes. To help ensure that this research was suitable for real world scenarios, only off-the-shelf software and hardware was used for both the implementations and the tests.

This thesis firstly presents an overview of IEEE 802.11 based wireless networking and the AODV protocol, along with wireless networking and networking in general within the Linux operating system. The thesis then presents the problems caused by hello messages and shows how the IEEE 802.11 wireless standard contributes to the dramatic decrease in throughput over multiple hop routes.

To overcome the hello message problems, an AODV implementation was developed which used existing mechanisms on the data link layer, specifically the transmit retry limit, rather then hello messages to detect broken links. To address the multiple hop route throughput problem, the use of two and four IEEE 802.11 based wireless network interfaces per node were investigated, rather than using just a single wireless interface per node. These proposed solutions, and the AODV implementation that was developed as part of this research, were then tested in the areas of functionality and throughput performance improvements.

The thesis concludes by presenting the performance improvements resulting from using multiple interfaces per node and the non hello message based AODV implementation along with outlining possible future research in this area.

Acknowledgments

I want to thank firstly my supervisors Amal Punchihewa, Liyanage De Silva and Firas Al-Ali. Your direction, encouragement and knowledge throughout my masters and honors research has been greatly appreciated.

I want to also thank my friends, family and especially my wife for always backing me and encouraging me even though I took much longer to complete this research then initially expected. Thank you so much.

Lastly I want to thank fellow post graduate research student and friend Michael Morrison. Thank you so much for all the ideas and knowledge you provided during this research, for your sense of humor and for always being a great friend.

Table of Contents

Abstract i

Acknowledgements ii

Table of Contents iii

List of Figures vi

Table of Tables ix

1 Introduction 1

2 Wireless Networking Overview 3

2.1 Wireless Network Types 3

2.2 Wireless Networking in the Linux 2.4 Kernel 6

2.2.1 General Network Operation 6

2.2.2 Wireless Network Operation 9

2.3 The Ad-Hoc On-demand Distance Vector (AODV) Routing Protocol 10

3 The Hello Message Problem 15

3.1 Gray Zone Problems 16

3.2 Transmission Rate and Packet Size Problems 18

3.3 Overhead and Power Related Problems 18

3.4 Past Research about Data Link Layer Notifications and AODV 18

4 Data Link Layer Feedback 20
4.1 Logical Link Control Sublayer 21
4.2 Medium Access Control Sublayer 22
4.3 Using the Transmit Retry Limit 24
4.4 Comparing Hello Messages and Data Link Layer Feedback as Error Detection Techniques ..25

5 The Multiple Hop Route Throughput Problem 28
5.1 The Access Method for the IEEE 802.11 IBSS Wireless Network 28
5.1.1 The CSMA/CA Algorithm over a Multiple Hop Route 30
5.2 Using Multiple IEEE 802.11 Wireless Interfaces Per Node 35
5.2.1 Areas of the CSMA/CA Algorithm Improved by Using Multiple Wireless Interfaces per Node 35
5.2.2 Multiple Wireless Interfaces and the Transport Control Protocol 39

6 Implementation 43
6.1 Basic Operation of the AODV-HM Implementation 44
6.2 Changing the IEEE 802.11 Device Driver 46
6.2.1 Changing the ADM8211 Chipset Driver to use the Transmit Retry Limit 47
6.3 Changing the AODV-HM Implementation to use Data Link Layer Feedback 49
6.3.1 Detection of a Broken Route 50
6.3.2 Management of the Neighboring Node Table 52
6.4 Making the AODV-LL Implementation Compatible with Multiple Wireless Interfaces 54
6.5 Making the AODV-LL Implementation Compatible with the AODV-HM Implementation 55

7 Testing Methodology 59
7.1 Functionality Test Methodology 64
7.1.1 Optimal Transmit Retry Limit Test 64
7.1.2 Operational Tests 65
7.1.3 Compatibility Tests 66
7.1.4 Multiple Interfaces Tests 66
7.2 Performance Test Methodology 67
7.3 Testing Using Actual Hardware and Software 69

8 Results And Discussion 71
8.1 Functionality Results 71
8.1.1 Optimal Transmit Retry Limit 71
8.1.2 Operation 72
8.1.3 Compatibility 82
8.1.4 Multiple Interfaces 86
8.2 Performance Results 90

9 Conclusion 96

10 Future Research 99

11 References 100

12 Appendices 102

List of Figures

Figure 1.1 Research structure 2
Figure 2.1 Differences between the IEEE 802.11 network configurations 5
Figure 2.2 An infrastructure, standard ad-hoc and ad-hoc with routing capable nodes type wireless networks before and after node movement around an obstacle 6
Figure 2.3 Basic Linux Networking Stack for the Kernel Version 2.4 7
Figure 2.4 Linux Network Stack IP Handler with Netfilters 8
Figure 2.5 Difference between the IEEE 802.11, IEEE 802.3 and Ethernet II Frame Headers 10
Figure 2.6 The route request process 13
Figure 2.7 The route reply process 13
Figure 2.8 The route reply process 14
Figure 2.9 Finding an alternate route after receiving a RERR 14
Figure 4.1 IEEE LAN/MAN Standards within the Physical and Data Link layers - numbers are the IEEE standard number 20
Figure 4.2 Difference between the information implied by not receiving a MAC sublayer ACK and the transmit retry limit being exceeded 25
Figure 5.1 The basic IEEE 802.11 MAC sublayer CSMA/CA operation 29
Figure 5.2 Example setup using a single IEEE 802.11 per node 32
Figure 5.3 Operations taking place in a single data transaction over a two hop wireless route using a single IEEE 802.11b device per node 33
Figure 5.4 Duration of a single TCP data transaction over a range of multiple hop routes 34
Figure 5.5 Throughput rate of a TCP data transaction over a range of multiple hop routes 34
Figure 5.6 Example setup using two IEEE 802.11 interfaces per node 36
Figure 5.7 Operations taking place in a single data transaction over a two hop wireless route using two IEEE 802.11 interfaces per node 36
Figure 5.8 Example setup using four IEEE 802.11 interfaces per node 37

Figure 5.9 Operations taking place in a single data transaction over a two hop wireless route using four IEEE 802.11 interfaces per node 37
Figure 5.10 Duration of a single TCP data transaction for a range of multiple hop routes and IEEE 802.11 interfaces per node..... 38
Figure 5.11 Throughput rate of a TCP data transaction for a range of multiple hop routes and IEEE 802.11 interfaces per node..... 39
Figure 6.1 The simplified command and data flow of the original hello message based AODV implementation – Kernel AODV v2.2.2..... 46
Figure 6.2 The process from the transmit retry limit being exceeded to AODV being notified..... 49
Figure 6.3 Simplified data and command flow through the AODV-HM implementation and the AODV-LL implementation as the result of detecting a broken route..... 52
Figure 6.4 The simplified data and command flow of the AODV-LL implementation..... 54
Figure 6.5 The different AODV-HM and AODV-LL RREP structures..... 57
Figure 7.1 IIST Networking Laboratory Massey University..... 60
Figure 7.2 Micronet SP906B PCI wireless interface with the ADM8211 chipset..... 62
Figure 7.3 Micronet SP906BB wireless interface with the RTL8180L chipset..... 63
Figure 7.4 Different views of the four wireless interfaces installed in the computer showing also the antennas used. The top three interfaces are the ADM8211 chipset interfaces while the bottom interface is the RTL8180L chipset interface..... 68
Figure 8.1 Number of Times Retry Limit was Exceeded vs Retry Limit Size..... 71
Figure 8.2 Messages displayed by the kernel on the source node in first operational test..... 73
Figure 8.3 AODV and ICMP network traffic on the intermediate node during the first operational test.. 73
Figure 8.4 Kernel messages on the intermediate node during the first operational test..... 74
Figure 8.5 Kernel messages displayed on the destination node during the first operational test..... 75
Figure 8.6 Kernel messages displayed on the source node during the second operational test..... 75
Figure 8.7 ICMP request results during the second operational test 76
Figure 8.8 Kernel messages displayed on the intermediate during the second operational test..... 77
Figure 8.9 Second operational test, intermediate node: detecting the broken route..... 78
Figure 8.10 Second operational test, intermediate node: restoring the route..... 79
Figure 8.11 Kernel messages displayed on the destination node during the second operational test....80
Figure 8.12 Third operational test, source node: finding the route..... 80
Figure 8.13 Third operational test, source node: detecting the broken route..... 81
Figure 8.14 Third operational test, source node: restoring the route..... 81

Figure 8.15 First compatibility test, source node: finding the route..... 82
Figure 8.16 First compatibility test, intermediate node: setting up the route..... 83
Figure 8.17 First compatibility test, destination node: setting up the route..... 84
Figure 8.18 Second compatibility test, source node: detecting and restoring the broken route.....85
Figure 8.19 Second compatibility test: destination node, detecting and restoring the broken route..... 86
Figure 8.20 Kernel route table on the intermediate node during the two interface per node test.....87
Figure 8.21 AODV-LL using the two wireless interfaces on the intermediate node during the first , multiple interface test..... 87
Figure 8.22 AODV-LL using four interfaces on the intermediate node.....88
Figure 8.23 Traffic on the four interfaces on the intermediate node during an ICMP packet transfer 89
Figure 8.24 Throughput over multiple hop routes for different wireless network interface configurations and AODV implementation types..... 95

List of Tables

Table 2.1 Differences Between On-Demand and Table Based Routing Protocols..... 12
Table 4.1 Advantages and Disadvantages of Beacons and MAC Acknowledgments..... 23
Table 5.1 Execution times for the different operations in a simple wireless transaction..... 32
Table 5.2 Duration of a single TCP data transaction and the throughput rate for a range of multiple hop routes..... 33
Table 5.3 Duration of a single TCP data transaction and the throughput rate for a range of multiple hop routes and interfaces per node. 38
Table 6.1 Significant areas of the original AODV-HM implementation that were changed or created to create the AODV-LL implementation 50
Table 8.1 Performance test results using AODV-HM with one wireless network interface 91
Table 8.2 Performance test results using AODV-LL with one wireless network interface 91
Table 8.3 Performance test results using AODV-LL with two wireless network interfaces 92
Table 8.4 Performance test results using AODV-HM with two wireless network interfaces 92
Table 8.5 Performance test results using AODV-HM with four wireless network interfaces 94
Table 8.6 Performance test results using AODV-LL with four wireless network interfaces 94

1 Introduction

Ad-hoc wireless networking is an exciting technology with huge potential. It allows networks to form without the need for any fixed infrastructure already in place, permitting clients to be completely mobile while remaining connected, and allowing networks to form in locations and over areas not easily possible for wired networks. Previous research by the author [1] looked at the Ad-hoc On-demand Distance Vector (AODV) routing protocol, a routing protocol commonly used worldwide to provide multiple hop routing capabilities to ad-hoc wireless networks. From this research two areas were identified as needing improvement to make AODV a more efficient and effective routing protocol. The two problem areas were:

1. The use of hello messages in AODV implementations resulting in increased network interference, decreased throughput and the possible creation of unusable routes
2. The rapid throughput decrease per hop over multiple hop routes

The focus of this research was to find solutions to these two problems; specifically to find an alternative route error detection technique to hello messages and improve throughput over multiple hop routes. The general research structure is shown in *Figure 1.1*.

Firstly, this thesis gives a brief introduction on wireless networking with more detail on the AODV routing protocol and networking within the Linux kernel. The thesis then focuses on finding an alternative to hello messages and the throughput decrease over multiple hop routes, proposing solutions to both problems. The solutions were: using the transmit retry limit, a feature of IEEE 802.11 based wireless networking, instead of hello messages, to detect broken routes and using multiple wireless interfaces per node to address the problem of throughput decrease over multiple hop routes. The thesis then covers the implementation stage of the research, which involved creating an AODV

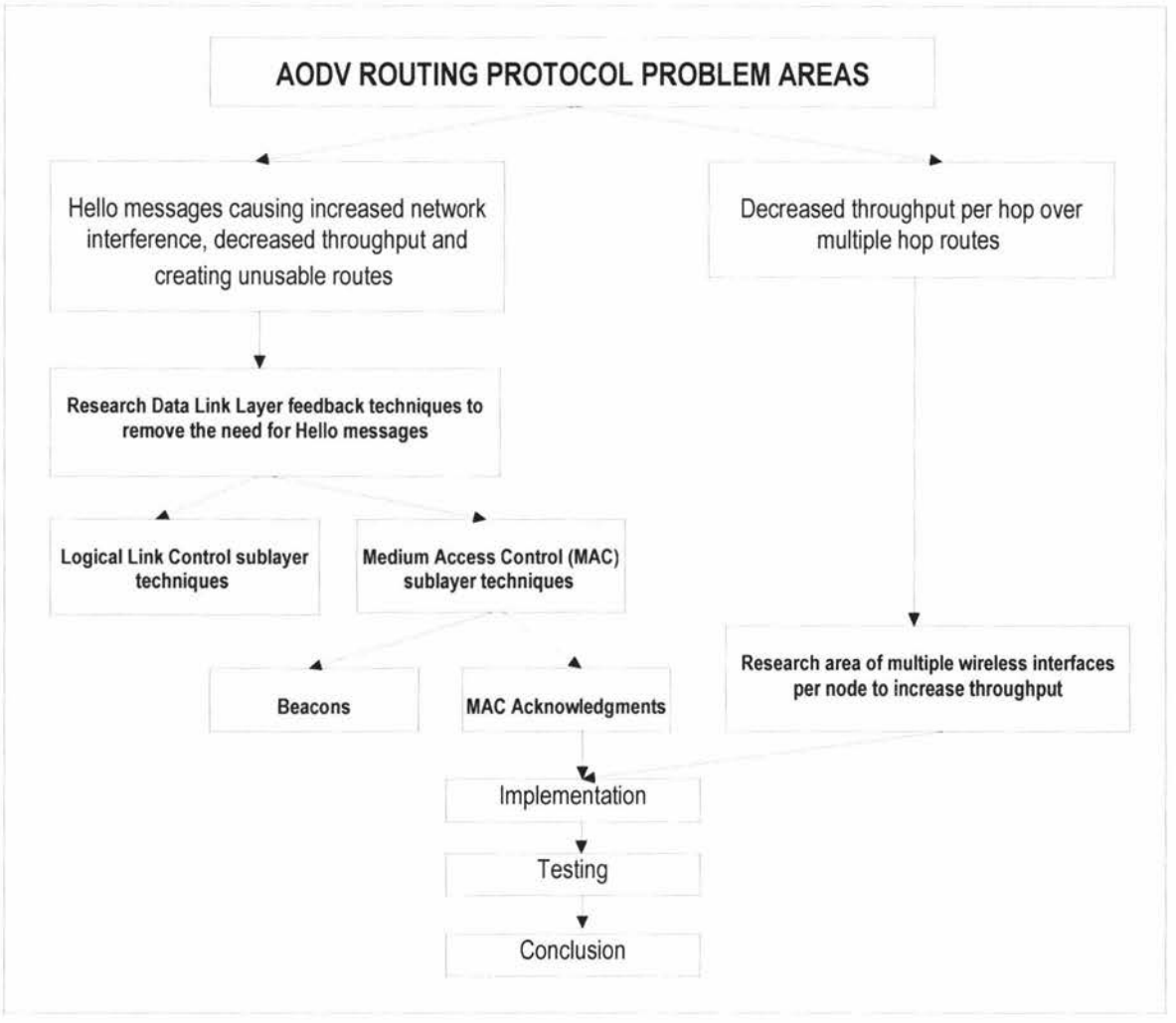


Figure 1.1 Research structure

implementation suitable for testing which uses feedback from the transmit retry limit on the data link layer to detect broken routes and is also capable of working with multiple wireless interfaces. Next the testing methodology is covered, followed by the results and finally conclusions are drawn.

2 Wireless Networking Overview

Wireless networking is about giving devices, such as computers, the ability to communicate with each other by using radio frequencies as the communication medium rather then wires. Wireless networking has meant devices don't need to be fixed by the networking medium and can now be mobile while still remaining part of network. Wireless networks provide a type of network that is less restricted by the surrounding terrain, resulting in the ability to form over areas and in locations not readily possible by fixed networks. It also provides a network which users can join and leave with ease and can also be easily installed and then removed as the networking medium is already present and the networking device already connected wherever the user goes.

Wireless networks also have disadvantages over fixed networks. Wireless networks are affected by random radio activity in the surrounding area, which means the behavior of wireless networks is harder to predict and model than the behavior of fixed networks. Furthermore, due to the nature of wireless networks, and the fact that users can easily connect and leave, wireless networks can be more prone to security issues than fixed networks.

2.1 Wireless Network Types

The types of wireless networks that currently exist can be grouped by the scale of the network that they offer. Large scale wireless networks include satellite and cellular telephone networks and cover global sized areas. Wireless Metropolitan Area Networks (WMANs) range from 5 to 90km and are commonly used for applications such as cable television. Bluetooth is a common WPAN (Wireless Personal Area

Networks) which cover the much smaller area of 1 to 10m. Between the WMAN and WPAN is the WLAN or Wireless Local Area Network. WLAN's generally have a maximum range of about 500m but this is purely dependent on the wireless networking equipment used. Currently the two major WLAN technologies is the High Performance Radio Local Area Network (HIPERLAN) protocol and the IEEE 802.11 protocol. HIPERLAN is most common WLAN standard in Europe, whereas IEEE 802.11 is used globally, making it the most popular wireless networking protocol of the two.

Two main network configurations exist within an IEEE 802.11 WLAN. Firstly there is the Basic Service Set (BSS) configuration, also known as Infrastructure mode. This configuration requires a central management node, usually known as the Access Point (AP), which controls communication between all the other client nodes in the network. For this to be possible, all wireless traffic from one client node to another is routed through the AP. APs can link up with other APs, creating an Extended Basic Service Set (EBSS), allowing the client nodes to be able to roam from the supervision of one AP to another, or likewise, from one BSS to another BSS. Generally APs are dedicated hardware devices whereas a wireless client node would usually be a wireless device installed in a computer. APs are also often fixed in place as they can be connected to a wired LAN. Within a BSS or EBSS, the coverage area of the wireless network equals the coverage provided by all the APs in the network.

The second main network configuration within an IEEE 802.11 WLAN is the Independent Basic Service Set (IBSS). This configuration doesn't require a central management node, instead, the wireless nodes themselves takes turns providing synchronization for the network, and nodes can connect directly to each other rather than through a central management node. This type of network configuration is also known as Ad-hoc mode, as it is the most common configuration used to form an ad-hoc wireless network. *Figure 2.1* shows the differences between the IEEE 802.11 network types.

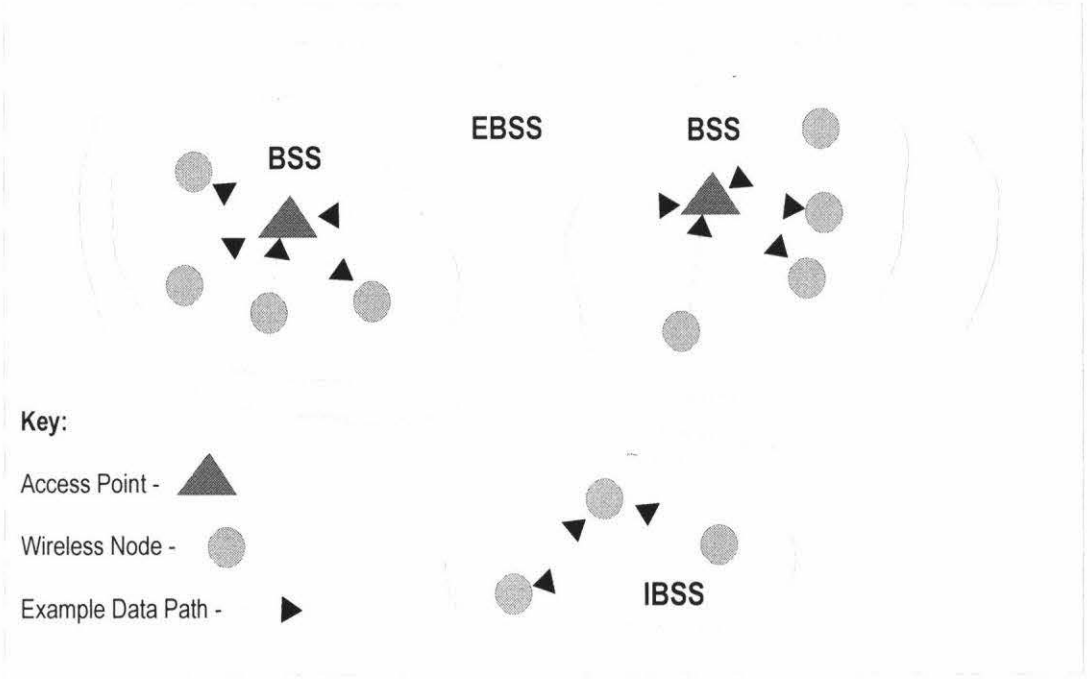


Figure 2.1 Differences between the IEEE 802.11 network configurations

The IBSS configuration, or ad-hoc mode, is the type of WLAN that this research focuses on. The Ad-hoc WLAN has more potential of being able to be a fully mobile network, and therefore; a more flexible type of wireless network, as it not restricted by the fixed nature of a central management node, as is the case with the Infrastructure mode (BSS) WLAN. If wireless nodes are added some extra routing capabilities, especially the ability to form multihop routes to destination nodes by going through intermediate nodes, Ad-hoc WLANs have even more potential to be able to cover large areas and have more flexibility when finding a route to a destination node in comparison to an Infrastructure mode WLAN. Having increased flexibility when finding a route to a destination node means physical obstacles that inhibit or block radio signals, can be overcome easily by using a route that goes around it. This is shown in *Figure 2.2*. For the Infrastructure type WLAN, the area of the network relates to the number of APs in the network, but for an Ad-hoc type WLAN, with multihop routing capabilities, the potential area covered by the network grows as more nodes join the network.

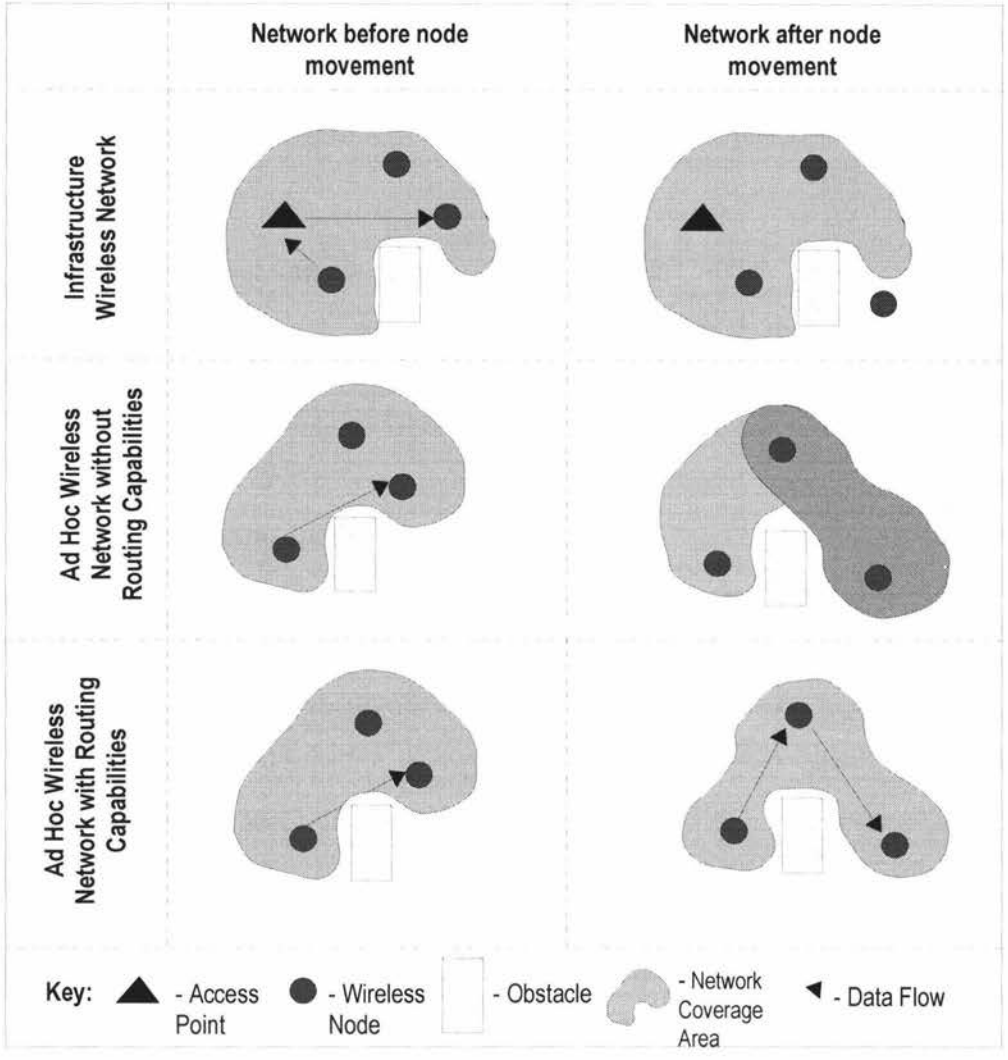


Figure 2.2 An infrastructure, standard ad-hoc and ad-hoc with routing capable nodes type wireless networks before and after node movement around an obstacle

2.2 Wireless Networking in the Linux 2.4 Kernel

Due to its open source nature, the Linux operating system with the kernel version 2.4 was chosen as the operating system for all testing and software implementations that were made as result of this research. The Linux 2.4 kernel was researched to see how it handled wireless networking and networking in general. The reason for choosing Linux was due to it's open source nature allowing easy modification of the kernel.

2.2.1 General Network Operation

The Linux kernel contains a network stack which is a series of stages that the network data passes

through as it moves up from the networking device to the software application or vice versa. At each stage the network data is processed for a different reason and the outcome determines the next stage the network data will move to. In this research, a Linux networking feature called Netfilters, was also compiled into the kernel. Netfilters adds several extra stages within the IP Handler section of the network stack at specific points, which allows applications outside the kernel to view and control the

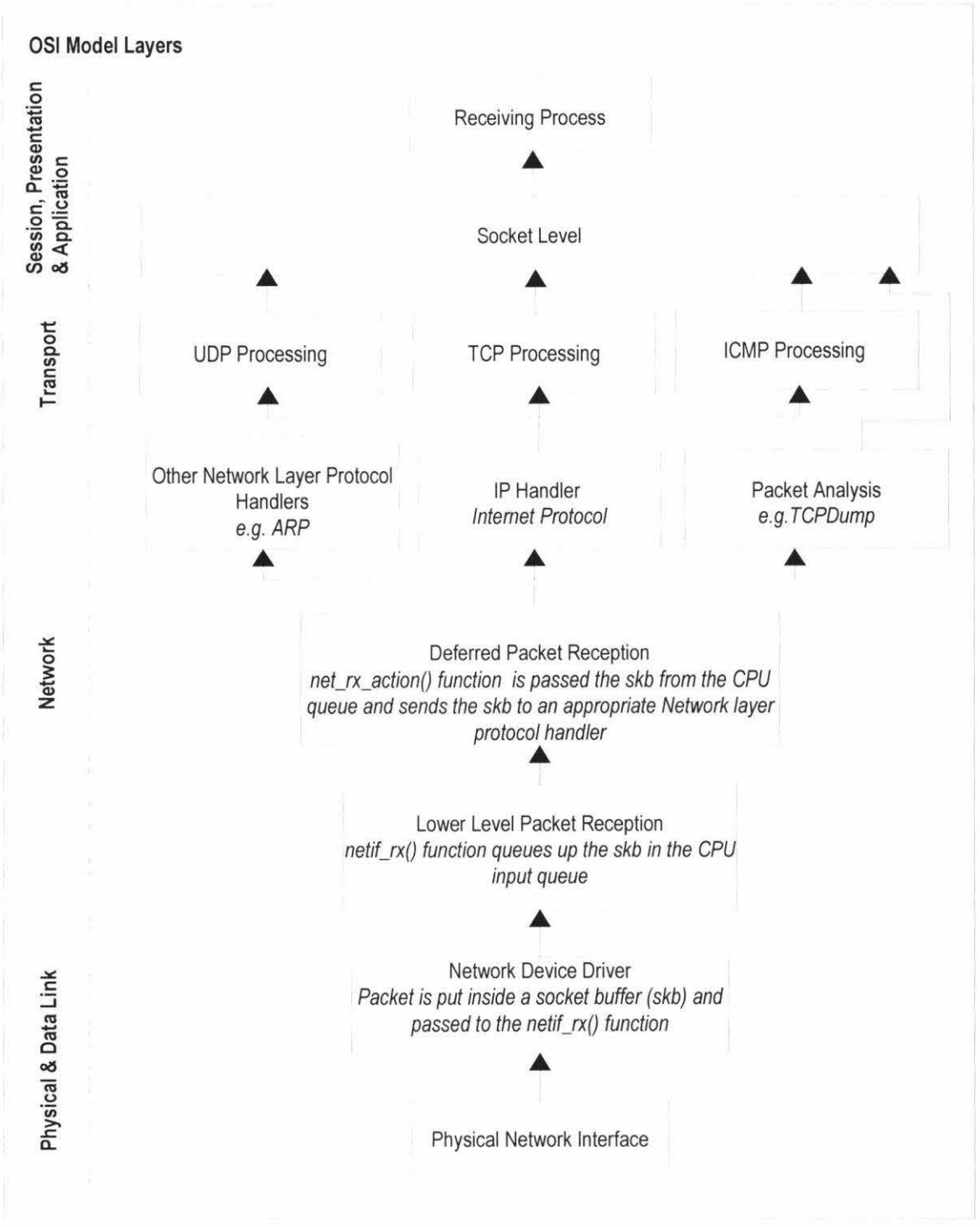


Figure 2.3 Basic Linux Networking Stack for the Kernel Version 2.4

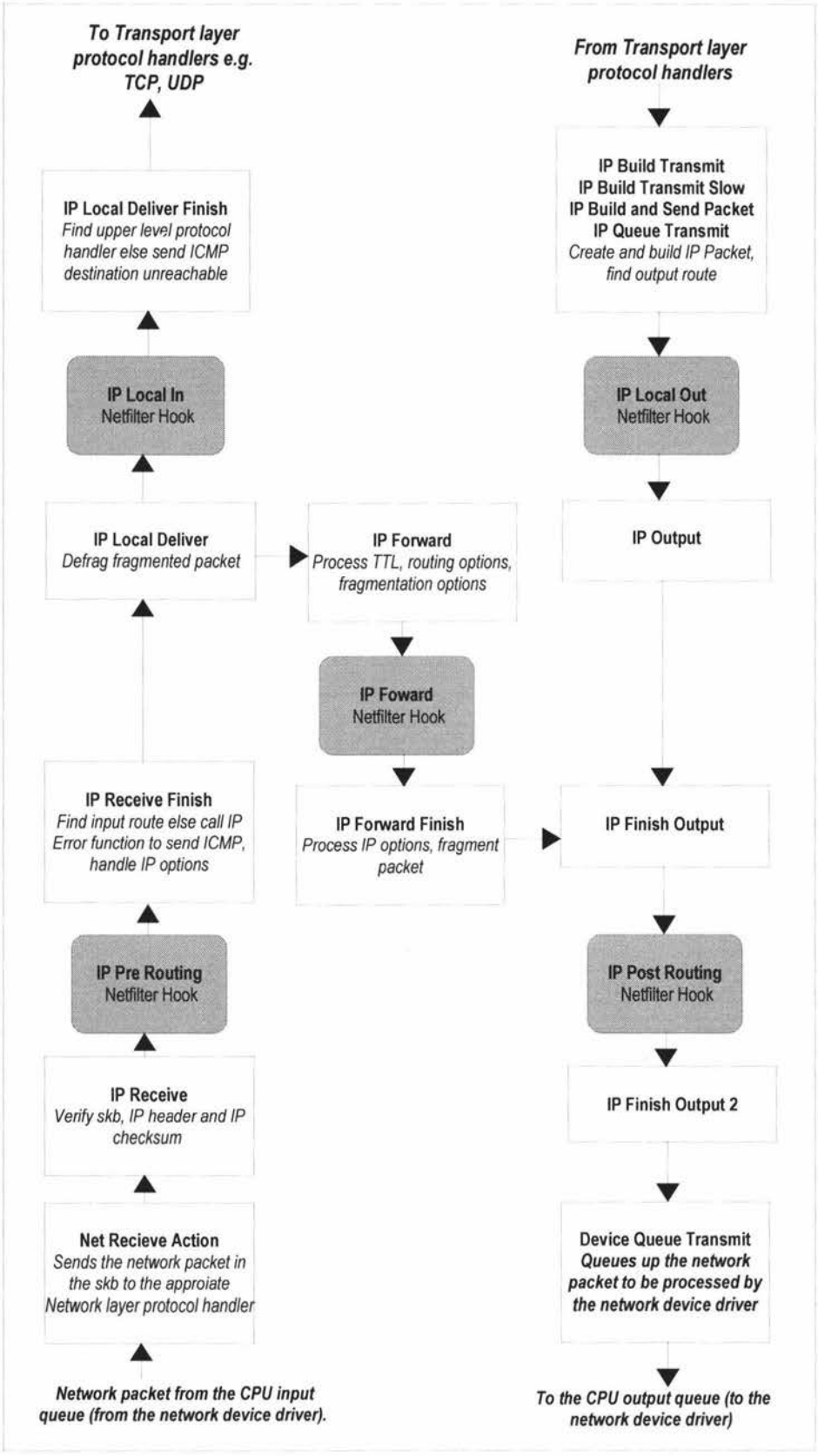


Figure 2.4 Linux Network Stack IP Handler with Netfilters

network data as it passes through the network stack.

Figure 2.3 shows the basic Linux network stack with the data flow going from the networking device to the software application. When the network device driver receives a complete network packet from the network device, the data is placed in memory along with an associated socket buffer structure (SKB). The SKB is a structure which contains pointers to the relevant sections of the network packet in memory, such as the MAC, IP, and UDP or TCP headers. Pointers to the SKBs in memory are kept within a list and processes within the network stack work through the SKB list, using the pointers within the list to access the actual SKBs in memory. This way, only pointers are passed around the network stack allowing the actual SKBs and network data to stay in the same location in memory throughout the whole process, improving memory efficiency and usage. *Figure 2.4* shows in more details the functions within the IP Handler section of the network stack, including the Netfilter hook functions. When a SKB reaches a Netfilter hook, it is passed to any programs registered with the hook. The program that the SKB is passed to can then decided to either keep the SKB, process the SKB and then pass it back to the hook so that is can continue through the network stack, or else simply pass the SKB straight back to the hook without processing it at all.

2.2.2 Wireless Network Operation

While the SKB structure supports IP headers, TCP headers and UDP headers, the only MAC header currently supported (as of the Linux kernel version 2.4) is the DIX (Digital, Intel and Xerox) Ethernet type MAC header which is also known as Ethernet II. IEEE 802.11 wireless network data uses an IEEE 802.11 MAC header which is significantly different to the DIX Ethernet header. *Figure 2.5* shows the difference between the IEEE 802.11, IEEE 802.3 Ethernet and DIX Ethernet headers. Both the IEEE 802.11 and IEEE 802.3 MAC headers require the IEEE 802.2 LLC header to operate, while the DIX Ethernet header does not. To make wireless networking in Linux possible, the IEEE 802.11 wireless network device driver converts the IEEE 802.11 packets received from the wireless device to a DIX Ethernet packet before encapsulating them within SKB structures. To do this conversion, the driver simply removes the IEEE 802.11 MAC header and replaces it with a DIX Ethernet header where all the relevant sections in the DIX Ethernet header are filled in by using information from the IEEE 802.11 MAC header. Because of this, the Linux network stack sees and treats a wireless network as an Ethernet network. The wireless device driver is responsible for converting the DIX Ethernet network packets sent to it by the network stack, to IEEE 802.11 packets, before sending them to the wireless

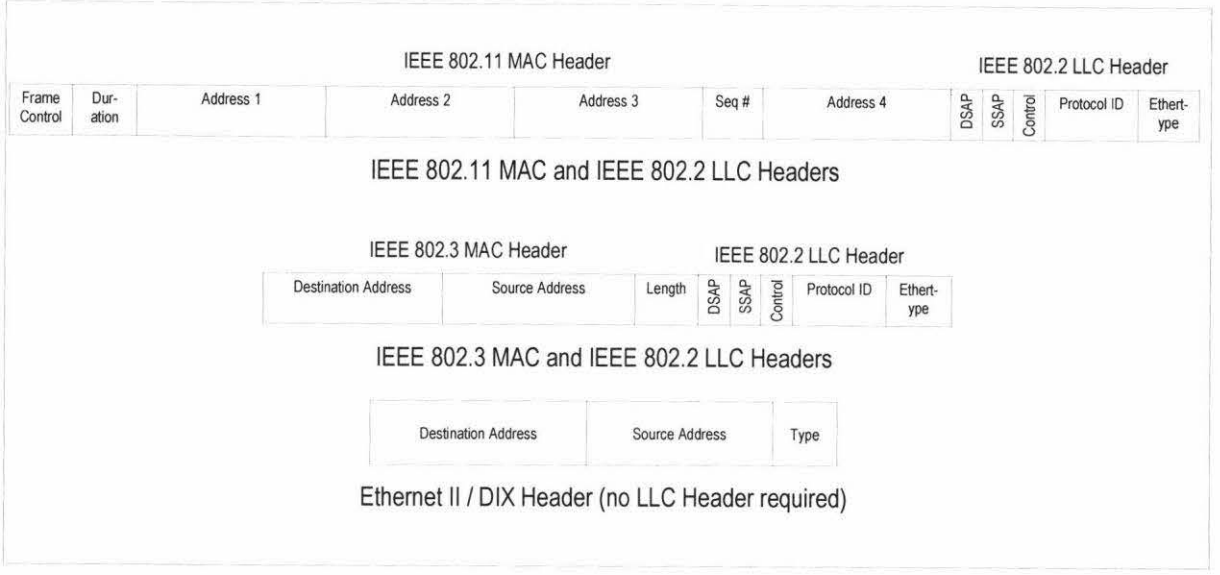


Figure 2.5 Difference between the IEEE 802.11, IEEE 802.3 and Ethernet II Frame Headers

device to be transmitted and vice versa for packets being received. This process works well for basic wireless networking needs, but as explained later, it creates some limitations for more advanced wireless networking needs such as those encountered in this research.

2.3 The Ad-Hoc On-demand Distance Vector (AODV) Routing Protocol

AODV is an ad-hoc, routing protocol capable of forming multiple hop routes. A route is a pathway from a start to a destination, while a routing protocol is a standard that defines how the correct pathway is found and maintained. Within a WLAN, the process of routing involves addressing the network data with an address obtained by the routing protocol so that the network data travels by the correct path from one wireless device to another until it reaches the destination.

Before wireless networks, the majority of routing protocols for LANs were designed for wired networks with fixed clients. With wired networks, such as Ethernet, dedicated hardware such as Ethernet routers, control all routing between the network clients. For these types of networks, the behavior of the networking medium – the wire cables – is generally reliable, easy to predict and constant. The overall network topology in an Ethernet network is also usually constant. The routing protocols designed for these types of networks reflected these features - they were designed to be operated by dedicated hardware in a network where the topology stayed relatively constant and the networking medium was relatively error free. These areas are quite different when applied to an ad-hoc WLAN. The topology of an ad-hoc WLAN can be changing constantly as nodes join and leave the network.

Radio signals, the networking medium used by WLANs, can be affected by the surrounding atmosphere and physical objects, creating a more error prone and unreliable medium than wire networking cables. To complicate matters more, if an ad-hoc WLAN is made up of nodes capable of forming multihop routes, every node is then acting as a router but is usually far from being dedicated hardware as apposed to routers in a wired network. Because of these reasons, routing protocols designed for wired networks are not as suitable for a multihop capable ad-hoc WLANs, and this has lead to the design of routing protocols such as the Ad-hoc On-demand Distance Vector (AODV) routing protocol.

AODV is one of a number of IP level ad-hoc routing standards being standardized by the Internet Engineering Task Force (IETF) Mobile Ad-hoc Networks (MANET) working group. AODV currently has RFC status [2] along with the DSR (Dynamic Source Routing), OLSR (Optimized Link State Routing) and TBRPF (Topology Broadcast Based on Reverse-Path Forwarding) ad-hoc routing protocols, all being worked on by the MANET group. There are also a number of other proposed ad-hoc routing protocols, although AODV has seen the most maturity due to some of the other proposed protocols having patent protection or no implementations which can be used for testing [3]. There are a number of AODV implementations with some currently being used by major wireless ad-hoc routing projects such as Locust World [4]. It was due to its popularity, maturity and readily available open source implementations that AODV was chosen as the routing protocol to focus on in this research.

AODV is classed as an on-demand routing protocol as opposed to a table based protocol. *Table 2.1* describes the function and the advantages and disadvantages of both on-demand and table based routing protocols. AODV is a distance vector type routing protocol, which means the protocol sees the shortest (or fastest) route as the best route to use. Because of this, AODV could potentially encounter problems like the "counting to infinity" problem where endless loops arise after a route is broken and nodes in the network receive out-of-date routing information from each other resulting in them seeing each other being an alternative route to the destination, but in reality there being no alternative route at all. To overcome this, AODV has incorporated the use of sequence numbers. Each AODV node has a sequence number that starts at zero and is incremented whenever routing information is sent to other nodes. The sequence number is added to the routing information that is sent, so that a receiving node can compare the sequence number that is included in the routing information, with the sequence number that it already has for the node in it's own route table, and so determining if the routing information is new information or not.

Table 2.1 Differences Between On-Demand and Table Based Routing Protocols

Type	On-Demand	Table based
Description	Obtains routing information only on-demand, and discards the information once it is no longer needed.	Stores and maintains all routing information.
Advantages	Routing tables are smaller and easier to manage, requiring less memory. Fresh routes are always used.	Once in the table, routes are quick to find.
Disadvantages	New routes can take longer to find than table based protocols	Routing tables are larger and more complex to maintain. Sometimes a route may be used from the table but instead there is a newer and better route available.

The AODV routing protocol is self starting, requiring no initial setup to fit in with an existing AODV network. AODV has low processing power requirements, has low memory overhead and also according to the AODV RFC, is quick to adapt to dynamic network conditions and has low network utilization, although these last two features depend a lot on the overall network conditions.

The basic operation of the AODV routing protocol is based around three routing message types that are used to find, create and maintain routes. The operation of the first message, the Route Request (RREQ) is shown in *Figure 2.6*. In order to find a route to a destination node, the source node will broadcast a RREQ, which is forwarded on any intermediate nodes, until it reaches the destination node. Each intermediate node saves a reverse route back to the source node as it forwards on the RREQ. This reverse route is created by setting the IP address of the node that the RREQ was received from as the gateway back to the source node specified in the RREQ. Each RREQ received by a node can be uniquely identified by the combination of an identification number that is included in the RREQ message and the IP address of the the source node that the RREQ originally came from. Being able to uniquely identify each RREQ controls the broadcast of the message across the network, as each node will know if the RREQ is new, or one that it has already forwarded on. If the RREQ is identified as being old information, then it will be discarded.

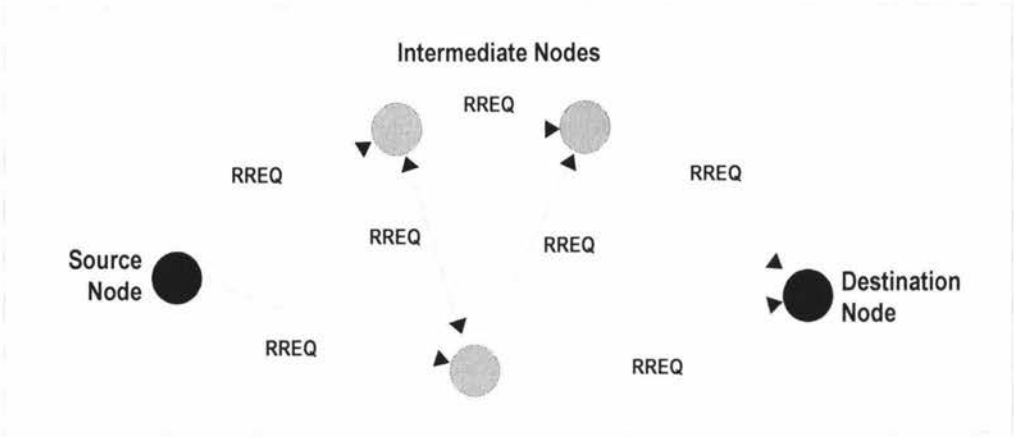


Figure 2.6 The route request process

When the first RREQ reaches the destination node, the destination node firstly creates a route back to the source node by setting the node that the RREQ was received from as the gateway back to the source node. As each RREQ can be uniquely identified, after receiving the first RREQ from the source node, the destination node discards any duplicate RREQs that have taken a longer path and are received at a later point. By responding to the first RREQ received shows the distance vector nature of AODV as it uses the shortest or fastest route as the preferred route.

The destination node then generates the second major AODV message, the Route Reply (RREP). See *Figure 2.7*. The destination node sends the RREP to the next hop node in the route back to the source node. The RREP then travels back along the reverse route to the source node, with all the intermediate nodes using the RREP to create a route back to the destination node, in the same way as the RREQ was used to create a route to the source node. When the RREP reaches the source node, the source node creates a route back to the destination node and the route formation is complete.

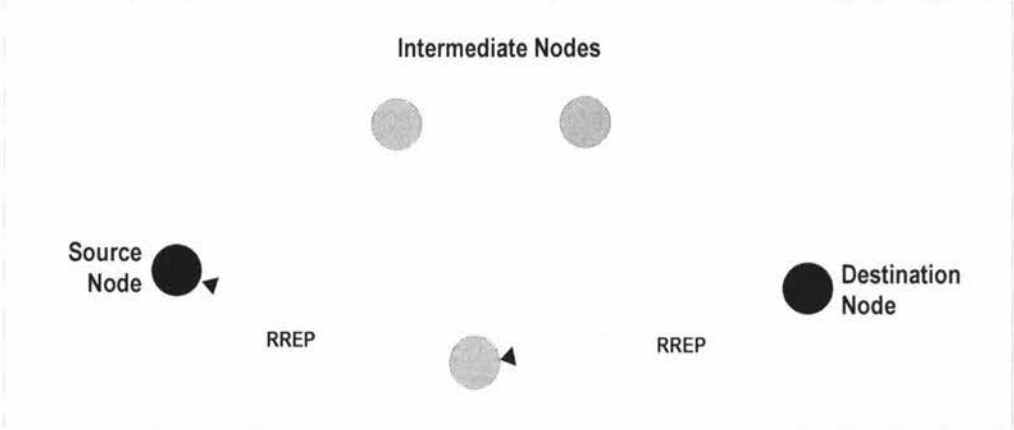


Figure 2.7 The route reply process

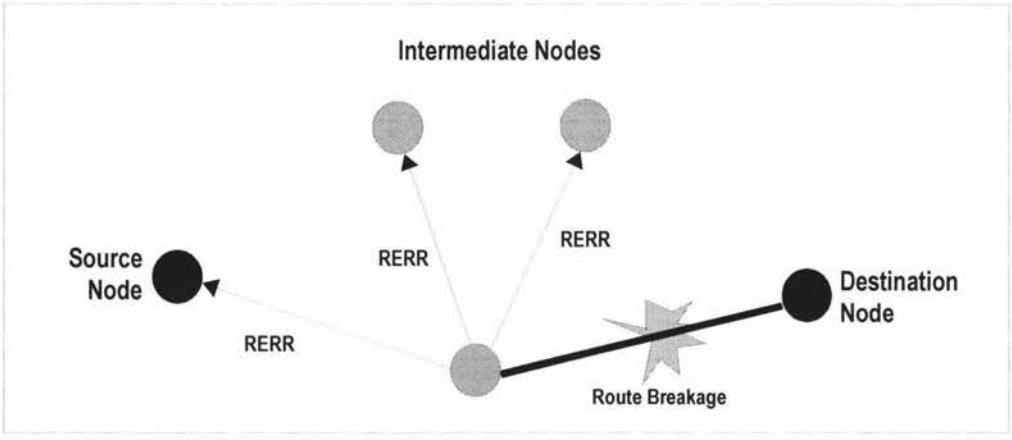


Figure 2.8 The route reply process

The third type of AODV routing message is the Route Error (RERR). As shown in *Figure 2.8*, when a link to another node is detected by a node as being broken, the node will firstly remove all routes that use the broken link, generate a RERR, and then broadcast it to all neighboring nodes so that they also can remove any affected routes. Only the nodes that receive the RERR, and are part of a route that includes the broken link, will rebroadcast the RERR. This way the RERR eventually makes it back to the sources of any routes that include the broken link, without flooding the network by simply sending it to every node. After receiving the RERR message identifying the broken route, the source node will start the route request process again to find an alternative route to the destination node. This is shown in *Figure 2.9*.

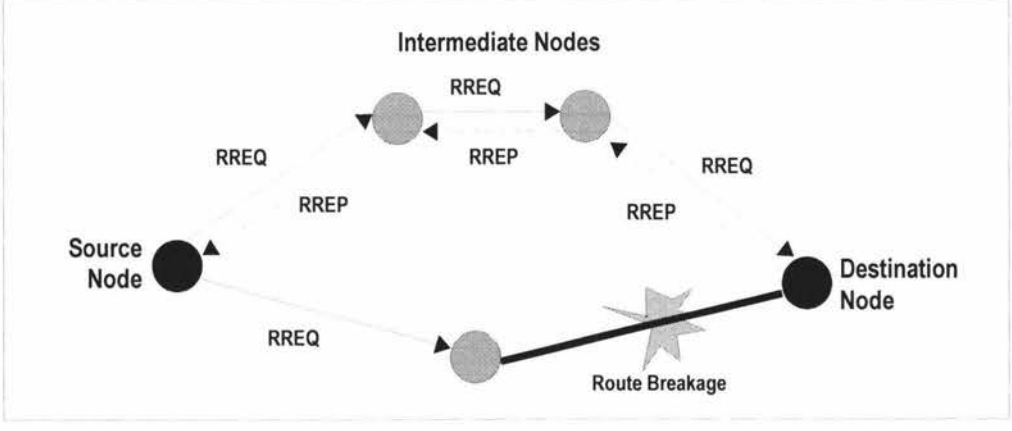


Figure 2.9 Finding an alternate route after receiving a RERR

3 The Hello Message Problem

The AODV routing protocol has been designed to be able to adapt quickly to dynamic link conditions. If a node detects that the link to another node is broken, AODV responds by: deleting routes in the node's route table that use the broken link; notifying affected nodes about the route changes; and then finding an alternate route to the destination node. A node that can detect broken links, does so by monitoring connectivity to any surrounding nodes that are part of active routes and are a single hop away. The AODV protocol doesn't define a single mechanism to monitor connectivity, but instead suggests using any of the following:

- ***Hello messages.*** Every node in the network periodically broadcasts a hello message. Nodes can then determine connectivity by listening for hello messages from neighboring nodes. When a node first receives a hello message from a neighboring node, the neighboring node is added to a list and from there forth, the connectivity to the neighboring node is monitored. If no hello messages are received from the neighboring node within a timeout period, then the link to the neighboring node is considered broken.
- ***Data link layer notifications***. These are mechanisms provided by the IEEE 802.11 standard. Notifications available on the data link layer include the data link layer acknowledgment (ACK) packet sent from the receiving node after every data packet received from the transmitting node, or the clear-to-send (CTS) packet from the receiving node after the transmitting node has sent a request-to-send (RTS) packet. By listening for either the data link layer ACK or the CTS, connectivity to the node can be monitored.
- ***Passive acknowledgments when the next hop is expected to forward a packet***. After a node forwards a packet to the next hop in a route, it can then listen for any transmission attempts by this

next hop node to determine if the link is still usable. If no transmission attempts are made within a timeout, or the next hop is the destination node (and therefore the packet is never supposed to be forwarded), then the following methods can be used to determine connectivity:

- receiving any packet (including a hello message) from the next hop
- detecting a RREQ being broadcast by the next hop
- sending an ICMP echo request to the next hop

All the popular implementations of the AODV routing protocol [5,6,7,8] that were investigated for this research currently used hello messages to monitor connectivity. Hello messages is currently the only method used in AODV implementations to monitor connectivity according to Chakeres and Belding-Royer [9]. Many other ad-hoc routing protocols also use techniques similar to hello messages to monitor connectivity. As described next, past research has shown that the use of hello messages is not an ideal mechanism to monitor connectivity.

3.1 Gray Zone Problems

Lundgren, Nordström, and Tshudin [10] showed that using hello messages with the AODV routing protocol can lead to the formation of unusable routes due to what is known as gray zone problems. They identified four factors involving hello messages that can create gray zones and result in unusable routes.

- *Different transmission rate.* When AODV is used with IEEE 802.11 wireless devices, broadcast packets such as hello messages are transmitted at the slower rate of 1Mbps as compared to data packets, which are usually transmitted at a faster rate, such as 11Mbps for IEEE 802.11b devices. This results in hello messages being able to travel further than data packets, which means a node may think a link is usable, since hello messages are being received on it, but in reality the link is too long for data packets being transmitted at the higher rates.
- *No acknowledgments.* With the IEEE 802.11 standard, broadcast packets such as hello messages require no data link layer ACK to be transmitted back by the receiving device. This means hello messages could potentially be sent over a unidirectional link, meaning a node that receives hello messages, may not be able to send data back across the same link.

- *Small packet size.* The AODV hello message is small in comparison to an average data packet. This has the same effect as the different transmission rate factor, where the small hello message will be more likely to travel further than the larger data packets. This in turn will result in a link that appears usable but is actually too long for large data packets.
- *Fluctuating links.* The range for which a device can transmit fluctuates and is not constant. At the transmission borderline of a device, communication is not reliable. A device operating on the transmission borderline of another may receive a hello message from this device, but the received hello message doesn't properly represent the quality of the link. As a result long reliable links may be replaced by shorter unreliable links.

Lundgren, Nordström, and Tshudin's research proposed three solutions to help overcome the above problems:

- *Exchanging neighbor sets.* This addresses the unidirectional link related problems. By adding an extension to the hello message, nodes could include in it their current set of neighbors (other nodes they had received hello messages from). Nodes could then compare neighbor sets when a hello message is received, so as to determine if the link is bidirectional.
- *SNR Threshold.* Signal-to-noise ratio (SNR) information from the IEEE 802.11 driver could be used to filter out weak links. By only processing hello messages that are received with a signal quality above a threshold, AODV will be forced to use routes with stronger signals, reducing the chances of a link where hello messages can get through but not data packets.
- *N-Consecutive hello messages.* This will help address the fluctuating link problem. If at least two or three hello messages have to be received from a source before being accepted, it will help remove the problem of routes being set up across unreliable links such as on transmission borderlines.

In the experiments where these solutions were implemented, it was found all reduced packet loss, with the SNR threshold solution being the most promising.

3.2 Transmission Rate and Packet Size Problems

Chakeres and Belding-Royer [11] also found that because the packet size of a hello message is small in comparison to an average data packet, along with the slower transmission rate that hello messages are transmitted at, unreliable routes can form, resulting in large packet losses. Their solution was to have the same transmission rates for both broadcast and data packets, as well as increasing the packet size of the hello message from 20 bytes to 512 bytes. Experiments where they varied the data transmission rate from 11Mbps to 1Mbps (the same as hello messages), increased the packet delivery rate from 60.7% to 84.5% and when the hello message packet size was varied from 20 bytes to 512 bytes, the delivery rate increased from 60.7% to 80.8%.

3.3 Overhead and Power Related Problems

A more obvious downside of hello messages, is the fact they increase the overhead of the network due to every node having to broadcast a hello message periodically. The periodic transmission rate by default for most AODV implementations is once every second. Network routing protocols are designed to keep overhead to a minimum in order to increase the maximum data throughput possible. The increased overhead from the hello messages may not be noticeable for a smaller network, but the impact could much larger for a large network. Past research [3] as stated that the ideal scenario where a hundred nodes can communicate with each other in a conference room using AODV routing, is in reality not possible as the hello messages from all the nodes would kill AODV along with all other AODV based communication. There is also the increased power usage resulting from hello messages. Although power usage is generally not an issue for fixed computers or most modern mobile laptops, it could be an issue if the AODV protocol was ever used in tiny networking devices that operated on small amounts of power. As well as each hello message requiring power to transmit, every hello message received needs to be processed which in turn requires power. In a large network, the number of hello messages having to process every second could be large.

3.4 Past Research about Data Link Layer Notifications and AODV

The other method to monitor connectivity which AODV supports is through data link layer detection.

This is where IEEE 802.11 mechanisms are used to notify AODV of broken links. Past research [12, 13, 14, 15] that has simulated AODV using link layer feedback to detect broken links, has shown that in the simulations, the AODV-LL (the AODV implementation using link layer feedback) performed better than the AODV-HM (the AODV implementation using hello messages). As well as performing better, the AODV-LL wasn't subject to the gray zone problems described earlier, which were a result of using hello messages. Despite the apparent benefits of AODV-LL over AODV with hello messages, there are currently no implementations of AODV that make use of data link layer feedback to detect broken links.

The reason for the lack of AODV-LL implementations, is that it is believed it is currently not possible for the upper network layers which AODV operates on, to be able to access the link layer information that would enable the detection of broken links [9, 16, 3]. Research by Lundgren, Nordström, and Tshudin [3] which looked at implementing ad-hoc routing protocols in real scenarios, commented that the ease of being able to simulate AODV-LL, along with the better performance that it delivers, has meant there have been a number of research experiments which have compared AODV-LL (rather than AODV-HM) to other ad-hoc routing protocols, even through currently AODV-LL hasn't been able to be implemented. Based on experience with testing with real world implementations of AODV, along with carrying out AODV simulations, their research also stated that simulations using AODV were often significantly different to how AODV behaved in a real scenario. For example, the gray zone problems described earlier were found as a result of real world testing rather than through simulations. One reason for this, was because the popular simulation models being used simulated all packets being sent at the slower transmission rate of 2Mbps, regardless of whether it was a control or data packet. This meant the problem of the hello messages traveling further than the data packets never existed.

Hello messages is currently the only technique used by AODV implementations to monitor connectivity with surrounding nodes. Hello messages have been found to cause problems in many common scenarios, whereas in simulations, AODV implementations using link layer feedback have performed better and have not had the same problems as hello messages. A goal of this research was that using off-the-shelf IEEE 802.11b devices, a real (non-simulated) and usable AODV implementation that uses link layer feedback to detect broken routes could be created.

4 Data Link Layer Feedback

The Open Systems Interconnection (OSI) networking model shows the data link layer being located between the physical and network layers. The data link layer is then divided into two sublayers when defining LAN and MAN networking protocols. On the bottom half is the medium access control (MAC) sublayer and on the top half is the logical link control (LLC) sublayer. The MAC sublayer controls how the physical networking device accesses the network medium in an organized way so that all the physical network devices that are part of the network can cooperate with each other so that the network is usable and not a random mess. The LLC sublayer is responsible for providing services that allow upper layers, specifically the network layer, to communicate with the MAC sublayer and vice versa. As shown in *Figure 4.1*, for every LAN/MAN physical network device supported by the IEEE committee, there is a different IEEE MAC sublayer standard. This is because each physical networking device either uses a different network medium which has to be accessed in a different manner, or the same

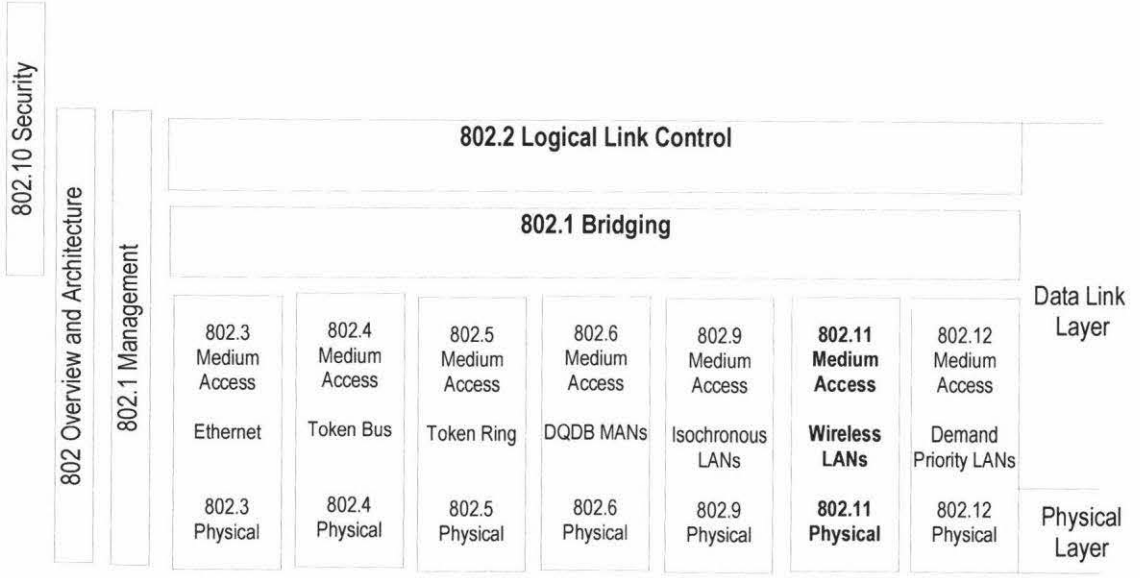


Figure 4.1 IEEE LAN/MAN Standards within the Physical and Data Link layers - numbers are the IEEE standard number.

network medium is used but requires a different way of accessing it. Each MAC definition provides common services to the LLC sublayer which allows for a common LLC sublayer standard to act as an interface between the network layer and MAC sublayers. Since part of this research focused on data link layer feedback for AODV, the IEEE 802.11 MAC wireless LAN standard and the IEEE 802.2 LLC standard were investigated for ways by which feedback about link status could be obtained.

4.1 Logical Link Control Sublayer

The IEEE 802.2 LLC (logical link control) sublayer provides an interface for the network layer to access the MAC (medium access control) sublayer. This research was based around the IEEE 802.11 wireless standard which operates on the MAC sublayer, and therefore; the IEEE 802.2 LLC sublayer standard was researched to identify any possible ways by which the network layer could access feedback from the IEEE 802.11 MAC sublayer about the status of transmitted packets. By achieving this, network layer protocols such as AODV, could monitor the connectivity status of links without the need for hello messages.

The IEEE 802.2 LLC sublayer acts as an interface by providing services to the network layer that can be used to access the MAC sublayer. It provides three forms of services to the network layer:

1. Unacknowledged connectionless-mode services
2. Connection-mode services
3. Acknowledged connectionless-mode services

The unacknowledged connectionless-mode services are fairly limited and provide no means of obtaining transmission status feedback from the MAC layer. Within the connection-mode services, the LLC sublayer establishes a data connection with the destination LLC and manages the connection itself in terms of error recovery and flow control. For this type of service, the LLC provides no information to the network layer about individually transmitted packets or though it does notify the network layer if the connection has been terminated using the DL-DISCONNECT indication service. The form of service that would provide the most information concerning link connectivity status, is the acknowledged connectionless-mode services. For this type of service , the LLC sublayer provides a primitive known as DL-DATA-ACK-STATUS indication. This primitive is passed from the LLC sublayer to the network layer after the network layer has sent data to the LLC sublayer to be transmitted. As it includes the

result of the data transmission – whether it was successful or not - the DL-DATA-ACK-STATUS indication primitive could be used by the routing protocol on the network layer to identify whether a link is still operational or whether it is broken. Unfortunately, as explained earlier, all wireless packets within the Linux 2.4 kernel are converted to DIX Ethernet packets, which unlike the IEEE 802.3 Ethernet packet, do not make use of the IEEE 802.2 LLC header, or any of the IEEE 802.2 LLC sublayer services. Therefore, since the Linux operating system was needed to be used for any implementations and rewriting the Linux kernel so that it included a LLC sublayer and handled wireless packets correctly was far beyond the goal of this research, the IEEE 802.2 LLC sublayer services were researched no further. Instead the IEEE 802.11 MAC sublayer was researched for ways by which it could provide feedback directly to the network layer.

4.2 Medium Access Control Sublayer

The IEEE 802.11 MAC standard controls how the physical wireless device accesses the wireless medium. The standard was researched for possible functions that could be used to provide feedback to the network layer about the status of a link. Any possible functions were then investigated in order to identify if it would be possible to use them in a real implementation with the wireless hardware used in this research. Since this research focuses on ad-hoc wireless networks, only the functions for the IBSS type (ad-hoc) IEEE 802.11 networks were researched. Two functions within the IEEE 802.11 standard were identified as being possible means of providing feedback to the network layer about the status of a link. They were the MAC sublayer acknowledgments (ACKs) and the MAC sublayer beacons.

Beacons are used to maintain synchronization over the whole IBSS wireless network. The beacon generation is distributed, which means every node in the network participates in beacon generation, rather than just a single node. The node which starts the network, starts sending beacons and sets the beacon interval. After every beacon interval, each node in the network will wait until the medium is free (no other transmissions taking place) then wait a random amount of time (the backoff period) where the node listens to the medium for any activity. If there is no activity after the end of the backoff period, the node transmits a beacon, but if during the backoff period a beacon is received from another node (which happened to have a shorter backoff period) then the node aborts transmitting a beacon until the next beacon interval. If during the backoff period, other activity is heard, the node waits until the medium is free and then waits another backoff period before attempting to send a beacon. This

distributed approach to beacon generation results in every node in the network having a chance at sending a beacon every beacon interval. But since a random length backoff is involved, only one node each beacon interval actually results in sending a beacon. The beacon frame contains information which allows the receiving node to synchronize to the other nodes in the network. Since the beacon frame also contains the address of the node that transmitted it, beacons could be used in a similar way to hello messages, where by listening for beacons from a particular node could determine whether a link to that node is usable or not. For the hardware used in this research, the beacon interval was 0.1 seconds.

The operation of MAC sublayer ACKs is less complicated then the beacon operation. The IEEE 802.11 standard defines that for every frame transmitted, the receiving node must transmit back a MAC sublayer ACK. Unlike the beacon frame format, ACK frames only contain the address of the sender node that initially transmitted the frame. When the MAC transmits a frame, it waits till the ACK for the frame is received. When it receives an ACK addressed to itself, the MAC then knows it is the ACK it has been waiting for. A node could listen for an expected ACK after transmitting a frame to determine if the link is usable or not, therefore using the ACKs as a feedback technique concerning the status of a link. *Table 4.1* shows the advantages and disadvantages of both the beacon technique and the MAC sublayer ACK technique.

Table 4.1 Advantages and Disadvantages of Beacons and MAC Acknowledgments

Link Notification Technique	Beacons	Acknowledgments
Advantages	- Happening all the time - Frame contains more information the the MAC ACK.	- Only used when a packet is transmitted making them more suited to on-demand type protocols
Disadvantages	- Beacons are normally only processed by the MAC chipset on the wireless network device. If the Network protocol started processing every beacon this would mean more extra work than processing hello messages (Hello messages normally sent every second, beacons every 0.1 seconds). - Due to the random backoff, for small networks , the time between each node having a turn to transmit a beacon will be quite small. For larger networks, the time between each node having a turn to transmit a beacon could be larger, resulting in the misinterpretation of broken links to certain nodes, due to the absence of beacons from them.	- ACK frames only contain the original frame sender address. This means the Network protocol would need to do some extra work to determine which node the ACK came from to determine which link or route information to update.

Based on the advantages and disadvantages of both link notification techniques, it was decided that MAC sublayer Acknowledgments would be the technique used to monitor the status of links.

4.3 Using the Transmit Retry Limit

Past research [16] has stated that for current IEEE 802.11 hardware, the MAC sublayer ACKs are not accessible by the upper network layers, making it impossible to use them for link notification purposes. For the IEEE 802.11 hardware used in this research, it was found in the implementation stage that the IEEE 802.11 device driver, which provided the interface between the network layer and the physical device, had no way of directly accessing the MAC sublayer ACKs. A method was found though that could be used to indirectly access the MAC sublayer ACKs and provided more useful information then using the ACKs themselves. This method was to monitor the transmit retry limit (TRL).

The IEEE 802.11 standard states that within an IBSS wireless network, for every data frame transmitted, the receiving node must acknowledge the reception of the data frame by transmitting back a MAC sublayer acknowledgment (ACK) control frame. If the original transmitting node does not receive back an ACK after a set period of time, then it will retry transmitting the data frame. There are limits though to how many times a node can retry transmitting a data frame, else a node might continue to retry forever, which could happen if a node receiving the data left the wireless network. The IEEE 802.11 standard specifies two retry limits along with two corresponding retry limit counters: the short retry limit (SRL) and the short retry count (SRC) as well as the long retry limit (LRL) and the long retry count (LRC). Whenever an ACK isn't received after a data frame transmission, then one of the counters is incremented. Which counter is incremented is determined by whether the size of the data frame in the failed transmission is larger or smaller than a preset threshold. If the data frame size is smaller, then the SRC is incremented, otherwise the LRC is incremented. If either counter reach their corresponding limit, then the retry attempts are ceased and the data frame is discarded. If a MAC sublayer ACK is received after transmitting a data frame, then the corresponding counter is reset based on the data frame size. Both retry limits are configurable parameters and having two limits with the threshold based on the data frame size can help improve reliability. For example the LRL may be set higher than the SRL since larger packets are more prone to transmission failure than smaller packets.

If configured correctly, the short and long TRLs provide notification of a broken link whereas failing to

receive a MAC sublayer ACK only provides notification of a failed transmission. After failing to receive an ACK, the ACKs still need to be monitored, since it is only after several failed transmissions that it can be assumed that the link is broken. Monitoring the result of several transmission attempts is what the TRL process already does, making it more useful feedback than straight ACKs (see *Figure 4.2*).

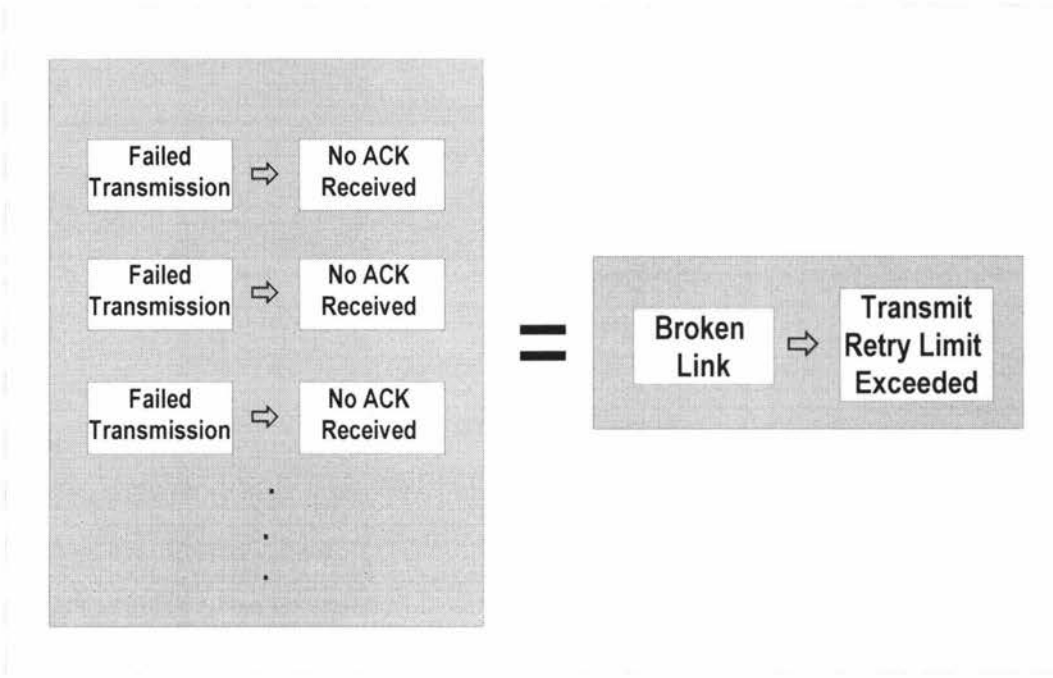


Figure 4.2 Difference between the information implied by not receiving a MAC sublayer ACK and the transmit retry limit being exceeded.

4.4 Comparing Hello Messages and Data Link Layer Feedback as Error Detection Techniques

Using hello messages or data link layer feedback represents two significantly different styles of detecting broken links. Firstly it is important to note that within this research, the term "detecting a broken link" is simply the process of determining that the actual connection to a neighboring node one hop away no longer exists. This is different to receiving an AODV route error (RERR) message which informs the node that the route to a destination node is broken. A whole *route* is broken due to a single *link* being broken, and it is the broken *link* that a node must be able to detect to inform other nodes with a RERR that the *route* is broken. Hello messages represent a constant polling style of detecting broken links, whereas using data link layer feedback with the transmit retry limit technique as proposed in this research, represents a more variable method but with a much more "on-demand" characteristic that

suits the AODV protocol.

The default settings, set by the AODV protocol, specify that hello messages are transmitted once every second, and once two hello messages are not received from a neighboring node, then the link to that node is considered broken. With data link layer feedback, and the transmit retry limit, if the MAC layer on the wireless interface within a set timeout period fails to receive a MAC layer acknowledgment after transmitting a data packet, then the wireless interface will retry transmitting the packet. If still no acknowledgments are received and the number of retry attempts exceed the transmit retry limit, then the link is considered broken. The time a wireless interface waits for an acknowledgment to arrive back after transmitting a data packet is not specified by the IEEE 802.11 protocol, instead it is set by the wireless interface vendors. Results of tests done as part of this research though showed that the time between attempting to transmit a data packet, detecting the link is broken and then the data link layer generating feedback as a result to inform AODV, was around 0.13 seconds.

Data link layer feedback, using the transmit retry limit, results in fast detections of broken links, but it is only through an attempt to send data across the broken link that the broken link can be detected, giving the transmit retry limit technique a variable and on-demand style. In comparison, hello messages, result in longer times to detect broken links, but since they are always being transmitted, hello messages are capable of detecting broken links despite whether data is being transmitted over the link or not, resulting in them being a more constant error detection technique. These characteristics could result in the following scenarios occurring:

1. A link is broken while data is being transferred across it. The data link layer feedback based AODV implementation (AODV-LL) would quickly detect the broken link and respond by finding an alternative route around it. The hello message based AODV implementation (AODV-HM) using default settings would take at least two seconds to detect the broken link and respond by finding the same alternative route as the AODV-LL implementation. Assuming the process of finding the new route takes the same for both AODV types, then overall the AODV-LL implementation would detect and repair the broken link and have the data transaction up and running again the fastest.

2. A link is broken while no data is being transferred across it but ten seconds later the link is needed again to transfer data. The AODV-LL implementation doesn't detect the broken link

until ten seconds after it is broken when it attempts to send data across it. The AODV-LL implementation now has to halt the data transfer until it finds an alternative route around the broken link. The AODV-HM implementation detects the broken link two seconds after it is broken as in the first scenario. An alternative route is then found before the link is needed, causing no disruptions to the data transfer. Overall the AODV-HM implementation would be the fastest to detect, repair and start the data transaction.

These two scenarios show more clearly the constant nature of hello messages compared to the more variable and on-demand nature of data link layer feedback. From the second scenario, it appears that the AODV-LL is at a disadvantage by detecting the broken link only when it is needed. Using default AODV values through, the result of the second scenario would actually be different Part of the AODV protocol is an active route timeout to keep routing tables current and up-to-date. If no data is received or transmitted across a route within the timeout period, then the route expires and is later deleted. If the node tries to then connect to any nodes that are part of the expired route, the normal route finding process is required to set up a new route. The default active route timeout is two seconds. This means that in the case of scenario two, since the route is not used for more then ten seconds, then it would be deleted anyway, which would result in the AODV-HM implementation having to go through the same route finding process as the AODV-LL implementation. The only time then the AODV-HM implementation would save, is that it would already know the link is broken and so immediately go about the route finding process, whereas the AODV-LL implementation would first have to detect the route is broken. If an active route time less then ten seconds was being used by the AODV-LL implementation as well then it would also immediately go about finding a new route and the AODV-HM implementation would be no faster.

Theoretically therefore; an AODV implementation using data link layer feedback and the transmit retry limit to detect broken routes should be able to detect and respond to a broken route faster or just as fast as a hello message based AODV implementation. Evidence of this is shown later in the research results.

5 The Multiple Hop Route Throughput Problem

Within an ad-hoc IEEE 802.11 wireless network, the throughput over multiple hop routes decreases rapidly as the number of hops in the route increase. This decrease in the throughput is mostly caused by the IEEE 802.11 overhead operations that take place at each hop. As the number of hops in a route increases, the overhead for the data being transmitted across the route increases and the overall throughput decreases. This problem severely affects the number of hops in a route possible before the route becomes usable due to the throughput being so low. It was recognized that this was a problem that needed to be addressed for a multiple hop routing protocol such as the AODV routing protocol to be feasible in a real world, wireless, ad-hoc network. To understand the cause of the rapid throughput decrease, the IEEE 802.11 MAC sublayer ad-hoc wireless network access method was researched.

5.1 The Access Method for the IEEE 802.11 IBSS Wireless Network

The MAC sublayer is responsible for controlling how the the network medium is accessed. For an IEEE 802.11 IBSS wireless network, the fundamental access method used by the MAC sublayer is a distributed coordination function (DCF) known as Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). The CSMA/CA algorithm ensures that nodes in the network can access and use the wireless medium in an organized manner. The following describes how the CSMA/CA algorithm operates (see also *Figure 5.1)*.

When a node in the network wants to transmit, it will firstly sense or listen for any activity on the medium. If the medium is sensed as being busy, then the node will wait. If no activity is sensed on the

medium for a length of time called the DCF Interframe Space (DIFS), then the node will be able to contend for the medium. To contend for the medium the node will wait for a random amount of time known as the random backoff period, sensing the medium while it is waiting. The random backoff period helps ensure that each node in the network has a fair chance of accessing the network medium and decreases the chances of two or more nodes trying to transmit at the same time, resulting in a collision. If during the random backoff period, activity on the medium is sensed, then the node will start over – waiting until the activity has stopped and the medium is free for a duration equal to a DIFS. If during the random backoff period no activity is sensed, then the node will immediately transmit the data frame. The node that receives the data frame will wait for a length of time called a Short Interframe Space (SIFS) before sending a acknowledgment (ACK) frame back to the original transmitting node. The SIFS is deliberately shorter than the DIFS to help ensure that a node should always be able to transmit an ACK without resulting in a collision and the ACK being lost. If the original transmitting node doesn't receive an ACK from the receiving node within a set length of time, then it will retry transmitting the data frame using the CSMA/CA method. The IEEE 802.11 standard defines a refinement to the access method to further help avoid collisions which uses Request To Send (RTS) and Clear To Send (CTS) control frames. The RTS and CTS control frames are used by the transmitting and receiving nodes to determine if the medium is free and also to help notify surrounding nodes of the data

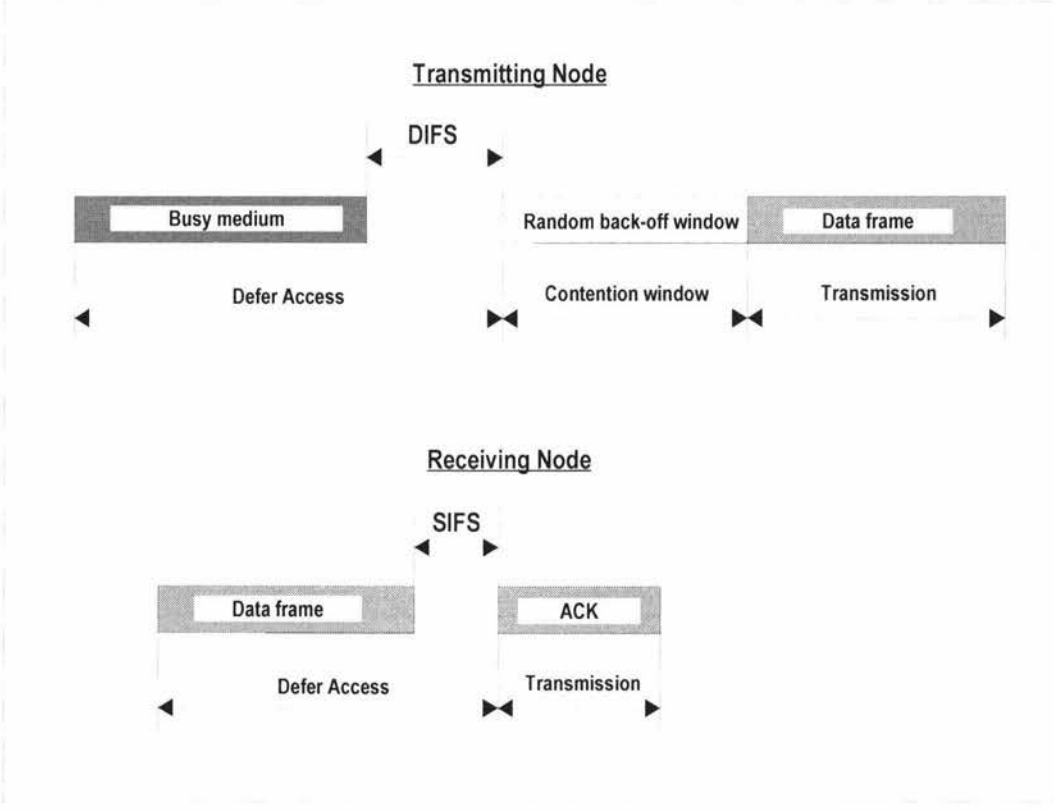


Figure 5.1 The basic IEEE 802.11 MAC sublayer CSMA/CA operation

transmission about to occur. This IEEE 802.11 standard states this is an optional method and was not used in this research.

5.1.1 The CSMA/CA Algorithm over a Multiple Hop Route

To help understand how the CSMA/CA algorithm affected throughput over a multiple hop route, a theoretical scenario involving the transfer of a single TCP data packet over a two hop route was researched. The two hop wireless route consisted of a source node; an intermediate node; and the destination node. To simplify the scenario, the following assumptions were made:

- The network is an IEEE 802.11b IBSS type network
- Frames are transmitted at 11Mbps
- The RTS/CTS collision avoidance method was not used
- The random backoff period always equaled zero
- A simple TCP connection model was used where for each TCP data packet transmitted, a TCP ACK is transmitted back
- It took zero time for the node to process all data
- The IEEE 802.11b long preamble was used

The theoretical throughput was calculated as part of the scenario research. To calculate the throughput, the entire data transaction over the two hop route was firstly broken down into the different operations that took place. The time taken for each operation to execute was then calculated. Using these times, the entire data transaction duration was calculated and then multiplied by the amount of actual data transferred to give the overall throughput rate. The data transaction was broken down into five operations:

1. DIFS wait period
2. SIFS wait period
3. TCP data packet transmission
4. TCP ACK packet transmission
5. MAC ACK transmission

For this scenario the IEEE 802.11b standard was used, which is an extension of the IEEE 802.11 standard. The IEEE 802.11b standard is the same as the original IEEE 802.11 standard but instead allows transmission rates of up to 11Mbps as apposed to only 2Mbps which is the maximum for the

IEEE 802.11 standard. For every frame transmitted, the IEEE 802.11 standard specifies that a Physical Layer Convergence Protocol (PLCP) preamble and header is transmitted before every frame. The PLCP preamble and header is transmitted at 1Mbps regardless of what rate the following frame is transmitted at so as to ensure that the IEEE 802.11 and IEEE 802.11b standard are compatible with each other. For this scenario the long PLCP preamble and header is used as this is the default for most IEEE 802.11b wireless devices. The IEEE 802.11 standard states that the long PLCP preamble and header combined is 192bits long, which when transmitted at 1Mbps, will take 192μS to transmit. The following shows the calculations involved to find the time taken for each operation to execute:

1. SIFS wait period: 10μS (from the IEEE 802.11 standard)

2. DIFS wait period: 50μS (from the IEEE 802.11 standard)

3. TCP data packet transmission:

IEEE 802.11 MAC frame size with TCP data packet encapsulated: 1536bytes

28	8	20	20	1460
IEEE 802.11 MAC Header	SNAP Encapsulation Header	IP Header	TCP Header	Data

Data frames are transmitted at 11Mbit/s which is 1375bytes/s. Time taken to transmit IEEE 802.11 MAC frame with TCP data packet encapsulated:

$$\frac{1536\,bytes}{1375\,bytes/s} = 1118\,\mu s$$

Total time to transmit TCP data packet including PLCP preamble and header:

$$1118\,\mu s + 192\,\mu s = 1310\,\mu s$$

4. TCP ACK packet transmission:

IEEE 802.11 MAC frame size with TCP ACK packet encapsulated: 76bytes

28	8	20	20
IEEE 802.11 MAC Header	SNAP Encapsulation Header	IP Header	TCP Header

Time taken to transmit IEEE 802.11 MAC frame with TCP ACK encapsulated:

$$\frac{76\,bytes}{1375\,bytes/s}=56\,\mu s$$

Total time to transmit TCP ACK packet including PLCP preamble and header:

$$56\,\mu s+192\,\mu s=248\,\mu s$$

5. MAC ACK frame transmission:

IEEE 802.11 MAC ACK frame size: 14 bytes (from IEEE 802.11 standard)

Time to transmit IEEE 802.11 MAC ACK frame:

$$\frac{14\,bytes}{1375\,bytes/s}=11\,\mu s$$

Total time to transmit IEEE 802.11 MAC ACK frame including PLCP preamble and header:

$$11\,\mu s+192\,\mu s=203\,\mu s$$

Table 5.1 shows the time taken for each operation to execute based on the previous calculations.

Table 5.1 Execution times for the different operations in a simple wireless transaction

Operation	***Time taken to execute***
SIFS wait period	10μS
DIFS wait period	50μS
TCP data packet transmission	1310μS
TCP ACK packet transmission	248μS
MAC ACK frame transmission	203μS

Using these times the theoretical throughput for the scenario could be calculated by dividing the total duration of the wireless transaction by 1460bytes (11680bits) which is the amount of actual data transmitted in a single transaction. *Figure 5.2* shows an example setup of the scenario while *Figure 5.3* shows how the scenario, consisting of a single TCP data packet transaction over a two hop wireless route using a single wireless IEEE 802.11 device in each node, would take place.

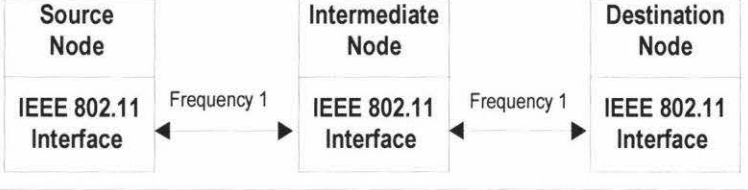


Figure 5.2 Example setup using a single IEEE 802.11 per node

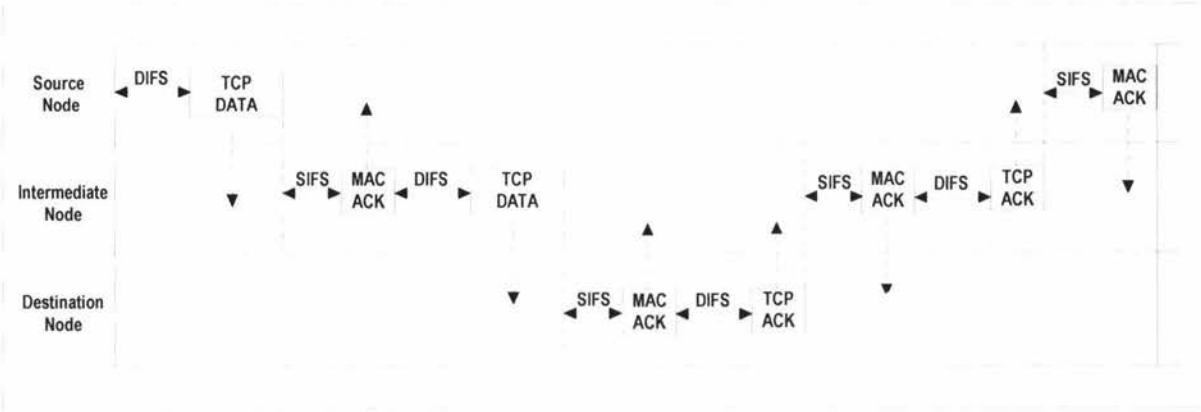


Figure 5.3 Operations taking place in a single data transaction over a two hop wireless route using a single IEEE 802.11b device per node

$$\frac{11680\,bits}{4168\,\mu s} = 2.80\,Mbps$$

The total time for the transaction to take place is: **4168µS**. Therefore the throughput is: **2.80Mbps**

Table 5.2 shows the theoretical duration and throughput rates for the same scenario but over a range of hops in the route.

Table 5.2 Duration of a single TCP data transaction and the throughput rate for a range of multiple hop routes

Number of hops in the route	***Total duration of the transaction (µS)***	***Throughput rate (Mbps)***
1	2084	5.60
2	4168	2.80
3	6252	1.87
4	8336	1.40
5	10420	1.12
6	12504	0.93
7	14588	0.80
8	16672	0.70
9	18756	0.62
10	20840	0.56

As shown by *Figure 5.4* the duration of the TCP transaction in the example scenario is directly proportional to the number of hops in the route, while *Figure 5.5* shows how the throughput rate is proportional to *1/n* where *n* is the number of hops in the route. In a real scenario, the throughput rates could be even lower due to factors not included in this example scenario, such as the random backoff and the occurrence of collisions causing repeated transmissions. Between one and four hops in the

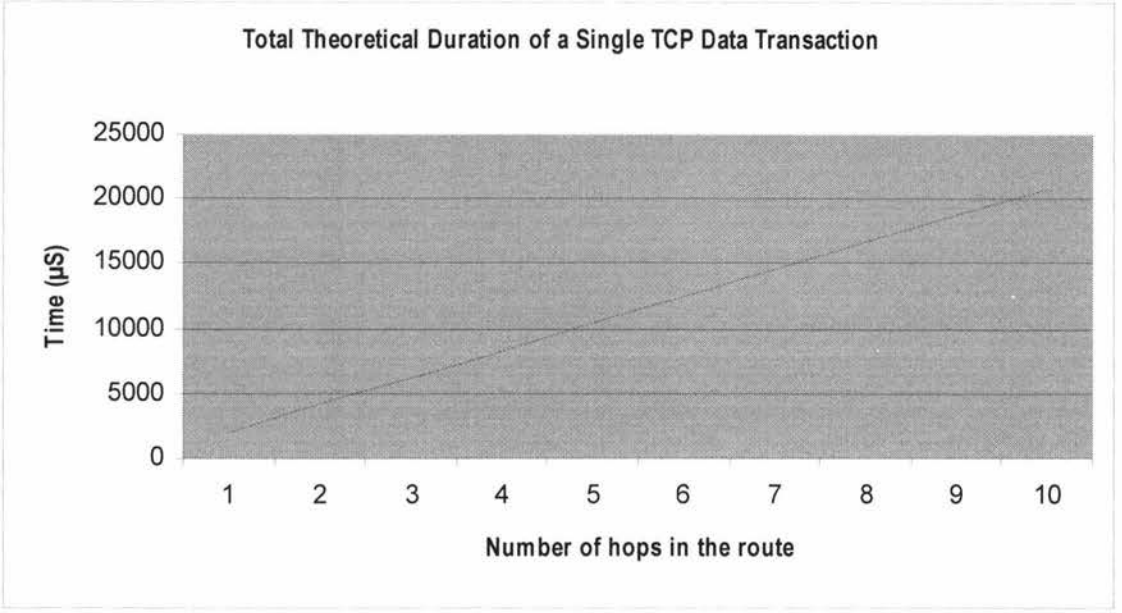


Figure 5.4 Duration of a single TCP data transaction over a range of multiple hop routes

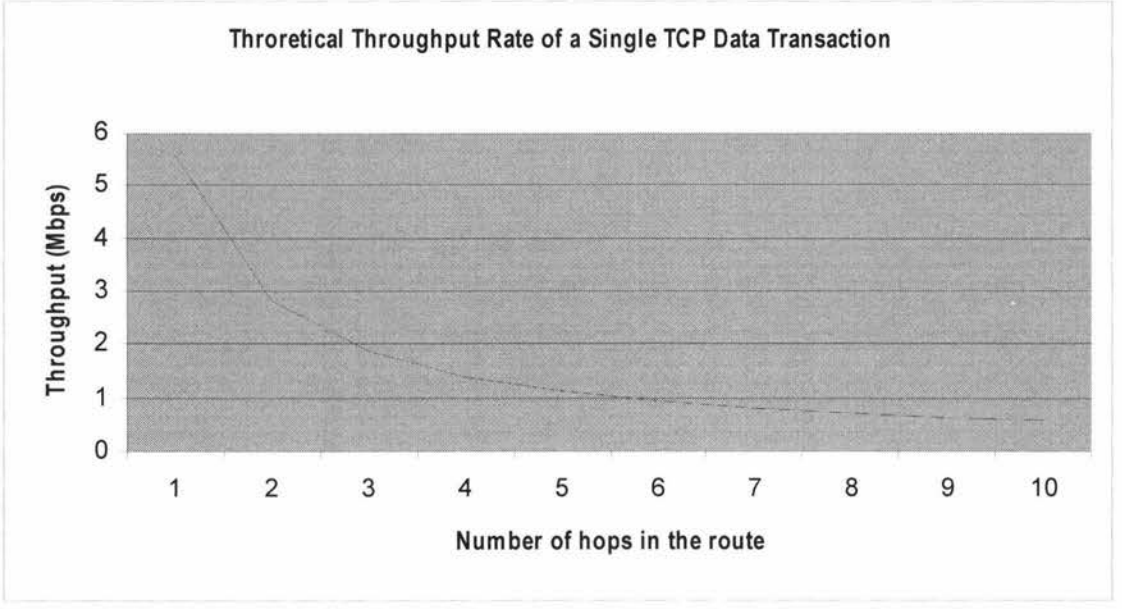


Figure 5.5 Throughput rate of a TCP data transaction over a range of multiple hop routes

route, the decrease in throughput is quite dramatic, with the throughput decreasing by 75% of the original throughput rate, whereas from four hops onwards the decrease in throughput is more gradual. From this example scenario the duration for a single data transaction for n hops in a route equaled:

$$2048\,n$$

and the throughput rate for n hops in a route equaled:

$$\frac{5.06}{n}$$

5.2 Using Multiple IEEE 802.11 Wireless Interfaces Per Node

For a route with *n* nodes in an IEEE 802.11 wireless network, the throughput rate is proportional to 1/*n* due to the overhead operations created by the CSMA/CA algorithm. Even after adding just a few hops to a route, the rapid decrease in throughput rate meant that this was an area that needed to be addressed to make AODV a feasible multiple hop routing algorithm in a real world scenario. Rewriting a new wireless network standard or even changing the IEEE 802.11 standard to address this problem was beyond the scope of this research and also impractical as it would mean the resulting new wireless standard would be incompatible with the well established IEEE 802.11 standard. A solution proposed by this research is using multiple IEEE 802.11 based wireless network interfaces per node, specifically two or four interfaces per node as opposed to using just a single interface per node. With the decreasing cost of typical IEEE 802.11 wireless devices, using multiple interfaces per node is feasible in a practical sense and also directly improves the operation of two areas of the CSMA/CA algorithm, resulting in improved throughput rates over multiple hop routes. Using multiple wireless interfaces can also improve the performance of the TCP transport protocol, one of the most common transport protocols used in computer networking.

5.2.1 Areas of the CSMA/CA Algorithm Improved by Using Multiple Wireless Interfaces per Node

From *Figure 5.3* there are two areas of the CSMA/CA algorithm that decrease the throughput rate and could be improved by using two and four interfaces per node. The first area is when an intermediate node in the route receives the data packet from the previous node in the route. Before the intermediate node can pass on the data packet to the next node in the route, it must firstly wait a SIFS period and send a MAC sublayer ACK back to the previous node in the route. If the intermediate node was equipped with two IEEE 802.11 wireless network interfaces, both operating on non-conflicting frequencies, then the intermediate node could use one interface to send the MAC ACK back to the previous node, and at the same time use the second interface to carry on forwarding the data packet to the next node in the route. As shown in *Figure 5.6,* using two interfaces per node , configured in this way, allows each hop in the route to be non-conflicting and independent of each other. *Figure 5.7* shows what the resulting CSMA/CA operations would look like for the same example scenario

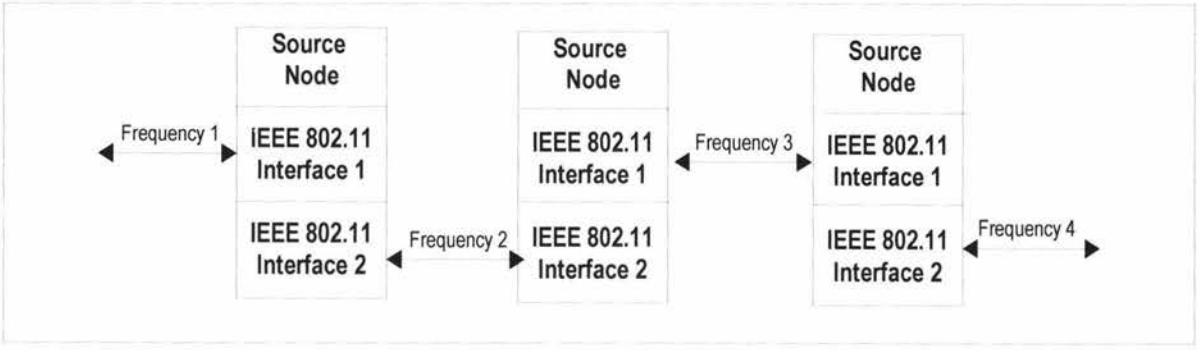


Figure 5.6 Example setup using two IEEE 802.11 interfaces per node

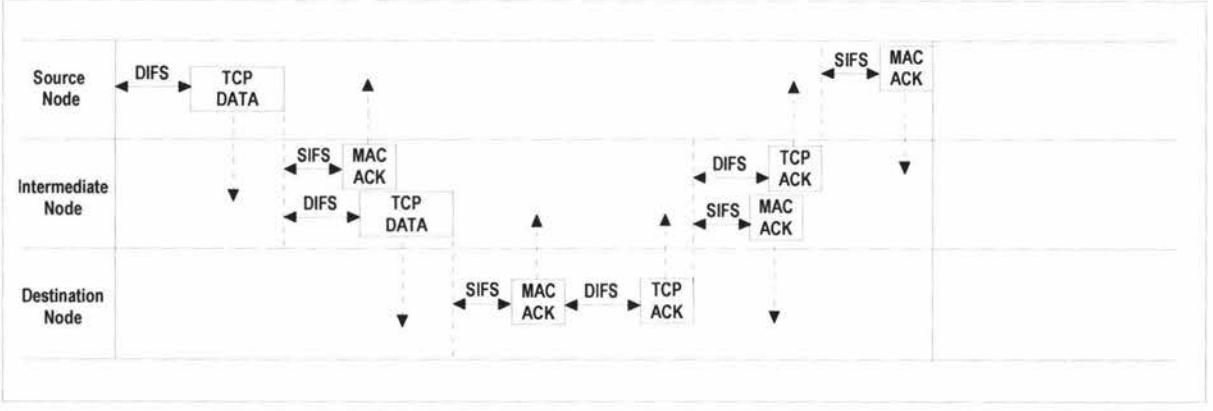


Figure 5.7 Operations taking place in a single data transaction over a two hop wireless route using two IEEE 802.11 interfaces per node

previously described. Based on the calculations used for the example scenario, using two IEEE 802.11 wireless interfaces per node in the route would result in the data transaction taking 3742μS and the having the increased throughput rate of 3.12Mbps.

The second area from *Figure 5.3* that can be improved is at the destination node. With a single interface, a node must wait until the MAC ACK is sent back to the previous node in the route before sending back the TCP ACK. If each node in the route used four interfaces, each configured to operate on non-conflicting frequencies, each hop in the route would have two channels to communicate on and with each channel operating in a different direction. This is shown more clearly in *Figure 5.8*. Using four interfaces per node would allow the destination node to transmit back the TCP ACK without having to wait for the MAC ACK to be sent first, as well as allowing the intermediate nodes to forward on the data without having to wait for the MAC ACK to be sent like with the two interface configuration. *Figure 5.9* shows what the resulting CSMA/CA operations would look like using four interfaces per node. Based on these calculations, using four IEEE 802.11 wireless interfaces per node in the route would result in the data transaction taking 3529μS and having the increased throughput rate of 3.31 Mbps.

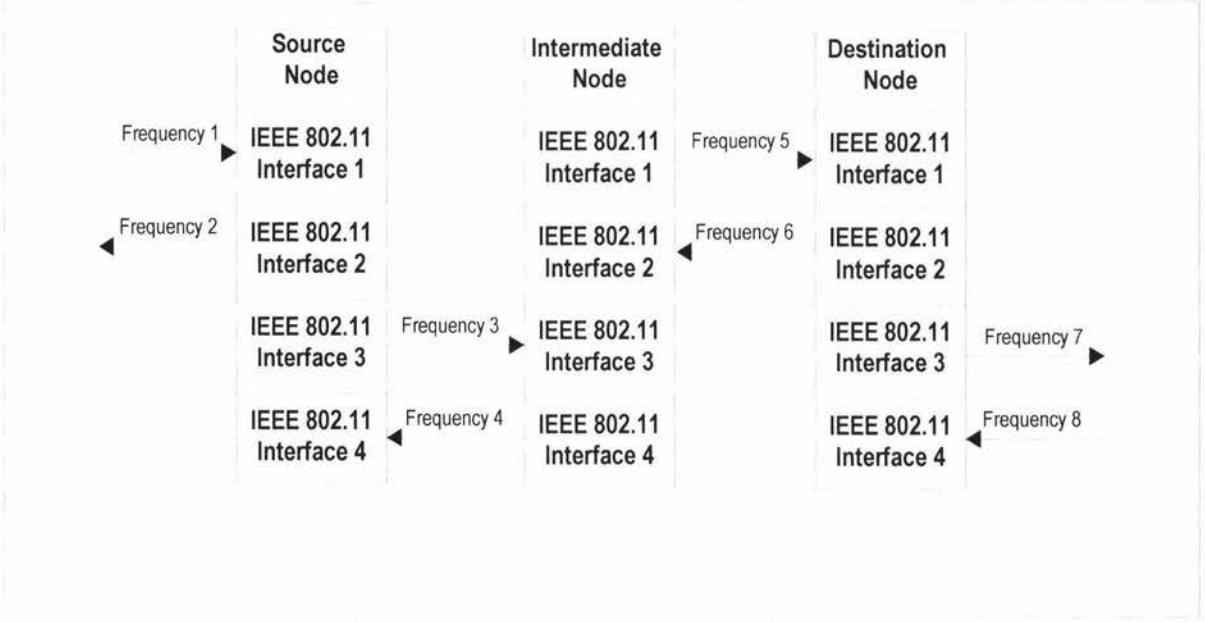


Figure 5.8 Example setup using four IEEE 802.11 interfaces per node

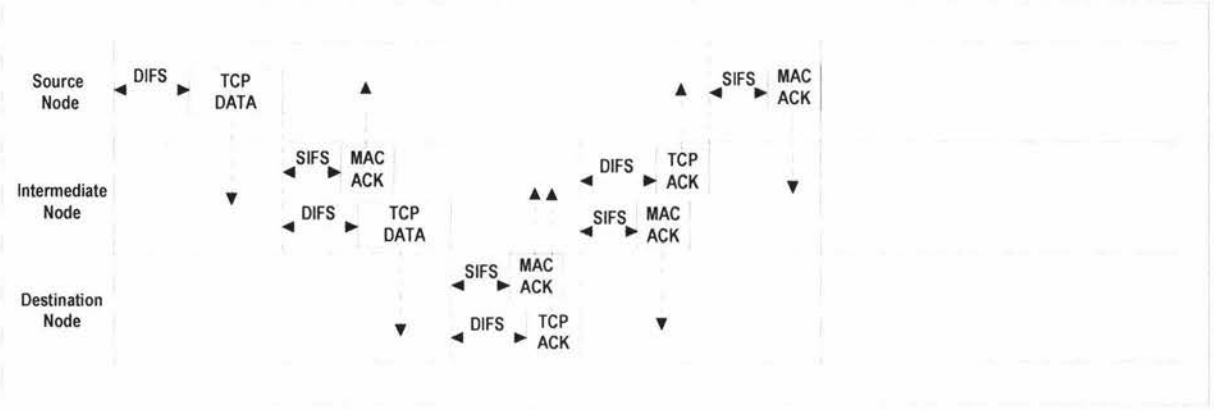


Figure 5.9 Operations taking place in a single data transaction over a two hop wireless route using four IEEE 802.11 interfaces per node

Table 5.3, *Figure 5.10* and *Figure 5.11* following compare the duration of a single TCP data transaction and the throughput rate over a range of multiple hop routes for one, two and four IEEE 802.11 interfaces per node. These theoretical throughput rates were based on the conditions of the example scenario described previously. The calculations show using four IEEE 802.11 interfaces per node has a significant improvement over using two IEEE 802.11 interfaces per node but only for routes with a small number of hops. As the number of hops in the route increases, the difference in the throughput rate between using two or four interfaces per node is minimal as the only difference between the two configurations is that using four interfaces per node decreases the turnaround time at the destination node which stays constant for any size route. Therefore based on the scenario conditions, the extra performance provided from using four interfaces in this way is only significant for a routes with a small

Table 5.3 Duration of a single TCP data transaction and the throughput rate for a range of multiple hop routes and interfaces per node.

Number of hops in route	***One Interface Per Node***		***Two Interfaces Per Node***		***Four Interfaces Per Node***	
	Duration	Throughput	Duration	Throughput	Duration	Throughput
1	2084	5.60	2084	5.60	1871	6.24
2	4168	2.80	3742	3.12	3529	3.31
3	6252	1.87	5400	2.16	5187	2.25
4	8336	1.40	7058	1.65	6845	1.71
5	10420	1.12	8716	1.34	8503	1.37
6	12504	0.93	10374	1.13	10161	1.15
7	14588	0.80	12032	0.97	11819	0.99
8	16672	0.70	13690	0.85	13477	0.87
9	18756	0.62	15348	0.76	15135	0.77
10	20840	0.56	17006	0.69	16793	0.70

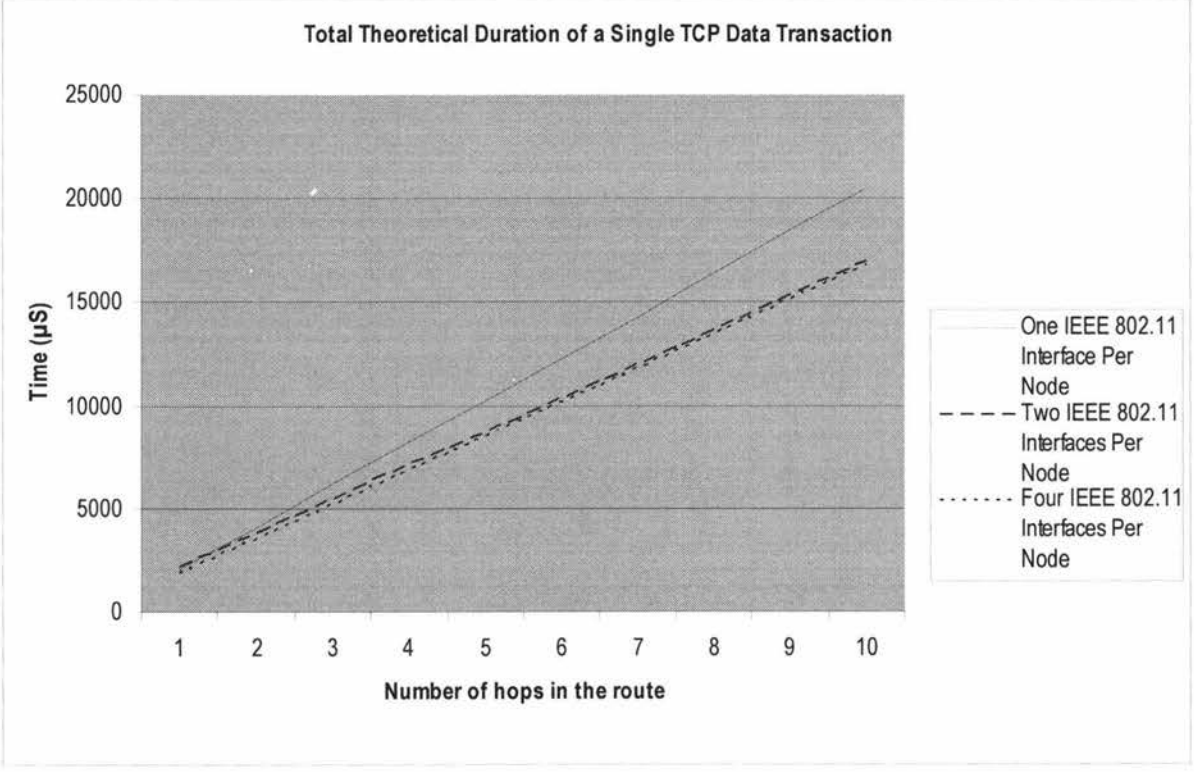


Figure 5.10 Duration of a single TCP data transaction for a range of multiple hop routes and IEEE 802.11 interfaces per node

number of hops and it would seem that from these figures, using two IEEE 802.11 interfaces per node seems to be the best cost vs throughput rate performance solution. As explained in the next section though, the TCP connection model used in the scenario to calculate these throughput rates was kept

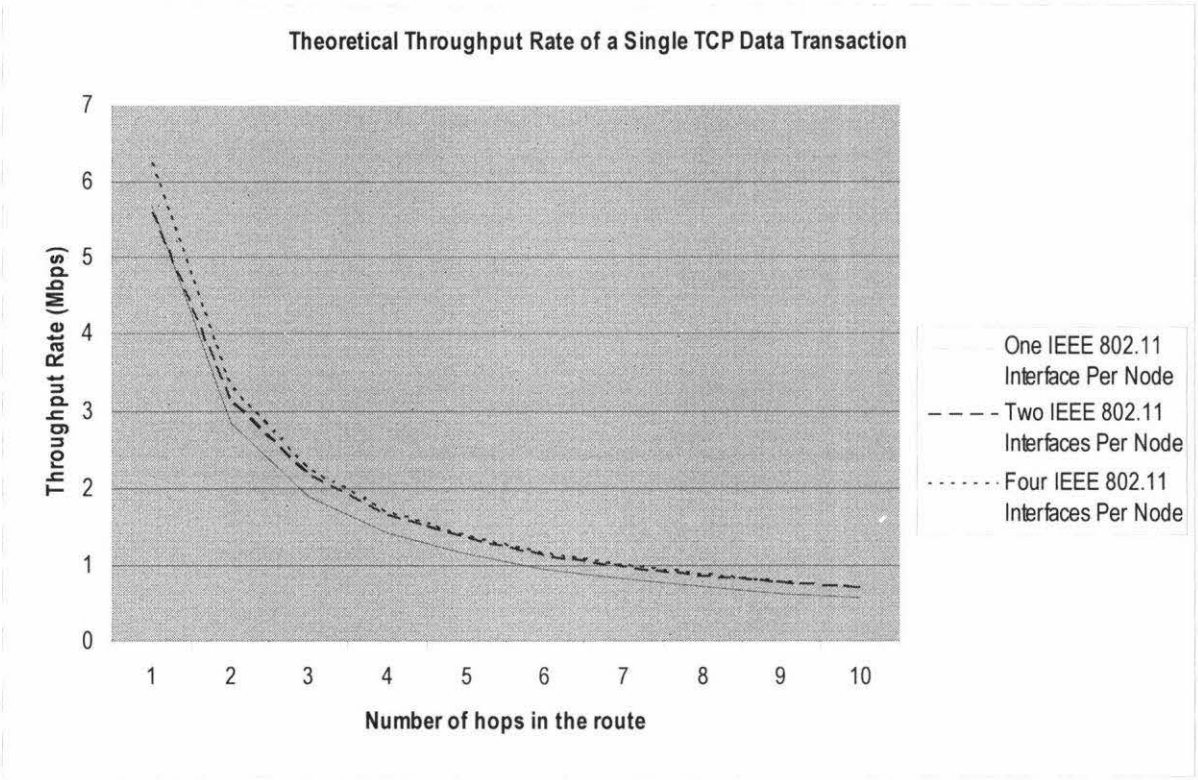


Figure 5.11 Throughput rate of a TCP data transaction for a range of multiple hop routes and IEEE 802.11 interfaces per node

very simple in order to simplify the calculations and highlight the effects of the CSMA/CA algorithm. In reality, the TCP connection model used in Linux is far more complex and would much more likely benefit from using four wireless interfaces per node.

5.2.2 Multiple Wireless Interfaces and the Transport Control Protocol

A major assumption used in the previous example scenario was that a simple TCP connection model was used. With this simple model, for every TCP segment (the TCP data packet) transmitted, the source node would wait until a TCP acknowledgment arrived back before transmitting the next TCP segment. In reality, TCP uses a much more complex data flow control process.

The TCP implemented in Linux is defined by RFC 793 [17], RFC 1122 [18] and RFC 2001 [19]. TCP uses data flow control in order to use the network more efficiently. Part of the flow control process is the use of buffers; a sender buffer which is located at the source node, and a receiving buffer that is located at the destination node. The receiver buffer is more important in terms of flow control. The buffers are simply space in the node's memory which can be used by TCP to store segments for

different purposes, such as: waiting to be transmitted in the case of the source node, or waiting to be processed by the receiving application as in the case of the destination node.

TCP also uses acknowledgments in order to help maintain flow control. Every TCP segment transmitted is assigned a sequence number by the sender, and the receiver can then acknowledge which segments it has received by transmitting back an acknowledgment specifying the sequence number of the *next* segment it is expecting to receive. This is basically the receiver saying that it has received every segment up to the one with the sequence number specified in the acknowledgment. For example, a receiver transmitting back an acknowledgment containing the segment sequence number five, means that is has received segments zero to four.

The receiver can also specify a new window size within the acknowledgment that is sent back. The window size is how much space is left in the TCP buffer before it is full. Since a receiving application may not be able to constantly keep the receiving TCP buffer empty, it is important the the receiver can inform the sender of the buffer status to prevent a buffer overflow and data being lost. As well as knowing the receiver window size, the sender also has a process which calculates what is known as the congestion window size, which is a calculation of how many segments the network can take before congestion occurs.

The minimum out of the receiver window and the congestion window specifies how many segments the sender can transmit across the network to avoid network congestion and also prevent a buffer overflow at the receiver end. More importantly though in terms of this research, the minimum of these two window sizes specify how many segments the sender can transmit before it must wait for an acknowledge back from the receiver. This means that the sender doesn't actually wait for an acknowledgment for every segment before transmitting the next segment, but instead the sender can transmit as many segments as specified by the minimum of either the congestion or receiver window before it must stop and wait for an acknowledgment.

The time when the receiver transmits acknowledgments though is independent of the rate at which the sender transmits segments, but instead is triggered by the occurrence of a number of events defined in RFC 813 [20]. The main events that result in acknowledgments being transmitted is a pause in the TCP segment stream from the sender (often indicating the sender is waiting for an acknowledgment), but more commonly, a significant change in the receiver window size caused by the receiving

application being able to remove the TCP segments from the buffer memory to process. While a TCP segment stream is being transmitted, the receiver window size will grow and shrink in proportion to how fast the receiving application removes the TCP segments from the buffer. As a result, for each significant change in the window size, the receiver will transmit an acknowledgment which will also be used to inform the sender of the new receiver window size. The sender, since it has now received an acknowledgment, can adjust both windows sizes, the receiver window from the information in the acknowledgment, and the congestion window due to the event of receiving an acknowledgment. The sender can then also decrease it's record of how many segments it has transmitted since the last acknowledgment was received. This process ultimately results in a continuous flow of TCP segments and the full duplex behavior of TCP.

The critical difference between the basic TCP connection model specified in the scenario in *Section 5.1,* and the actual TCP model used in Linux, is that TCP can operate with full duplex capabilities, whereas the TCP model used in the example scenario was essentially only operating with half duplex capabilities. Using four wireless interfaces per node is the only configuration with full duplex capabilities but the performance benefits from this wasn't reflected in the theoretical throughput calculations since only a half duplex TCP model was being used. Being full duplex means the TCP acknowledgments can be transmitted from the receiver to the source while the TCP segment stream is taking place, reducing the overall time of the transaction. Also, for multiple hop routes, as soon as one segment has left the first hop and is being transmitted across the second hop, then the next segment can be transmitted across the first hop without interfering. This also reduces the overall transaction time. Since each hop operates on a separate channel preventing interference, this type of behavior could also occur when using two wireless interfaces per node. This behavior could occur when using just a single interface per node although it would only work once two different hops are out of radio range of each other, as all hops are on the same channel and interfere with each other.

The congestion window feature within TCP that Linux implements, is part of the *slow start process.* The slow start process is when initially the sender only transmits one TCP segment, waits for an acknowledgment, transmits two TCP segments, waits for both to be acknowledged and then transmits four segments and so on. This process of doubling the congestion window, which results in doubling the number of segments transmitted each time, is continued until a certain threshold is reached where the number of segments transmitted each time still increases but only by one segment each time. This continues until the number of segments being transmitted at a time equals the receiver window size. If

the sender fails to receive an acknowledgment for a segment before a timeout period, then the congestion window is set back to one and the slow start process begins again.

It would be difficult to calculate accurate theoretical throughput rates using an actual TCP throughput model with multiple wireless interfaces due to the complex nature of the TCP flow control algorithm and random occurrences like collisions on the network resulting in repeated transmissions. This section outlined the general flow control process used by TCP in Linux and how it could affect the throughput performance when using a full duplex configuration such as four wireless interfaces per node.

6 Implementation

The implementation stage of this research focused on modifying the original hello message based AODV implementation (AODV-HM) to create a data link layer feedback based AODV implementation (AODV-LL). The original AODV-HM implementation used in this research was Kernel AODV v2.2 by L Klein-Berndt from NIST, USA [5]. Currently, a number of AODV implementations are available, but Kernel AODV was used due to it being the implementation that we had the most experience with and it being Linux based. The AODV-LL implementation was created by significantly modifying Kernel AODV rather then creating a completely new AODV implementation from scratch. The basic structure and operation of Kernel AODV when creating the AODV-LL implementation didn't change apart from the key areas listed in this section.

Implementation was carried out in four stages:

1. The IEEE 802.11 device driver was modified to make use of the transmit retry limit and provide feedback to AODV.
2. The original AODV-HM implementation was modified to remove the use of hello messages and instead use information provided by the IEEE 802.11 device driver on the data link layer resulting in the new AODV-LL implementation.
3. The AODV-LL implementation was designed so that it was able to operate correctly with multiple interfaces. The AODV-LL implementation could operate correctly without any changes when using two wireless interfaces, but had to be modified to work with four interfaces correctly.
4. The AODV-LL implementation was then designed to be backwards compatible with the original AODV-HM implementation. It was decided this would be an important feature if the AODV-LL implementation was used in a real world scenario.

6.1 Basic Operation of the AODV-HM Implementation

Overall, the basic structure of the AODV-LL implementation remained unchanged from the AODV-HM implementation. The following describes the basic operation of Kernel AODV, the original AODV-HM implementation used in this research.

Kernel AODV is run as a kernel module rather then being compiled directly into the Linux kernel. The user inserts the Kernel AODV module into the kernel to start it and removes the module to stop it. User configured parameters can be passed in when the module is inserted. When the module is inserted by the user, Kernel AODV starts up by performing several initialization tasks. These tasks include initializing any processes, initializing tables or lists such as the AODV route table and initializing all the network interfaces that are being used, by setting up a route for each one in the AODV route table and the kernel route table. Another important initialization task is registering the AODV module with the different Netfilter hooks in the network stack. This allows the Netfilter hooks to know which network packets should be passed to the AODV module to process. After the initialization tasks, the main AODV process is then started.

The main AODV thread or process sits in memory "asleep" until it it is "kicked" or interrupted by another AODV process and is given a "task" to do. A task is a specific data structure defined by Kernel AODV. The structure contains what type of task it is, for example a TASK_RREQ used for processing RREQ messages, all the necessary data needed to complete the task, like the received AODV message, and also some other extra information that is required. The main AODV thread takes the task, determines which AODV process the task is intended for, sends the task to this process and then goes back to sleep. The AODV process that does the "kicking" is the *task_queue* process. The *task_queue* process is given tasks to queue up from either the the *packet_in* process or the *timer_queue* process. The *packet_in* process gets network packets passed to it by the Netfilters pre-routing hook which are addressed to the AODV port number. The *packet_in* process checks for the AODV message in the network packet and then passes it to the *task_queue* process to create a task for it. The *timer_queue* process takes care of assigning timers to tasks, as well as sending a task to the *task_queue* process when the timer for a task has expired. Using timers makes periodic tasks such checking the AODV route table for expired routes simple. To do this, when AODV starts up, a timer for the task to check the

route table is put into the timer queue. Later when the timer runs out, the task is passed to the main AODV thread which then prompts the appropriate process to check the route table. Before finishing, the main AODV thread reinserts a timer in the timer queue for the same task and so continues the cycle.

Similar to the *packet_in* process, the *packet_out* process is triggered by every out-going packet that goes out the Netfilters post-routing hook. *Packet_out* sends the destination IP address of each outgoing packet to the *aodv_route* process which compares the address with those already in the AODV routing table. If the address exists, then its lifetime for staying in the AODV route table is increased (since there are packets still going to that address), otherwise if the address is not found in the route table, then the *packet_out* process triggers the RREQ process, which goes about finding a route to the destination. Thus by monitoring the packets going out the post-routing hook, Kernel AODV is able to determine when a route to a destination node is needed.

The *aodv_route* process manages the AODV route table and calls on the *kernel_route* process to modify the actual kernel route table. The AODV route table is only used by the AODV module, while the kernel route table is used by the actual Linux kernel. The *aodv_route* process contains many functions which are used by itself and other processes to create, modify and delete routes in the AODV route table. A similar process in the original Kernel AODV is the *aodv_neighbor* process. It manages the neighboring node table, but as described later, the AODV-LL implementation doesn't use both the *aodv_neighbor* process or the neighboring node table. The *hello* process is used to generate hello messages and calls on the *timer_queue* process to create timers for every neighboring node in the neighboring node table. These timers expire when hello messages are no longer received from the neighboring node associated with the timer. If a neighboring node timer runs out, then that node is deleted from the neighboring node table. The *hello* process was also significantly changed in the AODV-LL implementation.

The *rreq*, *rerr* and *rrep* processes each process their corresponding AODV messages. These three processes along with the *aodv_route* process implement most of the actual AODV networking protocol on their own while the other processes within Kernel AODV exist to make everything work. The remaining processes within Kernel AODV include ones for working with the network devices, inserting AODV messages into the network stack to be transmitted and displaying statistics to the user about the Kernel AODV module.

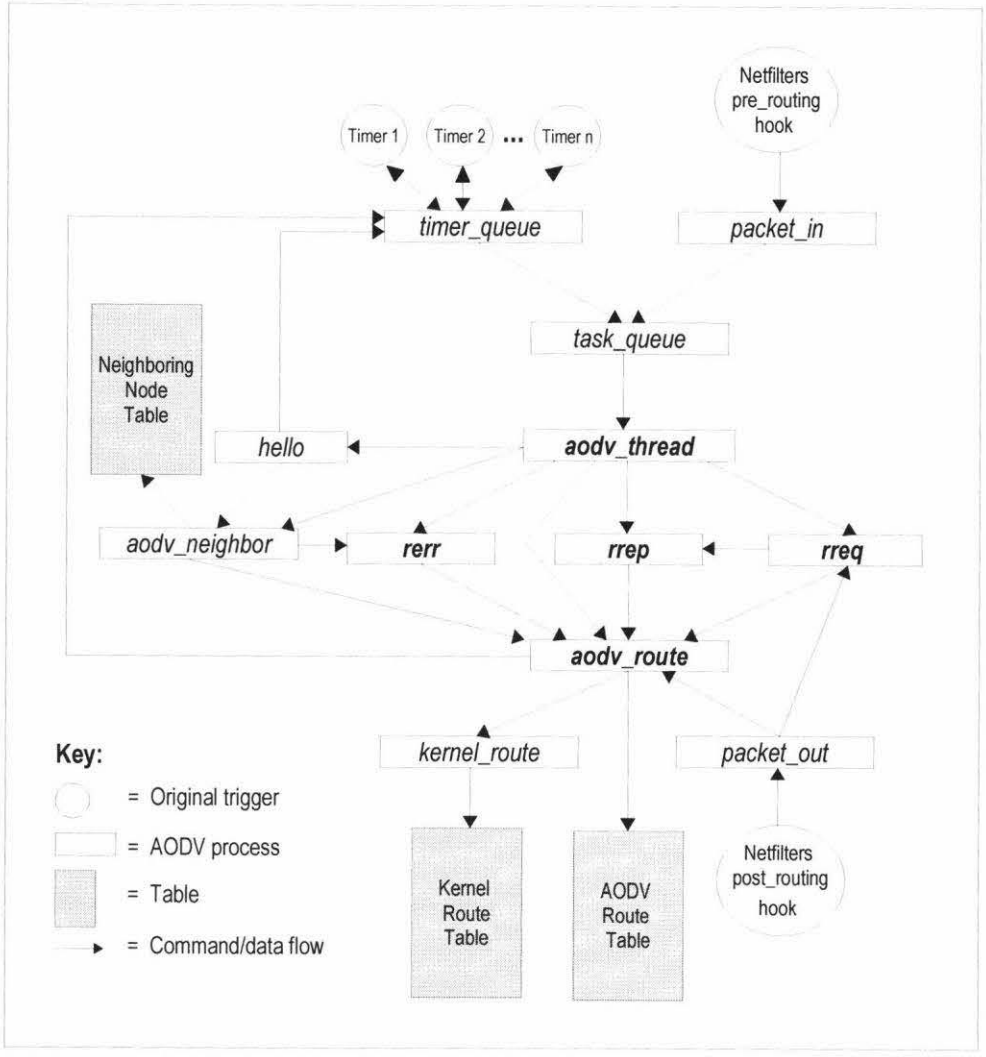


Figure 6.1 The simplified command and data flow of the original hello message based AODV implementation – Kernel AODV v2.2.2

Figure 6.1 shows the simplified data and command flow of Kernel AODV v2.2.2. The general data and command flow of the AODV-LL implementation is the same, except it does not include the *aodv_neighbor* process and the *Neighboring Node t*able.

6.2 Changing the IEEE 802.11 Device Driver

For this research IEEE 802.11b PCI wireless network interfaces were used. The IEEE 802.11b standard is an extension of the IEEE 802.11 standard. Both standards are the same except IEEE 802.11b devices are capable of operating at 11Mbps as apposed to only 2Mbps which the original IEEE 802.11 devices operate at. Two other extensions to the original standard are the IEEE 802.11g and

IEEE 802.11a standards. Both are capable at operating at 54Mbps but the IEEE 802.11g standard operates in the 2.4GHz spectrum like the IEEE 802.11 and IEEE 802.11b standards, while the IEEE 802.11a standard operates in the 5GHz spectrum. For this research, IEEE 802.11b interfaces were used since IEEE 802.11b devices are popular worldwide, and are currently cheaper then both IEEE 802.11a and IEEE 802.11g devices, making them a more feasible option when using multiple interfaces per node.

A total of eight IEEE 802.11b PCI wireless network interfaces were used; three using the the ADM8211 MAC chipset from Admtek and five using the RTL8180 MAC chipset from Realtek. Both Admtek and Realtek are companies based in Taiwan which are specialized in manufacturing networking equipment. The ADM8211 chipset based interfaces have open source drivers available for the Linux operating system which meant they could be modified to provide data link layer feedback using the transmit retry limit. The RTL8180 chipset based interfaces did not have any open source driver software available and therefore could not be configured to provide access to the transmit retry limit. The data sheet for the RTL8180 does specify status registers that monitor the transmit retry limit and therefore the RTL8180 chipset could be used in the same way as the ADM8211 chipset if open source drivers were available.

6.2.1 Changing the ADM8211 Chipset Driver to use the Transmit Retry Limit

The first part of modifying the ADM8211 chipset driver was to enable it to be able to detect when the transmit retry limit had been exceeded. The ADM8211 chipset created an interrupt whenever an error occurred and the interrupt status registers could be checked to identify which error generated the interrupt. Included in the events that could generate an interrupt was the event of the transmit retry limit being reached. Part of the ADM8211 driver was a section which was processed if an error interrupt had been generated. Within this section of the ADM8211 driver, code was added which checked the transmit retry limit status register which effectively enabled the driver to detect if the transmit retry limit had been exceeded. If the value of the transmit retry limit status register equaled one, then it meant the transmit retry limit had been exceeded and appropriate action was taken.

The second part of modifying the ADM8211 chipset driver was to enable it to be able to notify the AODV software of the broken link when it was detected that the transmit retry limit had been exceeded. It was found the best method to do this was for the ADM8211 chipset driver to create an AODV RERR

packet, encapsulated within a DIX Ethernet frame with all the necessary details included in it, and then have it sent up the network stack for AODV to process. By using data structures and systems already in place meant there was no need to modify the kernel network stack as the kernel could simply treat the AODV RERR from the chipset driver as a normal Ethernet frame. This approach also helped satisfy the OSI network model, by ensuring the different layers of the model only communicate through proper interfaces.

The AODV RERR packet and DIX Ethernet frame generated by the ADM8211 driver were configured so that they appeared to the kernel stack as being transmitted by the node at the other end of the broken link. This is obviously impossible, but causes no problems for the network stack, and less processing for the AODV-LL implementation. Most of the information needed to create a DIX Ethernet frame with a AODV RERR packet inside was obtained from the headers of the packet which had just failed transmission and had triggered the transmit retry limit interrupt due to the link being broken. The only information that could not obtained from the packet which had failed to transmit, was the sequence number required for the AODV RERR packet. AODV creates and manages sequence numbers for each route, and when hello messages are used, AODV itself detects when a route is broken and so can use the corresponding sequence numbers from it's own route tables to put in a RERR packet. The sequence numbers in the AODV RERR packet are important since it means nodes that receive the RERR can determine if the RERR is recent information or simply a stale RERR that it has already processed and is still being forwarded around the network. The IEEE 802.11 device driver had no easy method of accessing AODV route tables. Therefore to overcome this problem, the sequence number section of the AODV RERR was left blank and instead a one bit flag in the reserved section of the AODV RERR header was set to one. This meant AODV-LL could check for this flag and determine if the AODV RERR had been created by the IEEE 802.11 device driver or instead was a RERR that had actually been transmitted from another node. If the RERR was from the IEEE 802.11 device driver, then AODV knew to find the corresponding sequence number from the route table for the broken link, add it to the RERR, update it's own route tables and then send the RERR back down the network stack to be transmitted to the surrounding nodes. *Figure 6.2* shows the process used to inform AODV of a broken link after the transmit retry limit has been exceeded.

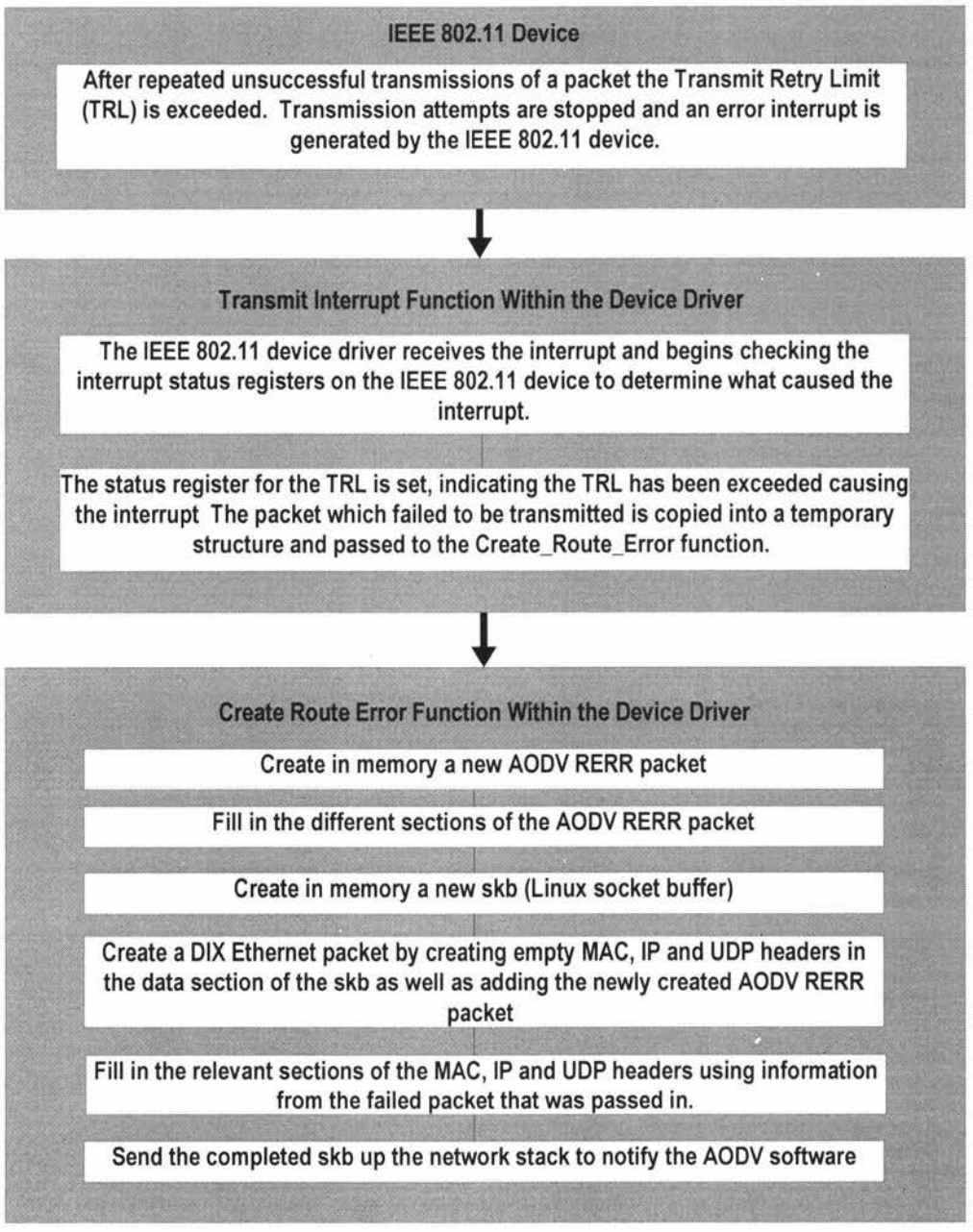


Figure 6.2 The process from the transmit retry limit being exceeded to AODV being notified

6.3 Changing the AODV-HM Implementation to use Data Link Layer Feedback

There were two main areas in the original AODV-HM software that were modified in order to create the AODV-LL implementation. *Table 6.1* shows the differences in these two areas.

Table 6.1 Significant areas of the original AODV-HM implementation that were changed or created to create the AODV-LL implementation

Created/Modified Area	1. Detection of a broken route	2. Management of the AODV neighboring node table
AODV-HM implementation	Checks periodically that hello messages are being received from a neighboring node to determine if that neighboring node is alive and that routes through the node are not broken.	Neighboring nodes are created or updated in the neighboring node table by the information in the hello messages received from neighboring nodes.
AODV-LL implementation	Doesn't check in any way if neighboring nodes are alive. Instead a special AODV RERR message from the wireless interface driver (the data link layer) informs AODV of the broken link.	Hello messages aren't used allowing the neighboring node table to be removed completely resulting in only the AODV routing table needing to be managed.

6.3.1 Detection of a Broken Route

The first area of the original AODV-HM implementation that had to be changed to create the AODV-LL implementation was to remove hello messages and instead give the AODV-LL implementation the ability to identify the special AODV RERR packets from the data link layer and then take the appropriate action.

In the original AODV-HM implementation, hello messages were generated through a timer task. A *task_hello* timer was created when AODV started up, and set to expire in one second which is the default hello interval rate. When the *task_hello* timer ran out, it created an interrupt, prompting the *timer_queue* process to insert the *task_hello* task in the *task_queue* process. This in turn interrupted the main AODV thread which sent the task off to the *hello* process, resulting in a hello message being created and sent to the network stack to be transmitted. After the hello message had been generated and transmitted, the *hello* process created a new *task_hello* timer set to the hello interval rate to ensure the hello messages continued. In the AODV-LL implementation, to stop the hello messages from being transmitted, the initial *task_hello* timer which was created when AODV started up was disabled and thus stopping the hello message cycle.

The next stage was to enable the AODV-LL implementation to be able to identify the RERR message sent to it from the data link layer. To do this, the *packet_in* process in the AODV-LL implementation checks the first bit in the reserved section of any AODV RERR message received. If the bit equals zero then the RERR is treated as a normal RERR from a another node, otherwise if the bit is equals one

then its treated as a RERR from the data link layer. For RERRs from the data link layer, *packet_in* sends the RERR message off to the *task_queue* process but tells the *task_queue* process that it is a TASK_NEIGHBOR message type rather then the normal RERR message type. Doing this means when the *task_queue* process receives the RERR it knows its from the data link layer.

Normally a RERR from another node results in AODV deleting any routes associated with the broken node or link, and then deciding whether it needs to forward on the RERR to downstream nodes in any affected routes. With the RERR from the data link layer, the sequence number for the affected route is missing since the data link layer has no easy way of obtaining this information. The RERR from the data link layer is treated simply as an indicator of a broken route, providing the only necessary information – the IP address of the inactive or broken node. When *task_queue* process receives the data link layer RERR it extracts the IP address of the node causing the broken link and then creates a new route removal task with the broken node IP address as the target IP. The route removal task is sent off to the main AODV thread process which passes it on to the *aodv_route* process. The *aodv_route* process expires any routes that pass through the inactive node (the expired routes are then removed later) and then generates a proper AODV RERR containing all the necessary information of the broken route including the sequence number and sends it off to the network stack to be transmitted.

The process used to remove routes using the RERR from the data link layer is similar to the process used with hello messages in the original AODV-HM implementation. Whenever a hello message was received from a neighboring node, a timer set to twice the length of the hello interval was created. This timer, when it ran out, would trigger the process which would remove the inactive node and any associated routes from the route table and then generate a RERR message to be transmitted. Every time a hello message was received from a neighboring node, its corresponding "removal" timer was reset, meaning that two hello messages from a node had to be missed before that node was considered inactive and the route broken. Using the data link layer technique meant detecting broken routes was almost instant, and in the two seconds needed for hello messages to detect a broken route, the AODV-LL implementation could have enough time to have an alternative route discovered and operational. *Figure 6.3* shows the command and data flow through both the AODV-HM and AODV-LL implementations as the result of detecting a broken route. The AODV-HM implementation diagram shows the trigger being the "removal" timer for a node from which no hello messages have been received for the period equal to twice the hello interval, while the AODV-LL implementation shows the trigger being the event of receiving a RERR from the data link layer.

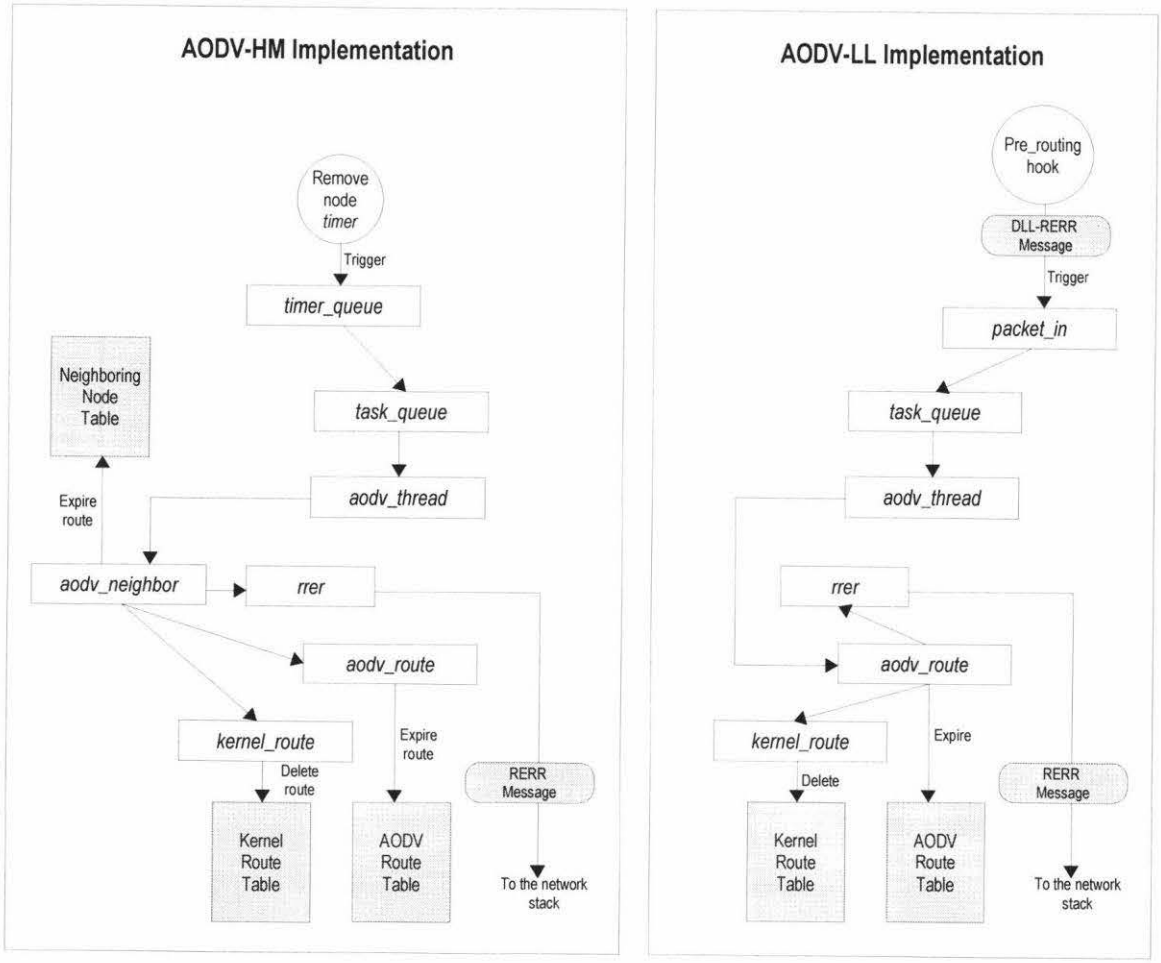


Figure 6.3 Simplified data and command flow through the AODV-HM implementation and the AODV-LL implementation as the result of detecting a broken route

6.3.2 Management of the Neighboring Node Table

As shown in *Figure 6.3*, the AODV-LL implementation no longer uses the *aodv_neighbor* process or the neighboring node table when processing a broken route. In fact, as a result of no longer needing hello messages, there was no need at all to maintain a neighboring node table which in turn removed the need for almost all the functions within the *aodv_neighbor* process. With the original AODV-HM implementation, the only way a broken route could be detected was through listening for the periodic hello message transmissions from neighboring nodes. Because of this, keeping track of neighboring nodes was important, resulting in a neighboring node table being used, along with several functions to access and manipulate the table that made up the *aodv_neighbor* process. Neighboring nodes were added to the table when a hello message was received from them (regardless of whether the node was used in a route), and were deleted from the table (along with any associated routes) when hello

messages stopped being received from them. Whenever an AODV message was received by the AODV-HM implementation, it firstly checked that the message was from a node transmitting hello messages by checking if it existed the neighboring node table. This was necessary since if a route was set up through a neighboring node, it had to transmit hello messages for AODV-HM to be able to monitor the status the route. Any AODV message received from a node not transmitting hello messages was discarded.

When the AODV-LL implementation was created, the neighboring node table and *aodv_neighbor* process was removed although two extra functions had to be created for the *aodv_route* process to make up for some functionality that the *aodv_neighbor* process originally provided. The first extra function created for the *aodv_route* process was for adding routes to neighboring nodes. When a route is added to the kernel route table, it is done by specifying to the kernel the IP address of the distant destination node, along with specifying the IP address of the gateway, or the neighboring node that the kernel must send any network packets to in order for them to be forwarded on down the route to the destination node. If the kernel route does not already have a separate route to the gateway or neighboring node, then it won't allow routes with the IP of an "unknown" neighboring node as the gateway being added. With the AODV-HM implementation, routes to neighboring nodes were added to the kernel route table as soon a hello message was received from the neighboring node and therefore separate routes to the neighboring nodes were always in the route table before any extra routes were added where the neighboring node was the gateway. Because hello messages were removed, a separate function was added to the *aodv_route* process which handled adding neighboring node routes to the kernel route table. This function simply checked that the IP address of the neighboring node that the AODV message came through was already in the route tables. If the neighboring node wasn't in the route table, then a route was created for it and was added to both the AODV and kernel route tables before AODV carried on processing the AODV message. The second function that was added to the *aodv_route* process was removing neighboring node routes. Previously this function was within the *aodv_neighbor* process and was triggered when hello messages stopped being received from a neighbor. The new route removal function was triggered when a RERR was received from the data link layer, resulting in the neighboring node route being expired in the AODV route table and removed from the kernel route table. *Figure 6.4* shows the general data and command flow in the AODV-LL implementation after removing the neighboring node table and *aodv_neighbor* process. The reason for the *hello* process still being present is for the purpose of making AODV-LL compatible with hello message based AODV implementations and is explained in more detail later in the thesis.

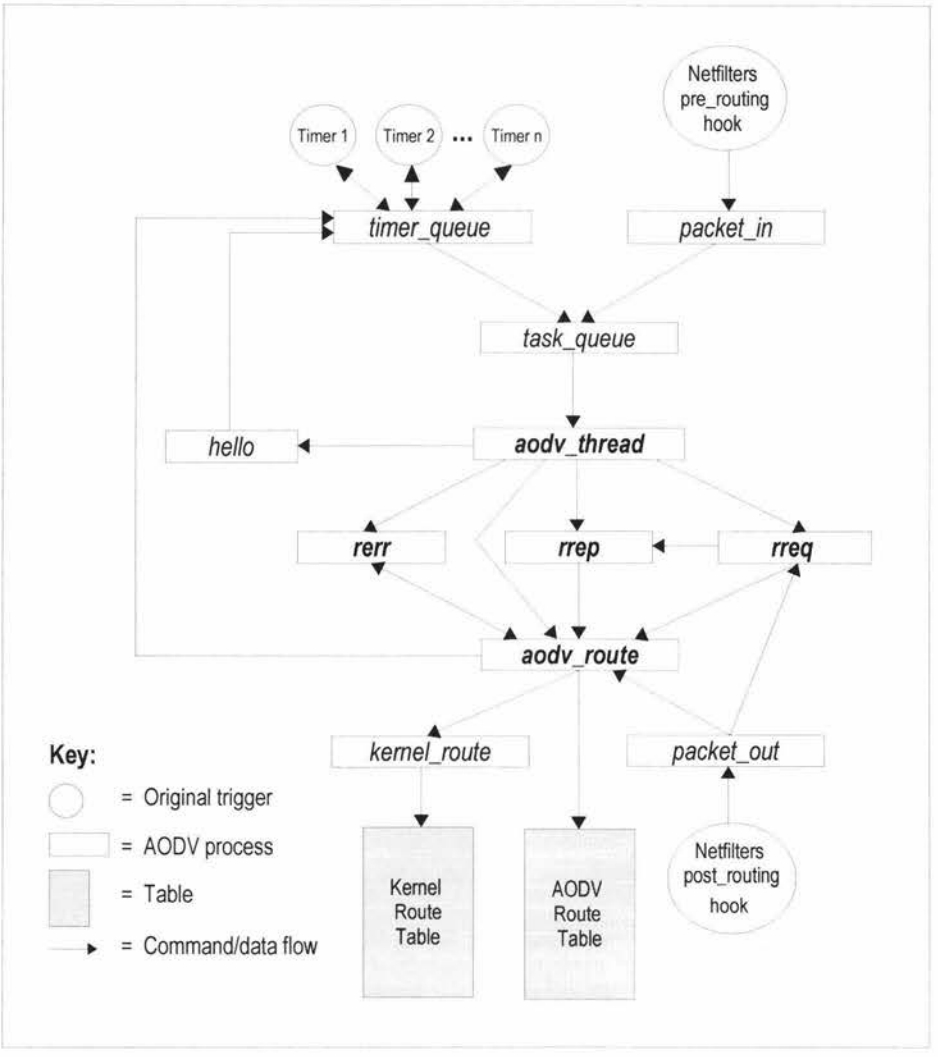


Figure 6.4 The simplified data and command flow of the AODV-LL implementation

6.4 Making the AODV-LL Implementation Compatible with Multiple Wireless Interfaces

The original AODV-HM implementation, Kernel AODV, is capable of assigning routes to go through specific interfaces, as it allows wireless nodes to act as gateways to other networks such as Ethernet networks. When an AODV message arrived from a neighboring node on an interface, Kernel AODV assigned all future routes that went through that neighboring node to go out through the interface that the neighboring node was detected on. As a result, the AODV-LL implementation required no modification to be compatible with two interfaces per node because of this feature.

AODV-LL needed to be modified though to be compatible with four interfaces per node. Using four

wireless interfaces per node can be understood easier when they are viewed as two pairs of interfaces. One pair handles the "downstream" traffic while the other pair handles the "upstream" traffic. But then within each pair (as far as the network layer and above are concerned), one interface is only used for incoming traffic and the other interface is only used for outgoing traffic and so overall creating four independent communication channels. Because of this, for AODV-LL to work with four interfaces, it had to be modified so that when a connection to a neighboring node was detected, then the interface it was detected on was considered the incoming interface, while the other interface within the pair was considered the outgoing interface. This way each route was configured to go out through the outgoing interface within the correct pair of interfaces.

Broadcasts are always transmitted out each network interface by default, and although an AODV broadcast such as a RREQ or RERR would only need to be transmitted out one interface per pair to achieve it's purpose, it was simpler to leave it unmodified where a broadcast packet went to all interfaces. Because of this, since both the incoming and outgoing interfaces will receive broadcast messages such as RREQs, which are used to set up neighboring node routes, either interface could be the incoming or outgoing interface, but it was decided that the first interface to have the RREQ or any other AODV message processed by AODV, would be considered the incoming interface.

To implement this, AODV-LL was modified so that when it started, the user specified the network interface pairs. In the initialization process, AODV-LL then linked each interface to its other pair interface. After this the only other major section that needed to be modified was in the routing section where neighboring node routes was created. When a neighboring node is detected, such as by receiving an AODV message from a neighboring node not already listed the AODV route table, then the network interface that the message was received from is considered the incoming interface, and the other network interface within the pair is considered the outgoing interface. A route to the neighboring node is then set up through the outgoing interface, and as a result all routes that pass through the neighboring node also go through the outgoing interface.

6.5 Making the AODV-LL Implementation Compatible with the AODV-HM Implementation

When the AODV-LL implementation was created, it was decided that, if possible, it would be designed

so that it would be able to function with a hello message based AODV implementation so that it would be possible to operate an AODV-LL node in a real world scenario where hello message based AODV nodes also operate. The following outlines necessary features that the AODV-LL implementation would need, in order to be compatible with the AODV-HM implementation, but at the same time keep hello messages to a minimum. The AODV-LL implementation would need to :

- be able to detect when a hello message based AODV node is operating within its radio range and when the hello message based AODV node has left its radio range
- transmit hello messages when a hello message based AODV node is detected but stop transmitting as soon as the hello message based AODV node is no longer detected
- be able to determine the difference between a hello message from an AODV-LL node and an AODV-HM node, so that an AODV-LL node doesn't misinterpret another AODV-LL node transmitting hello messages as an AODV-HM node, resulting in hello messages being transmitted by every node across the whole network

To detect a hello message based AODV node is simple – AODV-HM based implementations transmit hello messages whereas AODV-LL implementations don't. If an AODV-LL based node detected hello messages then it could assume that there is at least one AODV-HM node within it's maximum network range. As soon as no hello messages are detected, an AODV-LL node could assume that there are no AODV-HM nodes left within it's maximum network range. Using this approach, the AODV-LL implementation was designed so that as soon as a hello message from an AODV-HM based node was detected, the AODV-LL implementation would start broadcasting hello messages at the rate of one per second. The hello message transmitted by the AODV-LL implementation has the first bit of the reserved section in the hello message data structure set to equal one. This bit is not checked by AODV-HM implementations, so it can be used by AODV-LL implementations to determine if a hello message is either from an AODV-HM node or an AODV-LL node resulting in any hello messages from AODV-LL nodes being silently discarded and no action being taken. As so as the AODV-LL implementation detects no more hello messages are being received from AODV-HM nodes, it stops broadcasting hello messages.

To implement this in the AODV-LL implementation, a combination of timer tasks are used. As soon as the AODV-LL implementation receives a genuine hello message, two timer tasks are created. The first task is set to the hello message interval period (one second) and results in the *hello* process being called and a hello message being transmitted. After the hello message is transmitted the *hello* process

checks a flag which specifies if AODV-HM nodes are still present. If true then the hello message timer task is set running again so that hello messages will still be transmitted. If the flag is false then the hello message task is deleted and the hello messages stop being transmitted. The second timer task that is created when a hello message is received is for stopping the hello messages. It's timer is set to twice the hello message interval (two seconds), but is reset back to zero every time a hello message is received from another node. While hello messages are being received, the timer never expires. But if no hello messages are received from another node for at least two hello message interval periods, then the timer expires, resulting in the *hello* process being called which results in the flag which specifies if AODV-HM nodes are present, being set to zero. This will result in the hello message timer task being deleted and hello messages will stop being transmitted. To determine if a hello message is from an AODV-LL node, the *packet_in* process checks the hello message when it arrives for the flag which if set means the hello message is from an AODV-LL node otherwise the hello message is from an AODV-HM node. Hello messages from AODV-LL nodes are silently discarded and no action is taken.

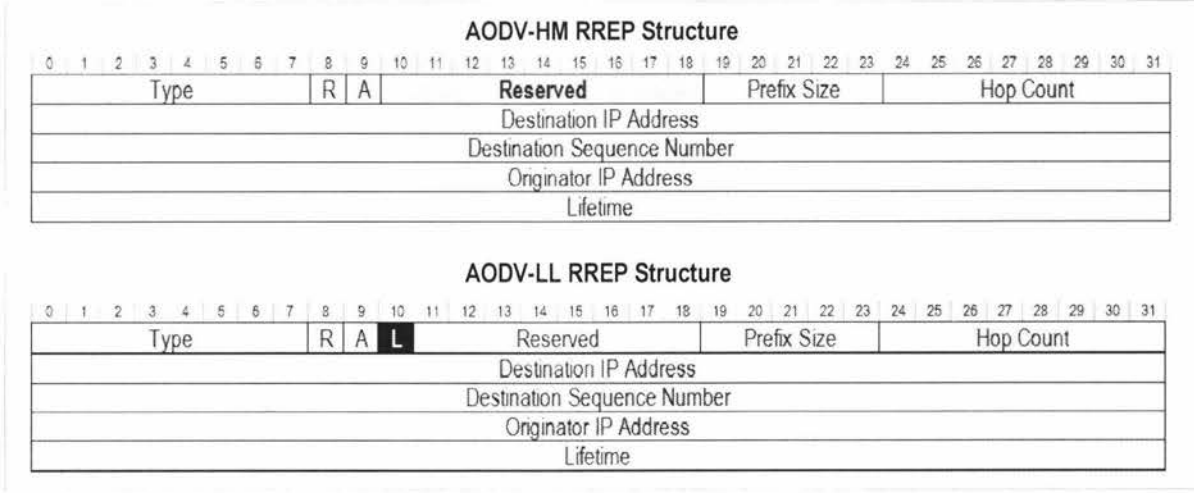


Figure 6.5 The different AODV-HM and AODV-LL RREP structures

To create a flag which indicated a hello message was from an AODV-LL based node, the original hello message data structure had to be changed. The hello message structure, as defined by the AODV protocol, is actually a RREP structure. To determine if a RREP message is a hello message, the AODV implementation checks if the IP address in the destination address field is the same as the IP address in the source address field *Figure 6.5* shows the original AODV-HM RREP structure and the new AODV-LL RREP structure. In the header of the AODV-HM RREP structure, there are ten bits which are defined as the *Reserved* section. The AODV-LL RREP structure was changed so that the

first bit (bit ten) of the old reserved section is now a flag, termed "L" since it is used to indicate that the hello message is from an AODV-LL based node. By checking if the "L" flag equals one, the AODV-LL implementation can assume that the hello message is from an AODV-LL node, otherwise if the "L" flag equals zero (since by default AODV-HM implementations set the reserved section to zero, which for AODV-HM implementations includes the "L" flag), then the AODV-LL implementation can assume it is a genuine hello message from an AODV-HM based node. AODV-HM based implementations do not check the *Reserved* section of the RREP structure (which for them includes the "L" flag), so therefore they will process all hello messages from an AODV-LL node as normal.

7 Testing Methodology

For this research the tests were designed to not only test the performance of the AODV-LL implementation and the the multiple wireless interface configuration, but they were also designed to test the functionality of the AODV-LL implementation; the multiple wireless interface configuration; and the the data link layer route error reporting mechanism in order to show that they all operate correctly.

The functionality tests are far from complete in terms of testing with all possible node and transport protocol configurations. They were designed to show that the AODV-LL implementation functions correctly with the data link layer route error reporting mechanism and the multiple wireless interface configuration in a simple TCP based network. Also included in the functionality tests, were tests to show that the AODV-LL implementation was compatible with the old AODV-HM implementation as well as tests to find the optimal transmit retry limit that should be used by the data link layer route error mechanism to ensure the correct detection of broken routes.

The performance tests are also far from complete in terms of testing with all possible performance metrics and with all possible node and transport protocol configurations. Instead this research has focused on improving throughput at the application layer when using the IP routing and TCP transport protocols. These tests were based on the single performance metric, data throughput, which was measured at the application or user layer. All the tests were performed within the IIST Networking Laboratory at Massey University, New Zealand (see *Figure 7.1*).

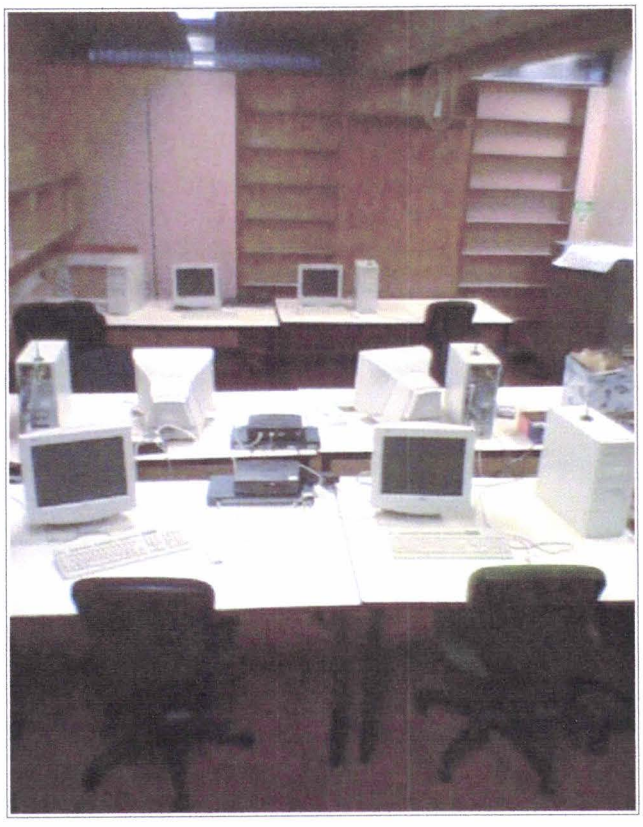


Figure 7.1 IIST Networking Laboratory Massey University

The major software used in the testing was:

- AODV-LL. The data link layer feedback based AODV implementation designed and created as part of this research

- Kernel AODV 2.1. This was the hello message based AODV implementation used. The software was completely unmodified and was obtained from the official NIST Kernel AODV website [5]. Kernel AODV version 2.1 was chosen over version 2.2 since version 2.1 operated better with multiple wireless network interfaces.

- The Linux kernel v 2.4. The Red Hat 9 distribution of the Linux kernel was used as the base operating system for all the tests. Each computer in all the tests had identical makes of the operating system on them.

- Netperf. This is a Linux based network performance testing software. It operates by the

destination running a Netperf server while the source runs the Netperf client. For these tests we only used the TCP transport protocol in the throughput tests. After the Netperf client had connected to the Netperf server running on the destination, the Netperf client would then proceed to transmit a TCP stream of data to the Netperf server, and display the throughput result. Options such as the size of the network buffers on the source and destination can be changed, along with the size of the data packet and the length of the throughput. For the performance based tests, each throughput tests ran for sixty seconds, with 8Mb of data being repeatedly transmitted until the sixty seconds were up. The sender and receiver network (socket) buffers sizes were set to 32Kb. For each performance test, the Netperf settings stayed constant.

- Ethereal. Ethereal is capable of monitoring and recording all incoming and outgoing network packets on the node's network interfaces. This software was used primarily in the functionality tests as it was capable of decoding AODV packets, making it an effective tool to identify when AODV and the route error detection mechanism functioned incorrectly. Ethereal was also capable of identifying the AODV RERR message being sent from the wireless interface driver on the data link layer to AODV on the network layer as it was sent up through the normal Linux network stack being monitored by Ethereal. Ethereal could identify the packet was for AODV by the packet UDP port number, but was unable to identify that the packet was a RERR packet. This was due to the fact that when the wireless interface driver software generated the RERR packet, the UDP checksum wasn't added on to the end of the RERR packet, as it was not needed. Therefore the packet size is smaller then expected by Ethereal and it checks the wrong bit in the RERR packet to determine what type of AODV message it is. As a result, the RERR message from the wireless interface driver software appears in the Ethereal packet capture list as an AODV packet but with out the AODV type being defined. This is shown more clearly in the Ethereal screenshots in *Section 8.0*.

- IP Tables. The Linux IP Tables filter/firewall software was used to artificially create hops between nodes when needed. Since all the tests were performed within a laboratory, it was not possible to separate the nodes far enough apart so that each node was only within the radio range of the next node one hop away in the route. Therefore the distance between the nodes was artificially created by setting up MAC address filters on each node using the

IP Tables software. This way a node could only accept frames with the MAC address of the nodes one hop away either upstream or downstream in the route. When using multiple wireless network interfaces, MAC address filtering wasn't needed, since each wireless interface pair could be configured to operate on a different frequency.

- General networking tools used for the functionality tests include the PING command which sends an ICMP echo request to a destination which responds with an ICMP echo reply.

The following outlines the hardware used in the testing:

- Micronet SP906B wireless interfaces. For all the tests a total of eight IEEE 802.11b compliant wireless network interface cards were used with a maximum transmit rate of 11Mbps. In all the tests, all security encryption functions on the cards were turned off, along with fragmentation functions and each card was forced to only operate at 11Mbps. For all the tests, the cards operated in ad-hoc mode (IBSS mode). Initially three Micronet SP906B PCI wireless cards were purchased for the testing. These cards were based on the ADM8211 chipset from Admtek. The ADM8211 chipset had open source Linux based drivers, which were modified to enable the use of the transmit retry limit interrupt. For all the functionality tests the SP906B cards were used. See *Figure 7.2.*

Figure 7.2 Micronet SP906B PCI wireless interface with the ADM8211 chipset

- Micronet SP906BB wireless interfaces. Later in the research another five wireless cards were needed to use with the other three for the performance tests. By this stage the SP906B wireless card was no longer produced and instead five IEEE 802.11 complaint SP906BB PCI cards were purchased. The Micronet SP906BB was the updated version of the SP906B and operated on the Realtek 8180 chipset. The Realtek chipset had Linux based drivers but they were not completely open source and therefore could not be modified to utilize the transmit retry limit. When used with the AODV-LL implementation, this meant broken links could not be detected. For the performance tests though, the focus was comparing throughput between the AODV-HM and AODV-LL implementations along using multiple wireless interfaces, as opposed to testing the route error detection capabilities like with the functionality tests. Because of this, the Realtek chipset cards could still be used in the performance tests since all the nodes in the routes were static and broken links never occurred. See *Figure 7.3.*

Figure 7.3 Micronet SP906BB wireless interface with the RTL8180L chipset

- Computers. For all the tests standard desktop PCs were used. For the functionality tests three identical PCs were used, consisting of 500MHz Intel Pentium 3 processors with 256MB of RAM. These three PCs were also used for the for the performance tests along five other PCs consisting of 2GHz Intel Pentium four processors with 256MB of RAM.

Initially there was concern that the faster PCs could affect the throughput performance, but testing done as part of the functionality tests showed that once the route was set up by AODV, there was very little processor usage while the actual throughput tests were taking place, even on the slower PCs, concluding that the effect on throughput caused by the PC hardware was negligible compared to the effect caused by the header overheads from the IEEE 802.11 protocol.

7.1 Functionality Test Methodology

The functionality tests were divided into three groups of tests. The first was designed to find the optimal transmit retry limit. The second group of tests tested the operation of the AODV-LL implementation to ensure that it conformed to the AODV protocol and more importantly, that the data link layer route error detection worked correctly. The third test ensured that the AODV-LL implementation operated correctly with both two and four wireless interfaces.

7.1.1 Optimal Transmit Retry Limit Test

The transmit retry limit determines how many times the MAC layer should retry transmitting a frame after is has failed to receive a MAC acknowledgment for a transmitted frame. Since the trigger for the data link layer route error generation mechanism designed in this research is the event of the MAC exceeding the transmit retry limit, the value of the transmit retry limit is critical to ensure that the data link layer router error generation process operates correctly and efficiently. If the transmit retry limit is to low, it will be exceeded every time there is even a slight disruption in the network traffic flow, resulting in the route error generation process incorrectly reporting a broken route. Incorrect reports of a broken route would have adverse effects on the performance of AODV-LL as it would result in all traffic going out the affected route being unnecessarily stopped while AODV sends out RREQs to look for a new route. If the transmit retry limit is too high, then there would be unnecessary delays before the route error generation process is notified of a broken route, which in turn would create more delays in the higher layers before a new route is found.

Due to these reasons, a test was performed to find the optimal transmit retry limit. In the test the transmit retry limit was initially set to zero and then increased by one for each test run. In each test run,

a TCP stream throughput test using Netperf lasting ten seconds was run three times. The number of times the transmit retry limit was exceeded during each individual throughput test was recorded and after the three throughput tests, the average number of times the transmit retry limit was exceeded was calculated. The tests runs continued, increasing the transmit retry limit by one each time, until the number of times the retry limit was being exceeded was zero for a significant number of test runs.

7.1.2 Operational Tests

The operational tests focused on two areas: the operation of the data link layer feedback error detection, and the operation of the AODV-LL implementation. The purpose of the operational tests was to ensure that the data link layer feedback error detection was working correctly, along with the AODV-LL implementation. The AODV-LL implementation was a significantly modified version of Kernel AODV, the original hello message based AODV implementation, especially in the areas of managing neighboring node routes and route errors using the data link layer feedback. Because of this the operational tests were designed to check that AODV-LL and the route error detection operated as expected.

The tests covered three major functions of the AODV, finding a route using RREQs, completing the route finding process using RREPs, and thirdly, detecting a broken route and then taking appropriate actions to find a new route. For the test, three PCs were used, forming a source, intermediate and destination node over a two hop route with the hops being artificially created using the IP Tables software.

A total of three operation tests were performed. In the first test ICMP requests using the PING command were sent from the source node to the destination node by passing through the intermediate node. This was simply to see if the AODV-LL implementation was capable of setting up and managing a basic AODV route. The same was performed again in the second test, with the source node sending ICMP requests to the destination node by going through the intermediate node. When the connection had been correctly set up by the AODV-LL implementation, and ICMP replies were arriving back from the destination node, then the network interface on the destination node was turned off for a several seconds and then back on, forcing the route to be broken but then allowing it to be found again. This showed firstly; if the route error process on the data link layer functioned correctly, and secondly; it showed how the AODV-LL implementation would handle the route errors sent to it by the data link

layer. This same test was performed again in the third test but instead the network interface on the intermediate node instead of the destination node was switched off and back on again.

7.1.3 Compatibility Tests

The compatibility tests, tested if the AODV-LL was compatible with the hello message based AODV implementation, Kernel AODV, but was still capable of keeping hello messages to a minimum. Providing compatibility between the data link layer feedback based AODV implementation and the hello message based AODV implementation was considered important if the AODV-LL implementation was used in a real world scenario where the existence of hello message based nodes would be quite probable. The compatibility test setup was the same as the operational tests, with three PCs forming a two hop route. With the compatibility tests through, the destination node ran Kernel AODV, a hello message based AODV, instead of AODV-LL.

Two compatibility tests were performed. In the first test ICMP requests were transmitted from the source node to the destination node through the intermediate node. This was simply to show the AODV-LL implementation operating on the intermediate node could correctly set up and manage a route between an AODV-LL node and an AODV-HM based node. For the second test, ICMP requests were once again sent from the source node through the intermediate to the destination node. This time through, once the route was found and the connection operating correctly, the intermediate node's wireless interface was halted for several seconds then turned back on, forcing the link to be broken and then reset again.

7.1.4 Multiple Interfaces Tests

The multiple interface tests were designed to test the compatibility between AODV-LL and multiple interfaces, specifically two and four interfaces per node. For these tests, three PCs over a two hop route were used just like the operational and compatibility tests, although in this case, IP Tables didn't need to be used to artificially create the hops, since each interface pair could be set to operate on separate frequencies, creating non-conflicting hops. The tests were preformed as the same as the operational tests, where ICMP requests were sent from the source to the destination nodes, forcing AODV-LL to setup a route through the intermediate node. Using Ethereal, the network traffic could be monitored so it could be seen if the AODV-LL implementation was capable of working with multiple

interfaces.

For the two interface per node test, only four wireless NICs were required (as apposed to six) since only the intermediate node required two (an upstream and downstream) interfaces while the source only needed the upstream interface and the destination only needed the downstream interface. In the same way, for the four interfaces per node test, eight wireless interfaces were required (four for the intermediate and two each for the source and destination) rather then twelve. Since only three wireless interfaces (the ADMTEK chipset based interfaces) were capable of generating the data link layer feedback, it was ensured that the interfaces that would detect the broken link in the tests were ADMTEK chipset based interfaces. This way the tests could be run as if all the interfaces were capable of data link layer feedback. See *Figure 7.4*

7.2 Performance Test Methodology

The performance tests were designed to show how throughput at the application layer was affected by the combination of two software factors and three hardware factors over a range of multiple hop routes. The software factors were:

- each node using the hello message based AODV implementation Kernel AODV
- each node using the data link layer feedback based AODV implementation AODV-LL

The hardware factors were:

- each node using a single wireless interface
- each node using two wireless interfaces
- each node using four wireless interfaces

Due to the number of wireless interfaces available, tests using one and two wireless interfaces per node were done for routes with one to four hops, while for the four interfaces per node tests, routes only up to two hops were used. Out of the eight wireless cards used, only three cards were capable of generating data link layer feedback route detection which meant when using the AODV-LL implementation, only the three nodes with the capable wireless interfaces were able to detect broken links. To overcome this problem, it was assumed that this test represented a scenario where each node was static, resulting in no broken links and therefore no route errors. This assumption was considered sufficient for the performance tests since they were only designed to show how hello

Figure 7.4 Different views of the four wireless interfaces installed in the computer showing also the antennas used. The top three interfaces are the ADM8211 chipset interfaces while the bottom interface is the RTL8180L chipset interface

messages and multiple wireless interfaces affected throughput rather then showing how well the data link layer feedback operated.

Within the performance tests three major tests were run. In the first major test each node used one wireless interface, in the second test each node used two wireless interfaces and in the third test each node used four wireless interfaces. For each test, firstly all the nodes ran Kernel AODV and then the tests were re-run with each node running AODV-LL.

In each test, first the AODV implementation was used to create a route to the furtherest node in the route, whether this was four hops away as with the single and double cards per node configuration or two hops away for the four card per node configuration. Once the route was set up, throughput tests were performed using Netperf. The throughput tests were run from the source node to the furtherest node over the route, and then re-run to the next node one hop closer and so forth so that all the different hop routes were tested. Each individual throughput test lasted for sixty seconds and was run ten times resulting in ten minutes per test and a total of forty minutes to test all the different hop routes in the single and double card configuration tests and twenty minutes to test the four card configuration test. From each of the ten individual throughput tests, the average was then calculated.

7.3 Testing Using Actual Hardware and Software

Performing tests using simulation software or similar has the advantage of easily being able to control the test environment but has the disadvantage of not always reflecting the actual test scenario. In this research, actual hardware and software was used to help reflect real world environments and also show that the results can be achieved using easily accessible hardware and software. The testing environment used in this research was based within a laboratory using computers with standard wireless cards. Even through this barely represents actual wireless mesh networks that could exist, such as a crowd of people on a campus with wireless capable notebooks, the problems encounted during the tests, still highlighted the often non-uniform behavior of actual wireless hardware. During the tests it was observed that wireless links are difficult to predict and that throughput could often be influenced by factors such as the wireless equipment not performing as expected, or the placement of the wireless antennas or even other equipment within the laboratory such as chairs and tables. Dealing with other wireless hardware in the surrounding university campus also presented a problem at times.

Because of these reasons it was always ensured that before each test, the environment was made as uniform as possible, that no other wireless networks were present while the tests were taking place, and that all wireless equipment was operating as expected. To ensure good placement of antennas, the destination node across the multiple hop routes was "pinged" while moving the antennas. When the ping times were consistent the antennas were considered to be in a suitable position and then this position was kept the same for all tests to ensure no test configuration had an advantage over any other tests. This way it was ensured that the test results best reflected the effects of the AODV-HM and AODV-LL implementations, as well as the different wireless interface configurations.

8 Results And Discussion

8.1 Functionality Results

The functionality tests were designed to test the AODV-LL implementation for correct AODV operation and compatibility with the data link layer route error detection technique, hello message based AODV implementations and multiple wireless interfaces. The following show the results of the functionality tests.

8.1.1 Optimal Transmit Retry Limit

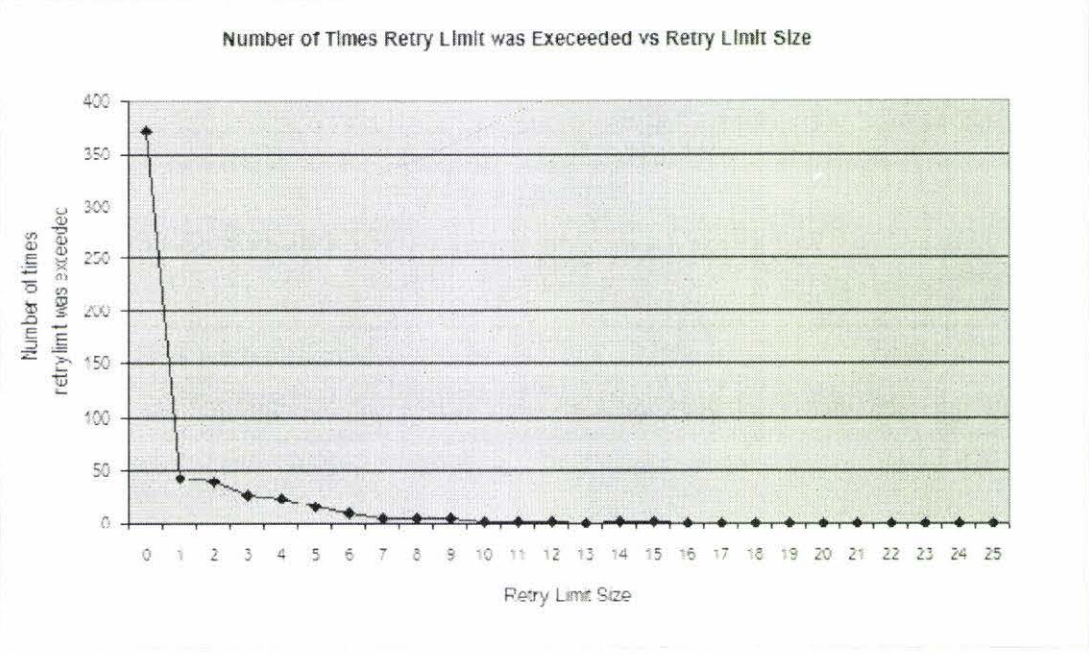


Figure 8.1 Number of Times Retry Limit was Exceeded vs Retry Limit Size

Figure 8.1 shows the results of the optimal transmit retry limit test. When the retry limit was set to zero, there was a large number of times it was exceeded which then dropped dramatically when the transmit retry limit was set to one. This shows how during the throughput test, a large number of packets needed to be retransmitted at least once. Increasing the retry limit from one to ten, decreases the number of times it is exceeded in a fairly linear fashion. Setting the retry limit to sixteen and above consistently resulted in zero retransmits. Based on these results it was decided to set the retry transmit limit to twenty, which was chosen to ensure that the retry transmit limit would only be exceeded when the link was truly broken but also keep the broken link detection time to a minimum.

The results from this test shows how critical it is to correctly set the transmit retry limit for the data link layer route error detection to work correctly. In a real scenario, if the transmit retry limit was set anyway below five, it would most likely result in all the routes being useless, since the broken route detection process in the data link layer would be constantly and incorrectly detecting broken routes, prompting AODV to find new routes. Even with the retry limit set between five and ten would cause serious disruptions to any route. Based on these test results, setting the retry limit to twenty would be sufficient for any applications that are as network intensive or less as the throughput tests used in the test. In a real world scenario though, if applications are used which are very network usage intensive, then it would be best to re-evaluate the retry limit size, in order to ensure that there are no incorrect broken route detections.

8.1.2 Operation

The first operation test showed if the AODV-LL implementation was capable of creating and managing a simple two-hop route from the source node which passed through the intermediate node to the destination node. In this first test, the kernel messages on each node – which includes messages output by the AODV-LL implementation – were collected along with the Ethereal capture of the network traffic on the intermediate node. *Figure 8.2* to *Figure 8.8* shows these results, starting from the source node through to the destination node.

Figure 8.2 shows the kernel messages on the source node for the first operation test. They show how AODV-LL firstly configures itself and the wireless network interfaces it uses when it loads up. AODV-LL then transmits a RREQ for 10.0.0.3 (the destination node's IP address) and then later receives the RREP back from the destination node. Before AODV-LL processes the RREP from the destination

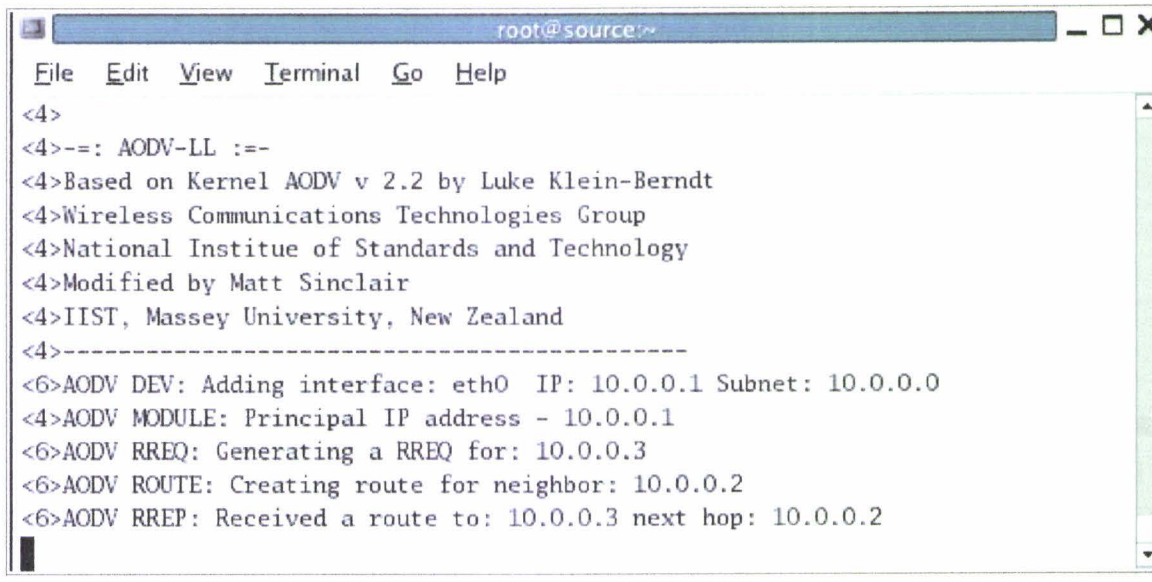


Figure 8.2 Messages displayed by the kernel on the source node in first operational test

node, the kernel messages shows how it firstly sets up a neighboring node route to the intermediate node (10.0.0.2). This is necessary as a route to the intermediate node is needed as it becomes the next hop or gateway to the destination node.

Figure 8.3 shows the Ethereal capture of the network traffic passing through the intermediate node. Although the network traffic could have been captured on both the source and destination nodes, it was sufficient to only capture the traffic on the intermediate node, since all traffic passed through it, and as a result providing the best picture of the operation of the AODV-LL implementation. In the Ethereal packet captures, the green bars show AODV RREQs, the yellow bars show AODV RREPs and the red bars show the ICMP requests and replies. *Figure 8.3* shows how firstly the source node (shown by the IP source address column) broadcasts (shown by the IP destination column where 255.255.255.255 is the broadcast address) an AODV RREQ packet (as shown by the info column). The intermediate node receives the RREQ broadcast and re-broadcasts it, as at that time it doesn't have a route to the destination node. The destination node receives the RREQ broadcast from the intermediate node and

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
1	0.000000	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.
2	0.000483	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.
5	0.003193	AODV	destination	intermediate	destination	intermediate	Route Reply, D: 10.0.
8	0.004761	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0.
9	0.040317	ICMP	source	destination	source	intermediate	Echo (ping) request
10	0.040436	ICMP	source	destination	intermediate	destination	Echo (ping) request
11	0.042120	ICMP	destination	source	destination	intermediate	Echo (ping) reply
12	0.042178	ICMP	destination	source	intermediate	source	Echo (ping) reply
13	1.005821	ICMP	source	destination	source	intermediate	Echo (ping) request
14	1.005897	ICMP	source	destination	intermediate	destination	Echo (ping) request
15	1.007398	ICMP	destination	source	destination	intermediate	Echo (ping) reply
16	1.007424	ICMP	destination	source	intermediate	source	Echo (ping) reply

Figure 8.3 AODV and ICMP network traffic on the intermediate node during the first operational test

then transmits a RREP back to the intermediate node. The intermediate node then forwards the RREP to the source node, completing the route. The source node can now begin the ICMP request/reply transactions with the destination node. For the ICMP packets, the IP source and destination addresses say the packets are going from the source node to the destination node for the ICMP requests and in the opposite direction for the ICMP replies, although by looking at the MAC addresses within the ICMP packets, it can be seen that as expected, they are actually traveling from the source node, to the intermediate node and then to the destination node, and vice versa for the ICMP replies.

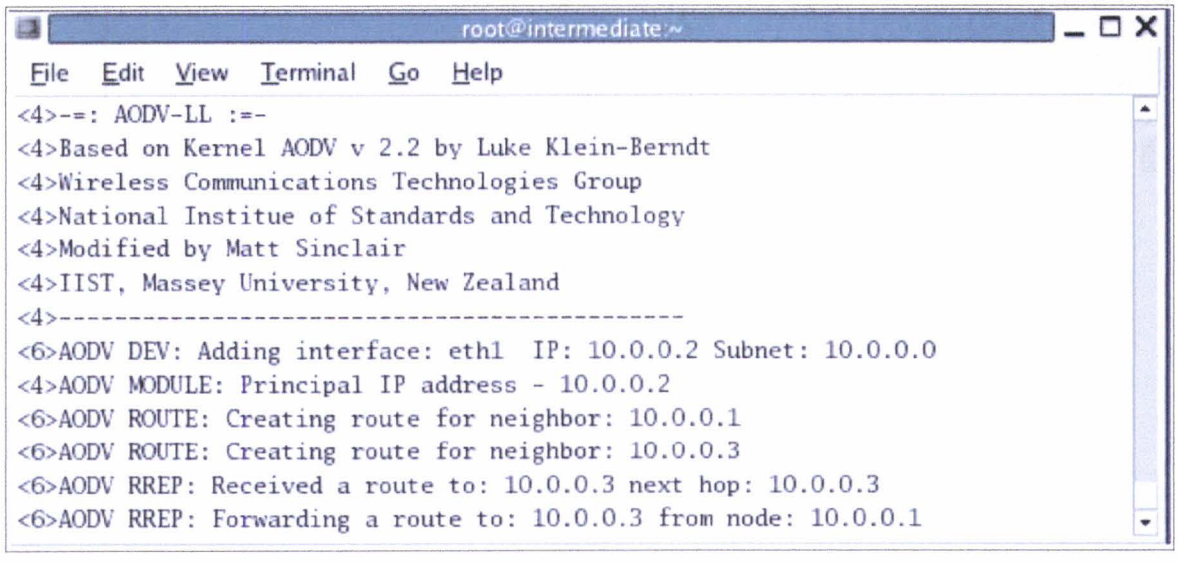


Figure 8.4 Kernel messages on the intermediate node during the first operational test

Figure 8.4 shows the kernel messages for the intermediate node during the test. Unlike with the Ethereal capture, the kernel messages also shows AODV-LL setting up routes to it's neighboring nodes, the source node (10.0.0.1) and the destination node (10.0.0.3). *Figure 8.5* shows the kernel messages displayed on the destination node during the test. They are the similar to the messages on the source node, expect a RREP is being generated, instead of the RREQ.

These results showed that the AODV-LL implementation could successfully find and create a simple two hop route.

The second operational test checked that the route error detection process built into the wireless interface drivers, operated correctly and also that the AODV-LL responded correctly to the route error information passed to it by the wireless interface driver on the data link layer. In the second operational test, a simple route was set up from the source node and through the intermediate node to the

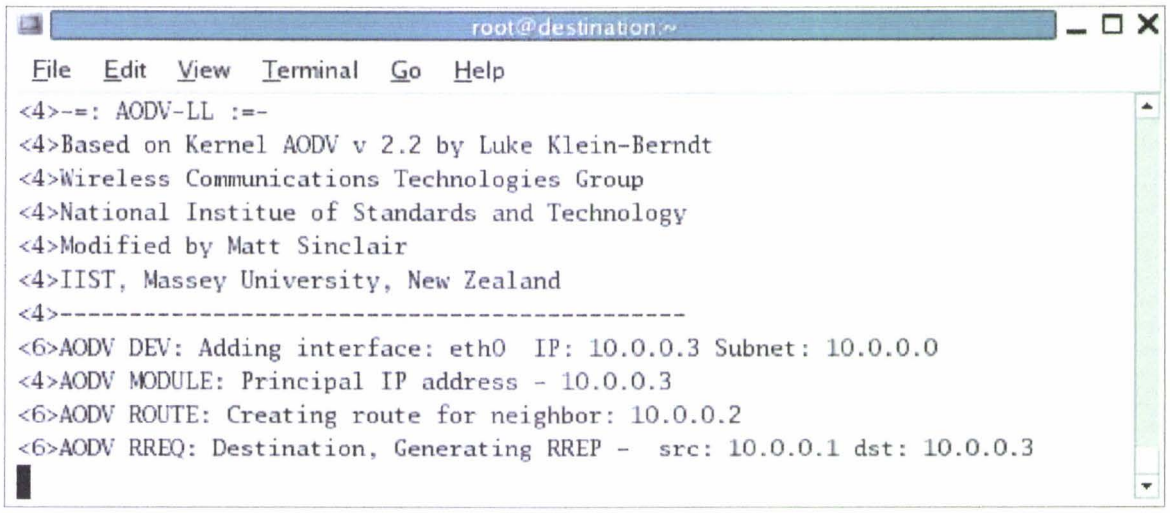


Figure 8.5 Kernel messages displayed on the destination node during the first operational test

destination node. The interface on the destination node was turned off then back on, forcing the route to be broken and then allowing it to be found again. The behavior of the AODV-LL implementation during this test was viewed through Ethereal and the kernel messages.

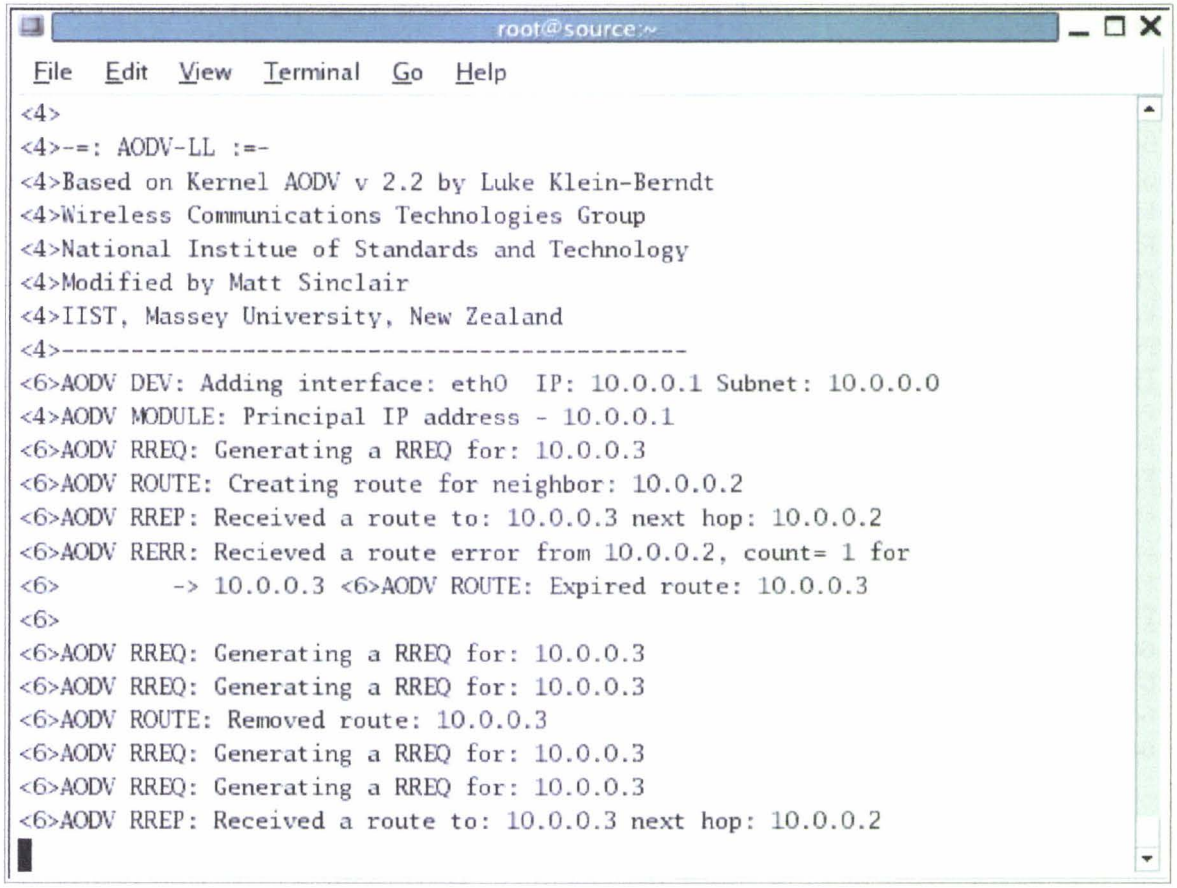


Figure 8.6 Kernel messages displayed on the source node during the second operational test

Figure 8.6 shows the AODV-LL implementation on the source node going through the process of finding and creating a route to the destination node and then receiving a RERR for the destination node, from the intermediate node. AODV-LL responds to this by immediately expiring the the route to the destination node so that it no longer used and then starts generating RREQs for the destination node. A short while later the route to the destination node is removed completely and the RREQs continue. This process of firstly expiring a route, which involves marking the route as invalid so it isn't used, but not completely removing the route (and the associated sequence number) until a certain time later, is required by the AODV protocol to stop routing loops occurring. The last part of the kernel messages show how the route to the destination is found again.

Figure 8.7 shows the ICMP request results during the test, showing the route going down and then restored. They also show how as a result of AODV removing the broken route, the kernel routing process then correctly reports the destination node as being unreachable.

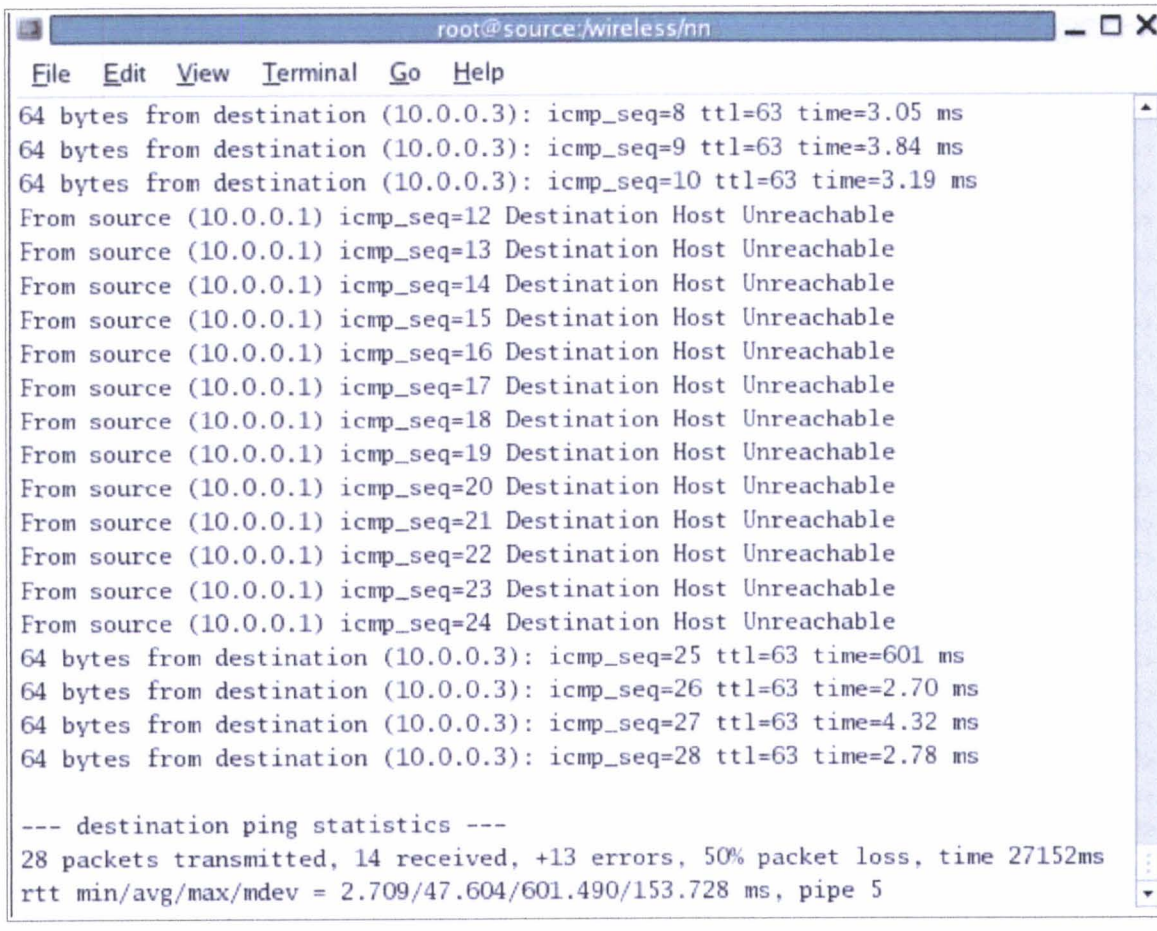


Figure 8.7 ICMP request results during the second operational test

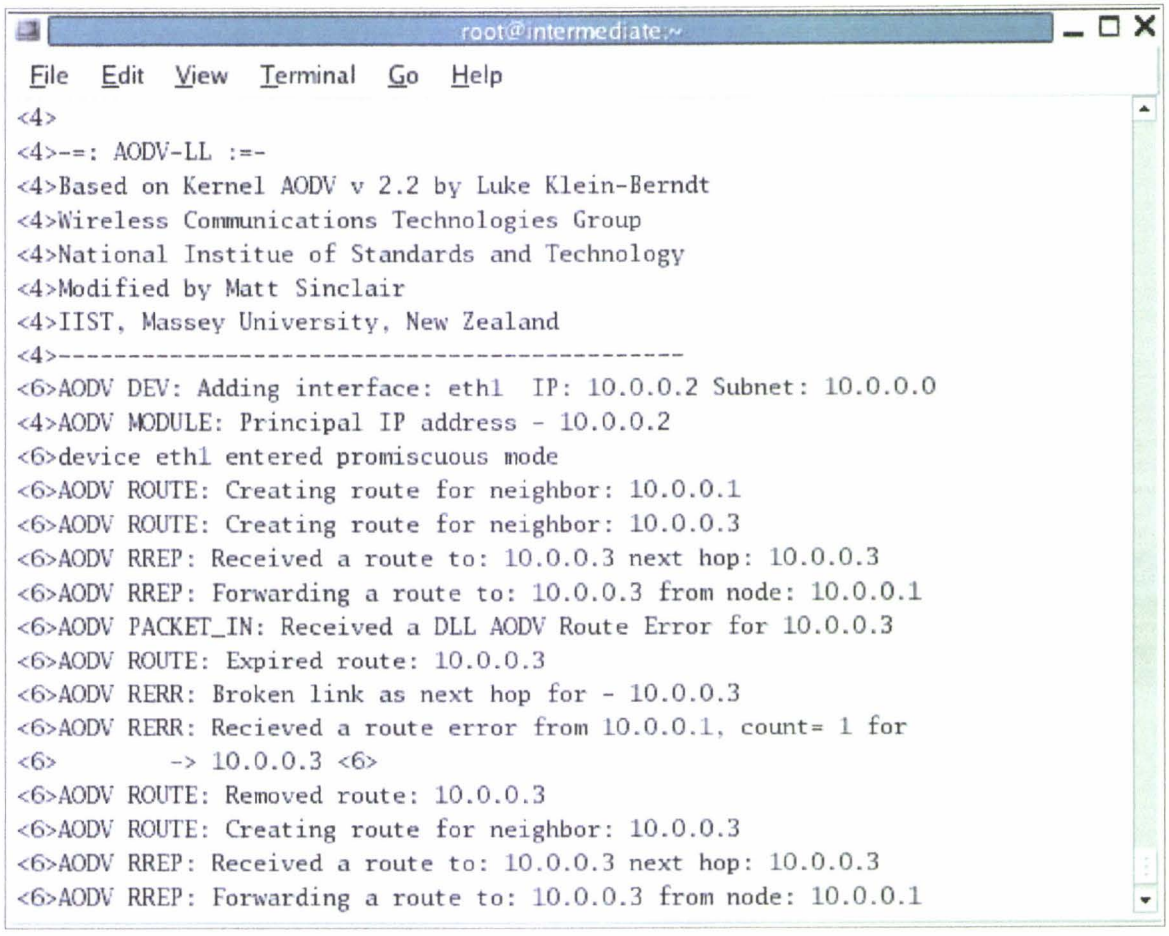


Figure 8.8 Kernel messages displayed on the intermediate during the second operational test

The next results (*Figure 8.8*) show the kernel messages displayed on the intermediate node during the test. This shows the intermediate setting up the route between the source and destination node, and then forwarding data between them. The *AODV Packet_in* process within AODV-LL then reports receiving a RERR from the data link layer. This is the RERR generated by the wireless interface driver, after it has detected that the transmit retry limit was exceeded, due to the link between the intermediate and destination node going down. The *AODV Route* process then immediately expires the route to the destination node after which the *AODV RERR* process takes over, notifying through the kernel messages that a broken link to 10.0.0.3 (the destination node) has occurred. AODV then broadcasts a RERR. Next the *AODV RERR* process informs it has received a RERR from 10.0.0.1 (the source node). This is the RERR rebroadcast by the source node as a result of the RERR that has just been transmitted from intermediate node. The intermediate node recognizes that this RERR from the source is the same as the one it has just broadcast and processes it no further. This can be seen by comparing *Figure 8.8* with *Figure 8.6*. In *Figure 8.6*, when the source node receives the RERR from the intermediate node, the *AODV RERR* process informs the kernel that the RERR has been received, then straight after, informs the kernel that the affected route has been expired. In *Figure 8.8* when the

intermediate node receives the RERR for the source node, the *AODV RERR* process informs the kernel that it has received the RERR but then takes no further action since the route to 10.0.0.3 has already been expired. The next message in *Figure 8.8* shows the *AODV ROUTE* process removing the route after waiting the "delete period". The next messages show that the a RREP is received from the destination node, as the link is restored, and then the intermediate node forwarding the RREP on to the source node to complete the route.

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
49	9.083552	ICMP	destination	source	destination	intermediate	Echo (ping) reply
50	9.083576	ICMP	destination	source	intermediate	source	Echo (ping) reply
51	10.091467	ICMP	source	destination	source	intermediate	Echo (ping) request
52	10.091524	ICMP	source	destination	intermediate	destination	Echo (ping) request
53	10.251078	AODV	destination	source	destination	intermediate	
54	10.251232	ICMP	intermediate	destination	intermediate	destination	Time-to-live exceeded
55	10.251624	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Error, Dest Count=1
57	10.410424	AODV	source	255.255.255.255	source	Broadcast	Route Error, Dest Count=1
58	11.091429	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.0.0.3
59	11.091874	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.0.0.3
60	12.691268	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.0.0.3
61	12.691644	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.0.0.3

Figure 8.9 Second operational test, intermediate node: detecting the broken route

Figure 8.9 and *Figure 8.10* both show the same Ethereal capture from the intermediate node during the second operational test but it has been split into two screenshots to make it easier to view. *Figure 8.9* shows the first half of the test. In this figure, the white bar is Ethereal detecting the RERR packet being sent from the wireless interface driver on the data link layer to AODV-LL in the routing layer. Although Ethereal decodes the packet as an AODV packet by viewing the UDP port address within the packet, it doesn't detect it as a RERR message, since the packet is missing the UDP checksum on the end, resulting in Ethereal checking the wrong parts of the packet due to the incorrect length of the packet. Having a UDP checksum on the packet wasn't necessary as the packet is simply sent up the network stack and not transmitted. When the route error detection process within the wireless interface driver creates the RERR packet, it sets the IP and MAC destination and source addresses as if the RERR was actually received from the node at the end of the broken link. The reason for this is so that AODV-LL didn't discard the RERR, thinking it was from itself. This can be seen in the Ethereal capture by looking at the IP and MAC source and destination address columns, for the data link layer RERR error packet. It is interesting to see that the time between detecting the broken link and the data link layer RERR being detected, and then the actual AODV RERR being broadcast, is almost instant. With hello message based AODV implementations, usually at least two hello messages have to be missed before a link is considered broken. Since hello messages are normally broadcast once a second, this means for the AODV-HM implementations, it would normally take two seconds at least before a RERR is

generated after a link. This is significantly longer then when using data link layer route error detection. In two seconds, while the AODV-HM implementation is still just detecting the link is broken, an AODV-LL implementation could have easily detected the broken route and found a new route to the destination, keeping the network disruption to a minimum.

The Ethereal capture in *Figure 8.9* continues on to show that after the data link layer RERR is received, the intermediate node then immediately broadcasts a proper AODV RERR, which is forwarded on by the source node. The source node then starts broadcasting RREQs, looking for the destination node, which are forwarded on by the intermediate node.

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
76	23.101151	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
76	23.110836	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10
77	23.111088	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
78	24.710711	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10
79	24.711113	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
88	24.715974	AODV	destination	intermediate	destination	intermediate	Route Reply, D: 10.0
89	24.716291	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0
90	24.718863	ICMP	source	destination	source	intermediate	Echo (ping) request
91	24.718904	ICMP	source	destination	intermediate	destination	Echo (ping) request
92	24.720961	ICMP	destination	source	destination	intermediate	Echo (ping) reply
93	24.720994	ICMP	destination	source	intermediate	source	Echo (ping) reply
94	25.130746	ICMP	source	destination	source	intermediate	Echo (ping) request

Figure 8.10 Second operational test, intermediate node: restoring the route

The second half of the test, shown by *Figure 8.10* simply shows how the intermediate node finally receives a RREP from the destination node, after it has started back up again. The RREP is forwarded on to the source node and the ICMP packet transfer starts again.

Figure 8.11 shows the kernel messages on the destination node during the test. The kernel messages show AODV-LL setting up the route to the source node, followed by the wireless interface being turned off (*adm8201 close)* and then being turned back on again (*adm8211 open*), followed by AODV generating RREPs for the RREQs received from the intermediate node. Four RREPs are generated since there is a significant delay between when the wireless interface starts receiving packets and when it is capable of actually transmitting packets. Because of this, around four RREQs were received by the wireless interface and sent to AODV to process, but it was not until the fourth RREQ was sent back down by AODV that the wireless interface was capable of actually transmitting packets.

The results from the second operational test, showed that the data link layer route error detection worked correctly and that the AODV-LL implementation also responded correctly when the data link layer route error was sent to it.

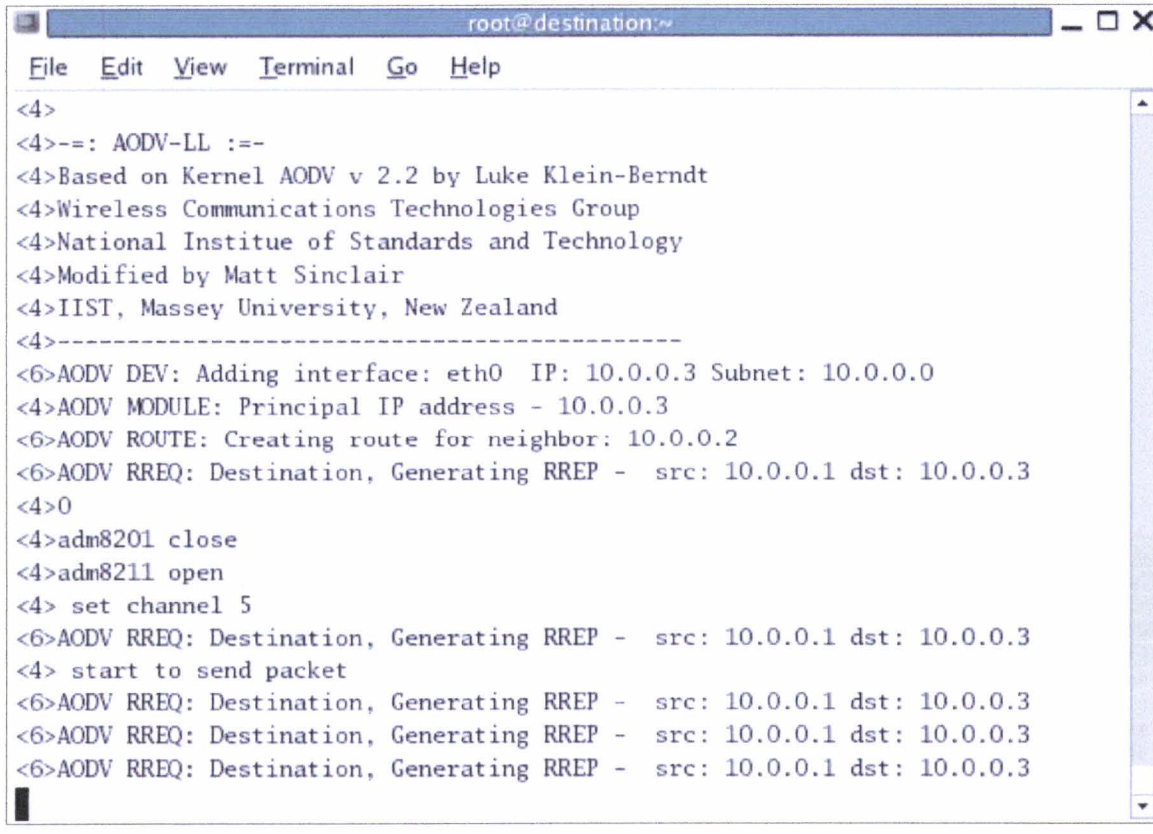


Figure 8.11 Kernel messages displayed on the destination node during the second operational test

The third operational test, was the same as the second, except the wireless interface on the intermediate node, rather then the destination node, was stopped and then restarted again some time later. For this test Ethereal could not be used on the intermediate node, as the wireless interface on this node was stopped. This test was designed to observe if the AODV-LL implementation could operate correctly when an intermediate node went down.

As shown in *Figure 8.12,* the third operational test started of with the source node going through the normal process of communicating with the intermediate node, to find a route to the destination node. Shortly after the intermediate node had forwarded on the ICMP ping reply shown by frame 18 in *Figure 8.13*, the wireless interface on the intermediate node is stopped, forcing the broken route. One second

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
1	0.000000	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.
2	0.001010	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.
5	0.004488	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0.
6	0.046073	ICMP	source	destination	source	intermediate	Echo (ping) request
8	0.051462	ICMP	destination	source	intermediate	source	Echo (ping) reply
9	1.010646	ICMP	source	destination	source	intermediate	Echo (ping) request
10	1.013596	ICMP	destination	source	intermediate	source	Echo (ping) reply
11	2.020608	ICMP	source	destination	source	intermediate	Echo (ping) request

Figure 8.12 Third operational test, source node: finding the route

No.	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
17	5.050614	ICMP	source	destination	source	intermediate	Echo (ping) request
18	5.053441	ICMP	destination	source	intermediate	source	Echo (ping) reply
19	6.060604	ICMP	source	destination	source	intermediate	Echo (ping) request
20	6.196334	AODV	destination	source	intermediate	source	
21	6.196751	AODV	source	255.255.255.255	source	Broadcast	Route Error, Dest Count=2
22	7.060655	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.0.0.3
23	8.660568	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.0.0.3
24	11.060619	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.0.0.3
25	11.060711	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.0.0.3

Figure 8.13 Third operational test, source node: detecting the broken route

later the source node tries to transmit another ICMP ping request to the destination, shown by frame 19 in *Figure 8.13*. About 0.1 seconds later, through trying to transmit the ICMP ping request, the wireless interface driver has detected that the transmit retry limit was exceeded, assuming correctly that the route is broken, and as a result generates and sends an AODV RERR up the network stack, as shown by frame 20. Immediately, in frame 21, AODV generates a proper AODV RERR, transmitting it to the broadcast address. One second later, and from then on every second, AODV transmits RREQs to find a route to the destination node.

Figure 8.14 shows the intermediate node starting back up, resulting it in forwarding on the source nodes RREQs as shown in frames 28 to 31. As a result, in frame 34, the source node finally receives a RREP back, through the intermediate node, from the destination. This sets the route back up and the ICMP packet transfer continues. *Figure 8.14* also shows the delay between the wireless interface on the intermediate node after it was restarted being able to transmit and then being able to receive packets. This delay resulted in four RREQs being generated by AODV before the wireless interface could transmit any of them, then resulting in four RREPs being transmitted back from the destination

No.	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
26	12.660570	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10
27	15.060577	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10
28	15.062425	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
29	15.062835	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
30	15.063450	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
31	15.064005	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
34	15.068000	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0
35	15.068385	ICMP	source	destination	source	intermediate	Echo (ping) request
36	15.068398	ICMP	source	destination	source	intermediate	Echo (ping) request
37	15.068421	ICMP	source	destination	source	intermediate	Echo (ping) request
38	15.068680	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0
39	15.070565	ICMP	source	destination	source	intermediate	Echo (ping) request
41	15.076317	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0
42	15.077418	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0
43	15.079005	ICMP	destination	source	intermediate	source	Echo (ping) reply
44	15.080625	ICMP	destination	source	intermediate	source	Echo (ping) reply
45	15.082966	ICMP	destination	source	intermediate	source	Echo (ping) reply
46	15.083522	ICMP	destination	source	intermediate	source	Echo (ping) reply
47	16.080578	ICMP	source	destination	source	intermediate	Echo (ping) request
48	17.080565	ICMP	source	destination	source	intermediate	Echo (ping) request
49	17.084669	ICMP	destination	source	intermediate	source	Echo (ping) reply
50	18.090578	ICMP	source	destination	source	intermediate	Echo (ping) request

Figure 8.14 Third operational test, source node: restoring the route

node.

Like the second test, the third operational test showed that the route error detection worked correctly and that the AODV-LL implementation was capable of managing nodes leaving the route and then joining the route again.

Overall the operational tests showed that the route error detection process within the wireless interface driver was fast and effective, that the AODV-LL implementation responded correctly when it was notified of a broken link and that it operated correctly as specified by the AODV protocol.

8.1.3 Compatibility

The compatibility tests were designed to show that the AODV-LL implementation was completely compatible with AODV-HM implementations but was still capable of keeping hello messages to a minimum. A total of two tests were performed.

In the first test, the source and intermediate nodes were running the AODV-LL implementation and the destination node was running the AODV-HM implementation. In *Figure 8.15* the purple bars show hello messages, while the green bars show RREQs, the yellow bars show RREPs and the red bars show ICMP packets. *Figure 8.15* shows a screenshot taken from the packet capture software Ethereal while the source node finds a route to the destination node and then begins to ping the destination.

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
3	2.014006	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
5	3.013996	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
7	4.014044	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
9	5.014086	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
11	6.014132	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
13	7.014194	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
15	8.014232	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
17	8.845193	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10
18	8.846281	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
19	8.847844	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0
20	8.885915	ICMP	source	destination	source	intermediate	Echo (ping) request
21	8.889194	ICMP	destination	source	intermediate	source	Echo (ping) reply
22	9.014314	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
24	9.855873	ICMP	source	destination	source	intermediate	Echo (ping) request
25	9.858959	ICMP	destination	source	intermediate	source	Echo (ping) reply
26	10.014307	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
28	10.865875	ICMP	source	destination	source	intermediate	Echo (ping) request
29	10.869718	ICMP	destination	source	intermediate	source	Echo (ping) reply
30	11.014449	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
32	11.875853	ICMP	source	destination	source	intermediate	Echo (ping) request
33	11.879216	ICMP	destination	source	intermediate	source	Echo (ping) reply

Figure 8.15 First compatibility test, source node: finding the route

During the whole test, the source node is receiving hello messages from the intermediate node, about once per second as according to the AODV protocol recommended rate. Even though the intermediate node is running the AODV-LL implementation, it is transmitting hello messages, as it is in range of the destination node which is running the AODV-HM implementation. To be compatible with a hello message based AODV implementation, the AODV-LL must broadcast hello messages, so that the AODV-HM implementation can detect the presence of the AODV-LL based node. *Figure 8.15*, shows how even though the source is receiving hello messages from the intermediate node, its not transmitting hello messages itself (if it was then they would be captured by Ethereal). This is because the AODV-LL on the the source node can detect that the hello messages its receiving are from another AODV-LL based node. This feature is important since it means that when an AODV-HM based node enters the network, the hello messages that need to be broadcast are kept to a minimum. The figure carries on to show that as normal, the source node transmits a RREQ, which is then forwarded on by the intermediate node to the destination node. Soon after a RREP is received back and the ICMP packet transfer begins.

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
8	4.003593	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
9	4.040042	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0
10	5.003591	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
11	5.050024	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0
12	6.003605	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
13	6.060028	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0
14	7.003597	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
15	7.070037	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0
16	7.835129	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10
17	7.835600	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10
18	7.836740	AODV	destination	intermediate	destination	intermediate	Route Reply, D: 10.0
19	7.837059	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0
20	7.875891	ICMP	source	destination	source	intermediate	Echo (ping) request
21	7.876060	ICMP	source	destination	intermediate	destination	Echo (ping) request
22	7.877851	ICMP	destination	source	destination	intermediate	Echo (ping) reply
23	7.877889	ICMP	destination	source	intermediate	source	Echo (ping) reply
24	8.003619	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
25	8.080021	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0
26	8.845785	ICMP	source	destination	source	intermediate	Echo (ping) request
27	8.845844	ICMP	source	destination	intermediate	destination	Echo (ping) request
28	8.847537	ICMP	destination	source	destination	intermediate	Echo (ping) reply
29	8.847564	ICMP	destination	source	intermediate	source	Echo (ping) reply
30	9.003584	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0
31	9.090030	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0
32	9.855745	ICMP	source	destination	source	intermediate	Echo (ping) request
33	9.855790	ICMP	source	destination	intermediate	destination	Echo (ping) request

Figure 8.16 First compatibility test, intermediate node: setting up the route

Figure 8.16 shows clearly the compatibility capabilities of the AODV-LL implementation. The intermediate node is receiving hello messages from the destination node every second and responds by transmitting a hello message back. As a result, the AODV route finding process between the two different AODV types can complete without any errors and the ICMP packet transfer begins.

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
11	6.014121	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
12	6.059955	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
13	7.014130	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
14	7.069951	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
15	8.014116	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
16	8.079957	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
18	8.846115	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.0.0.3, O: 10.0.0.
19	8.846427	AODV	destination	intermediate	destination	intermediate	Route Reply, D: 10.0.0.3, O: 10.0.0.1
20	8.887158	ICMP	source	destination	intermediate	destination	Echo (ping) request
21	8.887341	ICMP	destination	source	destination	intermediate	Echo (ping) reply
22	9.014128	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
23	9.089938	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
24	9.856727	ICMP	source	destination	intermediate	destination	Echo (ping) request
25	9.856862	ICMP	destination	source	destination	intermediate	Echo (ping) reply
26	10.014085	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
27	10.099941	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
28	10.866938	ICMP	source	destination	intermediate	destination	Echo (ping) request
29	10.867070	ICMP	destination	source	destination	intermediate	Echo (ping) reply
30	11.014170	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
31	11.109936	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
32	11.876661	ICMP	source	destination	intermediate	destination	Echo (ping) request
33	11.876814	ICMP	destination	source	destination	intermediate	Echo (ping) reply

Figure 8.17 First compatibility test, destination node: setting up the route

Figure 8.17 shows the test as seen from the destination node running the AODV-HM implementation. It receives hello messages from the intermediate node and therefore treats it as a node running AODV with hello messages. The first compatibility test showed that the AODV-LL implementation can operate with an AODV-HM node, but still be able to detect between AODV-HM and AODV-LL based nodes.

The second compatibility test involved setting the usual two hop route between the source and destination nodes, with the destination node running the AODV-HM implementation, and the other nodes running the AODV-LL implementations. Once the route was set up, it was broken by force by halting the interface on the intermediate node for a period of time.

Figure 8.18 shows how in the second test, when the route is broken, the source node detects the broken route on the data link layer using the transmit retry limit, and sends a RRER (shown by frame 66) to AODV which then generates a proper RERR to be transmitted. Later the wireless interface on the intermediate node is started again and the intermediate node starts forwarding the RREQs (frames 83 and 84) from the source node. Soon after this point, the intermediate node received a hello message from the destination node which is running the AODV-HM implementation. This is known because in frame 85, the intermediate node starts to broadcast hello messages in response to the hello messages being received from the destination node.

It is interesting to observe that it is only soon after the intermediate starts to broadcast hello messages that a RREP is received back from the destination node. This is because before the destination node received any hello messages from the intermediate node, all AODV messages from the intermediate

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
52	18.163863	ICMP	source	destination	source	intermediate	Echo (ping) request
53	18.166329	ICMP	destination	source	intermediate	source	Echo (ping) reply
55	18.527005	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.
56	19.173881	ICMP	source	destination	source	intermediate	Echo (ping) request
57	19.178185	ICMP	destination	source	intermediate	source	Echo (ping) reply
59	19.527068	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.
60	20.183860	ICMP	source	destination	source	intermediate	Echo (ping) request
61	20.186426	ICMP	destination	source	intermediate	source	Echo (ping) reply
63	20.527092	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.
64	21.193862	ICMP	source	destination	source	intermediate	Echo (ping) request
66	21.335932	AODV	destination	source	intermediate	source	
67	21.336397	AODV	source	255.255.255.255	source	Broadcast	Route Error, Dest Cou
68	22.193921	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.
72	23.793825	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.
76	25.193939	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.
77	26.194039	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.
81	27.793835	AODV	source	255.255.255.255	source	Broadcast	Route Request, D: 10.
83	28.279348	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.
84	28.279867	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.
85	28.281339	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.
86	28.281834	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.
87	28.283522	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.
90	28.285552	AODV	intermediate	source	intermediate	source	Route Reply, D: 10.0.
91	28.285945	ICMP	source	destination	source	intermediate	Echo (ping) request
92	28.285956	ICMP	source	destination	source	intermediate	Echo (ping) request
93	28.290261	ICMP	destination	source	intermediate	source	Echo (ping) reply
95	28.294033	ICMP	destination	source	intermediate	source	Echo (ping) reply
96	29.213865	ICMP	source	destination	source	intermediate	Echo (ping) request
97	29.217706	ICMP	destination	source	intermediate	source	Echo (ping) reply
98	29.277522	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.
100	30.223833	ICMP	source	destination	source	intermediate	Echo (ping) request
101	30.277554	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.
103	31.223841	ICMP	source	destination	source	intermediate	Echo (ping) request
104	31.228454	ICMP	destination	source	intermediate	source	Echo (ping) reply

Figure 8.18 Second compatibility test, source node: detecting and restoring the broken route

node were discarded, as it was not until hello messages were received, that the intermediate node was added to the destination node's route table. Hello message based AODV implementations have to take this precaution of discarding AODV messages from neighboring nodes that don't transmit hello messages, as it is only through the neighboring node transmitting the hello messages that the receiving node can monitor the link, and detect broken routes. By accepting AODV messages from a neighboring node not transmitting hello messages, routes can form which can't be monitored or managed properly.

In *Figure 8.19* which shows the test from the destination node's point of view, the route is broken soon after frame 49, which is the last hello message from the intermediate node. About two seconds later, the destination node has transmitted two hello messages but has received none from the intermediate node and then assumes that the route to the intermediate node is broken and a RERR is broadcast. Later, in frame 75, the destination node starts receiving RREQs from the intermediate node but no action is taken until the first hello message is received from the intermediate node in frame 78. The intermediate node is now added to the destination node's route table and a RREP is transmitted back in response to the next RREQ from the intermediate node. This then restores the route and the ICMP packet transfer continues.

No. .	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Dest Addr	Info
46	13.100179	ICMP	source	destination	intermediate	destination	Echo (ping) request
47	13.100305	ICMP	destination	source	destination	intermediate	Echo (ping) reply
48	13.126094	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
49	13.442443	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
50	14.136112	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
53	15.146100	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
55	16.156111	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
57	16.546207	AODV	destination	255.255.255.255	destination	Broadcast	Route Error, Dest Count=2
60	17.166109	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
63	18.176121	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
68	19.186098	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
70	20.196104	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
75	21.194297	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.0.0.3, O: 10.0.0.
76	21.194813	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.0.0.3, O: 10.0.0
78	21.196285	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
79	21.196783	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Request, D: 10.0.0.3, O: 10.0.0
80	21.196986	AODV	destination	intermediate	destination	intermediate	Route Reply, D: 10.0.0.3, O: 10.0.0.1
82	21.198468	AODV	intermediate	255.255.255.255	intermediate	Broadcast	Route Reply, D: 10.0.0.2, O: 10.0.0.2
84	21.202234	ICMP	source	destination	intermediate	destination	Echo (ping) request
85	21.202318	ICMP	destination	source	destination	intermediate	Echo (ping) reply
86	21.203630	ICMP	intermediate	destination	intermediate	destination	Redirect
87	21.206103	AODV	destination	255.255.255.255	destination	Broadcast	Route Reply, D: 10.0.0.3, O: 10.0.0.3
88	21.206089	ICMP	source	destination	intermediate	destination	Echo (ping) request
89	21.206162	ICMP	destination	source	destination	intermediate	Echo (ping) reply
90	22.130302	ICMP	source	destination	intermediate	destination	Echo (ping) request
91	22.130424	ICMP	destination	source	destination	intermediate	Echo (ping) reply

Figure 8.19 Second compatibility test: destination node, detecting and restoring the broken route

This second test showed how the AODV-LL implementation is capable of leaving and then joining a mixed style AODV network, as well as detecting and adapting to the different hello message and non-hello message based nodes around it.

The AODV-LL implementation, in terms of route error detection, doesn't use the hello messages from AODV-HM based nodes for any purpose and instead still detects broken links to these types of nodes using the data link layer. For this reason compatibility tests showing an AODV-LL node detecting a broken link with an AODV-HM didn't need to be performed, as the results would be the same as with the operational tests.

Overall the compatibility tests showed that the AODV-LL implementation is completely compatible with hello message based AODV implementations. The AODV-LL implementation is also capable of keeping hello messages to a minimum by being able to detect hello messages from AODV-LL nodes as apposed to genuine hello messages.

8.1.4 Multiple Interfaces

Figure 8.20 shows the intermediate node's kernel route table which has been set up by the AODV-LL implementation during the first multiple interfaces test which used two wireless interfaces per node. The route table shows a route to the source and destination nodes, followed by two routes to itself,

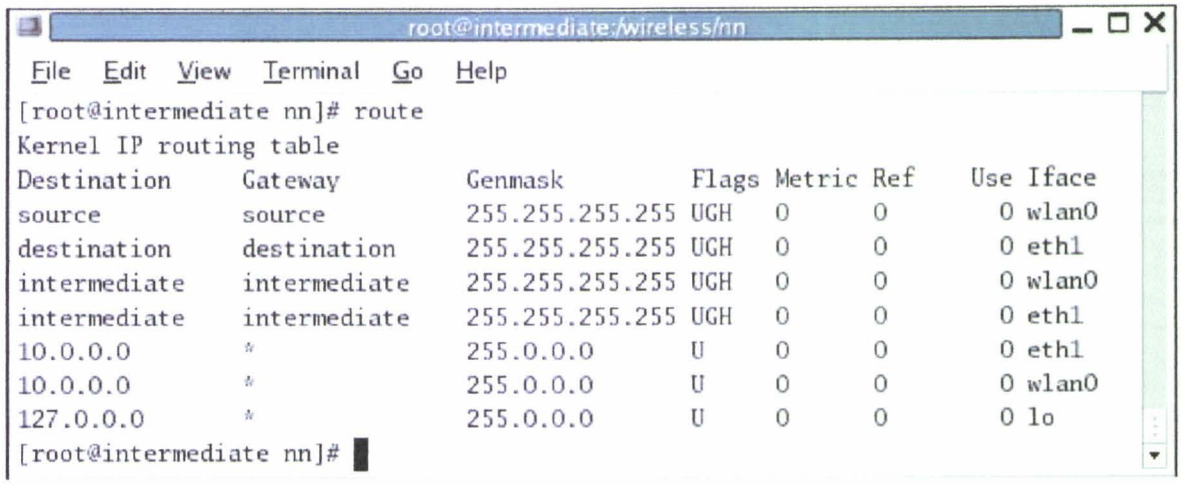
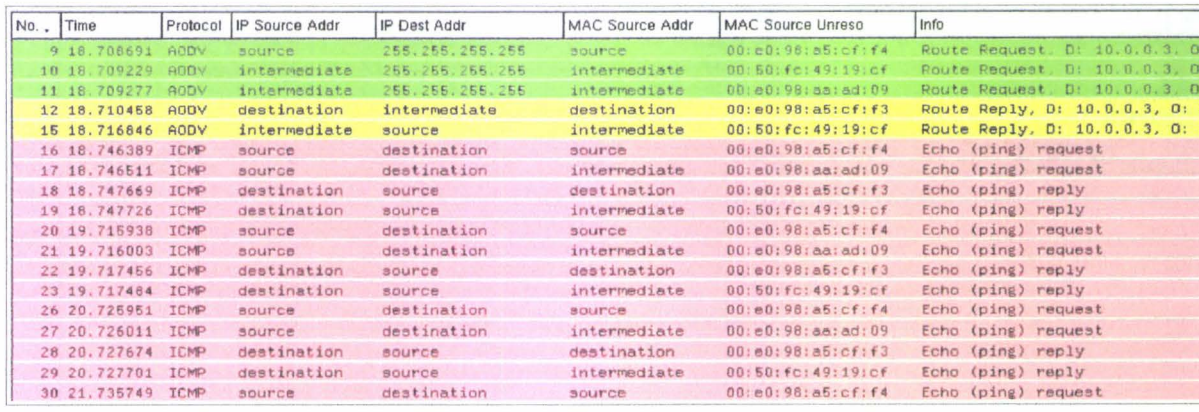


Figure 8.20 Kernel route table on the intermediate node during the two interface per node test

followed by two default routes to the 10.0.0.0 IP range and finally the loopback route. Evidence of the use of two wireless interfaces is shown in the *Iface* or interface column. In the interface column, *wlan0* is the name given to the Realtek chipset wireless interface, while *eth1* is the name given to the Admtek chipset wireless interfaceand *lo* is the loopback interface. There are two default routes and two routes to itself, since there are two wireless interfaces being used. As can be seen the route to the source node is through *wlan0*, while the route to the destination node is through *eth1* therefore showing how the two interfaces have been correctly assigned by AODV-LL to an "upstream" and "downstream" route.

Figure 8.21 shows AODV using the two interfaces on the intermediate during the multiple interface test. This can be seen by looking at the *MAC Source Unresolved* column when the ICMP packets are being forwarded between the source and destination nodes. The intermediate node's unresolved MAC source address when forwarding the ICMP request from the source to the destination is different to the unresolved MAC source address used when the ICMP reply is forwarded back in the opposite direction. This shows how the route has been set up by AODV so that one interface on the intermediate node is



No.	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Source Unreso	Info
9	18.708691	AODV	source	255.255.255.255	source	00:e0:98:a5:cf:f4	Route Request, D: 10.0.0.3, O:
10	18.709229	AODV	intermediate	255.255.255.255	intermediate	00:50:fc:49:19:cf	Route Request, D: 10.0.0.3, O:
11	18.709277	AODV	intermediate	255.255.255.255	intermediate	00:e0:98:aa:ad:09	Route Request, D: 10.0.0.3, O:
12	18.710458	AODV	destination	intermediate	destination	00:e0:98:a5:cf:f3	Route Reply, D: 10.0.0.3, O: 1
15	18.716846	AODV	intermediate	source	intermediate	00:50:fc:49:19:cf	Route Reply, D: 10.0.0.3, O: 1
16	18.746389	ICMP	source	destination	source	00:e0:98:a5:cf:f4	Echo (ping) request
17	18.746511	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request
18	18.747669	ICMP	destination	source	destination	00:e0:98:a5:cf:f3	Echo (ping) reply
19	18.747726	ICMP	destination	source	intermediate	00:50:fc:49:19:cf	Echo (ping) reply
20	19.715938	ICMP	source	destination	source	00:e0:98:a5:cf:f4	Echo (ping) request
21	19.716003	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request
22	19.717456	ICMP	destination	source	destination	00:e0:98:a5:cf:f3	Echo (ping) reply
23	19.717484	ICMP	destination	source	intermediate	00:50:fc:49:19:cf	Echo (ping) reply
26	20.725951	ICMP	source	destination	source	00:e0:98:a5:cf:f4	Echo (ping) request
27	20.726011	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request
28	20.727674	ICMP	destination	source	destination	00:e0:98:a5:cf:f3	Echo (ping) reply
29	20.727701	ICMP	destination	source	intermediate	00:50:fc:49:19:cf	Echo (ping) reply
30	21.735749	ICMP	source	destination	source	00:e0:98:a5:cf:f4	Echo (ping) request

Figure 8.21 AODV-LL using the two wireless interfaces on the intermediate node during the first ,multiple interface test

used for the link with the source node, while the other interface is used for the link with the destination node.

It is also interesting to observe that when the intermediate node receives the initial RREQ from the source node, the RREQ is then broadcast on both the interfaces, rather then a single interface. This is an important feature when using multiple interfaces, since there is no way to determine if the destination node is operating on the same channel as the first or second wireless interface. This also shows how the number of broadcast transmissions within an AODV network will increase significantly when multiple interfaces are used, since broadcast packets (at least when using two interfaces per node) need to transmitted out each interface. In the case of four interfaces per node, only two interfaces on each node are for outgoing traffic, while the other two are only for incoming traffic. Therefore, with four interfaces per node, the broadcast packets will only need to be transmitted out two interfaces. Using AODV-LL, RREQs and RERRs are the only AODV messages broadcast, while with a hello messages based AODV, every hello message is broadcast. Because of this, using multiple interfaces with a hello message based AODV could result in a significant extra amount of network traffic.

In the next set of tests four interfaces per node were used. As explained in the *Section 7.0,* AODV-LL was modified so that it could assign the four interfaces into two pairs: one pair for "upstream" traffic and the other for "downstream" traffic. Then within each pair, AODV-LL assigned one interface for only incoming traffic while the other interface was assigned only for outgoing traffic. This was also done in such a way that within each pair the "outgoing" interface was connected to the "incoming" interface on the neighboring node across the other side of the hop in the route, and vice versa for the other interface within each pair, thus creating a full duplex capable connection.

No.	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Source Unreso	Info
11	21.765784	AODV	source	255.255.255.255	source	00:30:00:10:90:9c	Route Request, D: 10
12	21.766828	AODV	source	255.255.255.255	source	00:e0:98:a5:cf:f4	Route Request, D: 10
13	21.766355	AODV	intermediate	255.255.255.255	intermediate	00:50:fc:48:fc:db	Route Request, D: 10
14	21.766402	AODV	intermediate	255.255.255.255	intermediate	00:30:00:10:90:a4	Route Request, D: 10
15	21.766460	AODV	intermediate	255.255.255.255	intermediate	00:30:00:10:90:b2	Route Request, D: 10
16	21.766511	AODV	intermediate	255.255.255.255	intermediate	00:e0:98:aa:ad:09	Route Request, D: 10
19	21.770296	AODV	destination	intermediate	destination	00:50:fc:49:19:cf	Route Reply, D: 10.0
22	21.771297	AODV	intermediate	source	intermediate	00:30:00:10:90:a4	Route Reply, D: 10.0
25	22.777042	ICMP	source	destination	source	00:30:00:10:90:9c	Echo (ping) request
28	22.788489	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request
29	22.792334	ICMP	destination	source	destination	00:50:fc:49:19:cf	Echo (ping) reply

Figure 8.22 AODV-LL using four interfaces on the intermediate node

Figure 8.22, shows AODV-LL setting up the AODV route and starting the ICMP packet transfer. Firstly in frames 11 and 12, the intermediate node receives two RREQs from the source from the incoming and outgoing interfaces that are assigned to the same hop that the source node is on. As explained previously, for AODV to work correctly, broadcast packets would only need to be transmitted out the outgoing interfaces on each node, but by default broadcast packets are transmitted out all interfaces and so for simplicity reasons this was left unchanged. The RREQ, shown in frames 13 to 16, is forwarded on out all four interfaces on the intermediate node. Later a RREP is received back from the destination node and forwarded to the source completing the route.

Figure 8.23 shows ICMP packet transfers between the source and destination nodes as viewed by each of the four interfaces on the intermediate node. By observing the source and destination MAC address columns in *Figure 8.23*, it can be seen that AODV has correctly assigned the four interfaces, as two pairs and then within each pair, one is the outgoing interface and the other is the incoming interface. The first Ethereal capture in *Figure 8.23* shows the first incoming interface on the intermediate node receiving the ping request from the source node. Shown by the next Ethereal

1: Incoming interface from source node

No.	Time	Protocol	Source Ac	IP Dest Addr	MAC Source Addr	MAC Source Unreso	Info	MAC Dest Addr	MAC Dest Unresolv
1	0.0	ICMP	source	destinatior	AllwellT_10:90:9c	00:30:00:10:90:9c	Echo (ping) request	AllwellT_10:90:a4	00:30:00:10:90:a4
2	1.0	ICMP	source	destinatior	AllwellT_10:90:9c	00:30:00:10:90:9c	Echo (ping) request	AllwellT_10:90:a4	00:30:00:10:90:a4
3	2.0	ICMP	source	destinatior	AllwellT_10:90:9c	00:30:00:10:90:9c	Echo (ping) request	AllwellT_10:90:a4	00:30:00:10:90:a4
4	3.0	ICMP	source	destinatior	AllwellT_10:90:9c	00:30:00:10:90:9c	Echo (ping) request	AllwellT_10:90:a4	00:30:00:10:90:a4
5	4.0	ICMP	source	destinatior	AllwellT_10:90:9c	00:30:00:10:90:9c	Echo (ping) request	AllwellT_10:90:a4	00:30:00:10:90:a4

2: Outgoing interface to destination node

No.	Time	Protocol	Source Ac	IP Dest Addr	MAC Source Addr	MAC Source Unreso	Info	MAC Dest Addr	MAC Dest Unresolv
1	0.000000	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request	destination	00:e0:98:a5:cf:f3
2	1.009328	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request	destination	00:e0:98:a5:cf:f3
3	2.019881	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request	destination	00:e0:98:a5:cf:f3
4	3.029218	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request	destination	00:e0:98:a5:cf:f3
5	4.039247	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request	destination	00:e0:98:a5:cf:f3
8	5.049131	ICMP	source	destination	intermediate	00:e0:98:aa:ad:09	Echo (ping) request	destination	00:e0:98:a5:cf:f3

3: Incoming interface from destination nodes

No.	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Source Unreso	Info	MAC Dest Addr	MAC Dest Unresolv
1	0.000000	ICMP	destination	source	destination	00:50:fc:49:19:cf	Echo (ping) reply	intermediate	00:50:fc:48:fc:db
2	1.008407	ICMP	destination	source	destination	00:50:fc:49:19:cf	Echo (ping) reply	intermediate	00:50:fc:48:fc:db
3	2.018781	ICMP	destination	source	destination	00:50:fc:49:19:cf	Echo (ping) reply	intermediate	00:50:fc:48:fc:db
4	3.027993	ICMP	destination	source	destination	00:50:fc:49:19:cf	Echo (ping) reply	intermediate	00:50:fc:48:fc:db
5	4.038046	ICMP	destination	source	destination	00:50:fc:49:19:cf	Echo (ping) reply	intermediate	00:50:fc:48:fc:db

4: Outgoing interface to source node

No.	Time	Protocol	IP Source Addr	IP Dest Addr	MAC Source Addr	MAC Source Unreso	Info	MAC Dest Ad	MAC Dest Unresolv
1	0.000000	ICMP	destination	source	intermediate	00:30:00:10:90:b2	Echo (ping) reply	source	00:e0:98:a5:cf:f4
2	1.008400	ICMP	destination	source	intermediate	00:30:00:10:90:b2	Echo (ping) reply	source	00:e0:98:a5:cf:f4
3	2.018776	ICMP	destination	source	intermediate	00:30:00:10:90:b2	Echo (ping) reply	source	00:e0:98:a5:cf:f4
4	3.027987	ICMP	destination	source	intermediate	00:30:00:10:90:b2	Echo (ping) reply	source	00:e0:98:a5:cf:f4
5	4.038099	ICMP	destination	source	intermediate	00:30:00:10:90:b2	Echo (ping) reply	source	00:e0:98:a5:cf:f4

Figure 8.23 Traffic on the four interfaces on the intermediate node during an ICMP packet transfer

capture, interface two, the outgoing interface to the destination node, transmits the ping request to the incoming interface of the destination node. The ping reply is transmitted back from the destination node's outgoing interface and is received by the intermediate nodes other incoming interface as shown by the third ethereal capture. The ping reply is then transmitted out the intermediate node's other outgoing interface to the incoming interface on the source node as shown by the fourth Ethereal capture in *Figure 8.23*.

Overall the multiple interface tests showed that the AODV-LL implementation was fully capable of operating using two and four interfaces per node.

8.2 Performance Results

The performance tests compared using a hello message based AODV and a data link layer feedback based AODV on nodes using one, two and four wireless network interfaces over a range of multiple hop routes. The performance metric used was the throughput rate at the user or application layer. The tests were designed with multiple factors in order to reveal any benefits resulting from a combination of factors.

The following tables show the results of the performance tests. The results are grouped according to the AODV type used, either AODV-HM the hello message based AODV or AODV-LL the data link layer feedback based AODV, and the number of wireless network interfaces used; either one, two or four. Each table shows the throughput rate for each of the ten iterations within the test, and then the average throughput rate for each test in the bottom row. All the throughput rates are measured in megabits per second (Mbps).

The first group of performance tests used one wireless network interface per node and compared nodes using the AODV-HM implementation with nodes using the AODV-LL implementation. The results for these tests are shown in *Table 8.1* and *Table 8.2*.

For both sets of results the dramatic decrease in throughput as the hops in the route increases is obvious. When compared with the theoretical throughput rates in *Table 5.2* for a TCP transaction using

Table 8.1 Performance test results using AODV-HM with one wireless network interface

AODV-HM One Wireless Interface			
One hop	Two hops	Three hops	Four Hops
4.60	2.15	1.34	1.00
4.61	2.24	1.32	0.98
4.59	2.21	1.33	0.99
4.59	2.10	1.31	0.98
4.62	2.25	1.35	0.94
4.59	2.32	1.35	1.01
4.61	2.19	1.34	1.02
4.62	2.30	1.32	1.01
4.59	2.24	1.36	0.89
4.67	2.31	1.36	0.98
4.61	2.23	1.34	0.98

Table 8.2 Performance test results using AODV-LL with one wireless network interface

AODV-LL One Wireless Interface			
One hop	Two hops	Three hops	Four Hops
4.67	2.33	1.46	1.04
4.11	2.35	1.48	1.05
4.89	2.33	1.48	1.06
4.90	2.33	1.48	1.06
4.92	2.28	1.49	1.04
4.92	2.35	1.49	1.05
4.92	2.31	1.50	1.08
4.89	2.34	1.47	1.06
4.91	2.31	1.46	1.09
4.88	2.32	1.45	1.10
4.80	2.33	1.48	1.06

a single wireless interface, the actual values from the performance tests are lower but follow a similar trend. The actual results were expected to differ to the theoretical calculations for a few reasons. Firstly the theoretical calculations were based on an almost perfect scenario where no errors or collisions occurred and for this reason the actual results could be lower. The second reason why the actual results could differ is that the theoretical calculations were based on a simple TCP model in order to simplify the calculations, but the actual TCP model used in Linux is much more complex, as explained in *Section 5.2.2*.

When comparing the two AODV implementation types in this first group of tests, the throughput results were only slightly higher for the nodes running the AODV-LL implementation compared to the nodes

running the AODV-HM implementation. The hello messages appeared to only cause a small decrease in the overall throughput performance when using a single wireless interface. This may have been caused by the hello messages simply using network bandwidth, or more likely, from the hello messages causing collisions, resulting in packets having to be retransmitted and a decrease in the throughput..

The second group of performance tests used two wireless interfaces per node, and again all the nodes in the test were either running the AODV-HM implementation that used hello messages, or the AODV-LL implementation that didn't use hello messages. This second group of performance tests showed a more significant difference between the AODV-LL and the AODV-HM implementations and showed more clearly the improvement from using two wireless interfaces per node.

Table 8.3 Performance test results using AODV-LL with two wireless network interfaces

AODV-LL Two Wireless Interfaces			
One hop	Two hops	Three hops	Four Hops
5.01	2.96	2.15	1.66
5.01	2.91	2.15	1.81
5.02	2.89	2.15	1.79
5.04	2.94	2.16	1.82
4.99	2.93	2.16	1.80
5.02	2.95	2.15	1.81
5.04	2.95	2.14	1.84
5.04	2.95	2.14	1.80
5.06	2.97	2.13	1.83
5.04	2.95	2.14	1.83
5.03	2.94	2.15	1.80

Table 8.4 Performance test results using AODV-HM with two wireless network interfaces

AODV-HM Two Wireless Interfaces			
One hop	Two hops	Three hops	Four Hops
4.94	2.43	1.56	1.28
4.96	2.28	1.56	1.08
4.96	2.55	1.55	1.29
4.96	2.60	1.54	1.33
4.97	2.54	1.55	1.30
4.96	2.58	1.53	1.30
4.98	2.56	1.55	1.05
4.98	2.58	1.54	1.37
4.92	2.58	1.55	1.06
4.98	2.58	1.56	1.31
4.96	2.53	1.55	1.24

The results from the second performance tests are shown in *Table 8.4* and *Table 8.3.* When comparing with the single interface test results, it is interesting to note that over one hop, there is no difference in throughput when two interfaces per node are used. This highlights how over one hop, the two interface per node configuration is just the same as using single interface per node. In terms of reducing the overhead caused by the IEEE 802.11 standard, it is only between hops that using two interfaces per node can increase performance.

After one hop, the results show there is a noticeable improvement in throughput for the AODV-LL nodes over the AODV-HM nodes when using two interfaces per node. Since the AODV-HM implementation broadcasts hello messages out every interface, when using two interfaces, twice as many hello messages are being broadcast for the same amount of traffic being transferred. This could be the cause of the decreased throughput in the AODV-HM nodes due to the extra hello messages causing collisions and retransmits.

The next results (*Table 8.5* and *Table 8.6*) show using four interfaces per node for both the AODV-LL and AODV-HM nodes. Due to the number of wireless interfaces available, tests only up to two hop routes could be conducted. Over just one hop there is an improvement in the throughput for both the AODV-HM and AODV-LL nodes. Unlike the two interface per node configuration, this showed how using four interfaces per node is effective from one hop onwards. The two hop test using four interfaces had the most surprising results out of all the performance tests. These results, being much higher then the calculated theoretical results, showed clearly TCP taking advantage of the full duplex nature of the four interface configuration. It is also interesting to view that there is only a small difference between the two AODV types, showing that the presence of hello messages had little effect on the overall throughput. With four interfaces per node, there is eight hello messages being broadcast every hop , but within each hop there is now two non-interfering communication channels. Therefore within each channel there is still two hello messages being broadcast every second, like with the two interface per node configuration. But when using four interfaces per node, data is spread between two channels over each hop, since it can be transferred in both directions. Overall this means although two hello messages are transmitted every second, creating the possibility of collisions occurring or interference, there is now less data to interfere with then with the two interface configuration, since within one hop, the data is spread between two communication channels. This could be a possible reason why the hello messages don't have an effect when using four interfaces per node. Although when compared to the number of TCP segments being transferred within the tests (around 30,000

Table 8.5 Performance test results using AODV-HM with four wireless network interfaces

AODV-HM Four Wireless Interfaces	
One hop	Two hops
5.49	4.02
5.35	4.30
5.49	4.33
5.56	4.31
5.59	4.30
5.59	3.91
5.58	4.27
5.60	4.02
5.59	4.20
5.56	4.26
5.54	4.19

Table 8.6 Performance test results using AODV-LL with four wireless network interfaces

AODV-LL Four Wireless Interfaces	
One hop	Two hops
5.46	3.66
5.42	4.29
5.51	4.39
5.60	4.31
5.58	4.50
5.60	4.46
5.59	4.37
5.59	4.47
5.59	4.42
5.59	4.53
5.55	4.34

separate packets), the chances of one hello message every second causing enough disruption to result in a decrease in the throughput seems unlikely. Because of this, even though in some results such as the two interface per node tests, the AODV-HM nodes had significantly lower throughputs, it could be a possibility that this was caused by some other factor apart from hello messages, whether hardware or software related.

The following graph (Figure 8.24) summarizes the results from the performance tests and highlights the different results from the different test configurations. The performance tests results graph shows a number of features of the different test configurations. Firstly it shows clearly the large throughput increase when using four interfaces per node although there is no significant difference between the

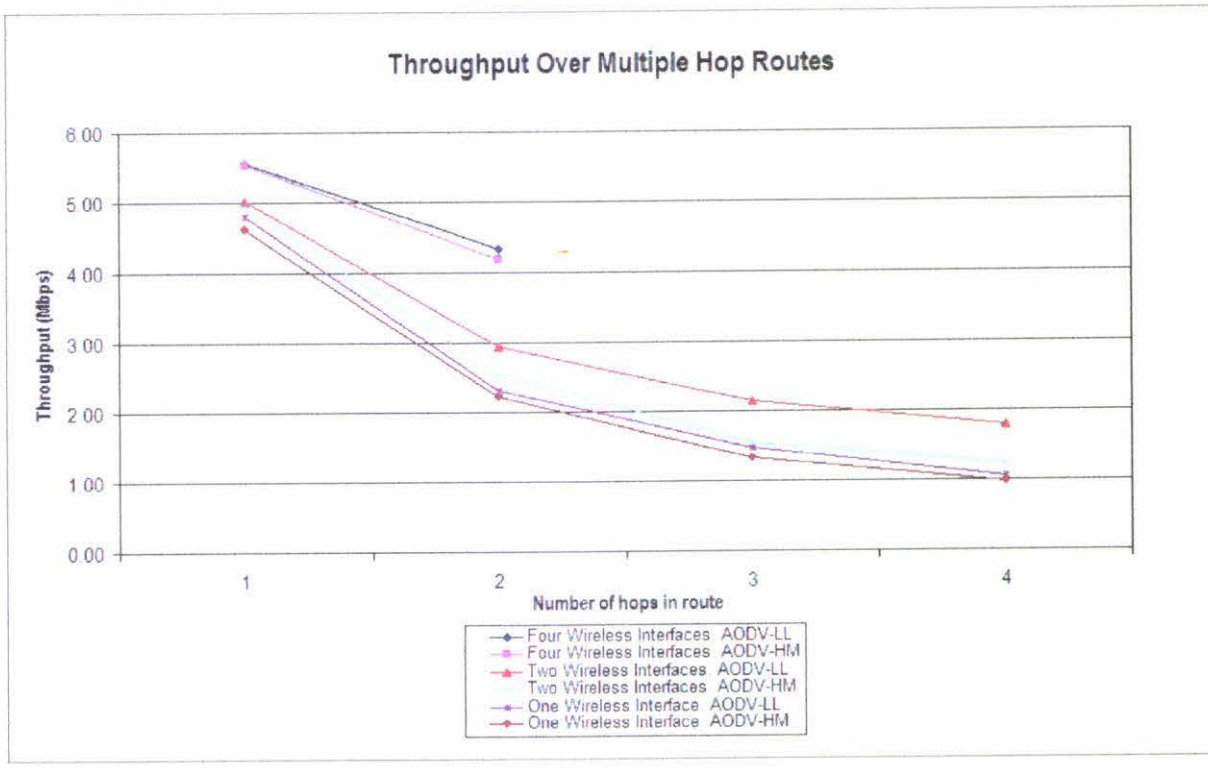


Figure 8.24 Throughput over multiple hop routes for different wireless network interface configurations and AODV implementation types

AODV-HM and AODV-LL implementations. Secondly, it shows there is a significant throughput increase, especially for two, three and four hop routes, when using two wireless interfaces per node but only with the AODV-LL implementation. The AODV-HM implementation with two interfaces per node has throughput results not much higher then when only a single interface is used. The third feature displayed by the graph is that throughput is lowest for both the AODV-HM and AODV-LL implementations when using a single wireless interface per node.

From these results is it can be concluded that using two and especially four interfaces per node results in a significant increase in throughput compared to when using just a single interface. It also appears - although this conclusion is made with less confidence - that a significant throughput increase when using two interfaces per node can only be achieved if a non-hello message based AODV implementation is used. This conclusion is made with less confidence due to the reasons explained earlier where there could be more factors then just the presence of hello messages which causes the decreased throughput and as a result further research would be needed to make a more final conclusion on this.

9 Conclusion

This research had the purpose of finding an alternative route error detection technique to hello messages and improving throughput over multiple hop routes for IEEE 802.11 based wireless networks. An alternative to hello messages was desired in order to remove problems that past research has found associated with their use, including the creation of unusable routes and the increased network traffic and overhead created by them. As a result of this research two solutions were proposed.

The proposed alternative to hello messages is the technique of monitoring if the transmit retry limit is exceeded when a packet is transmitted, resulting in the assumption that the link to the node that the failed packet was destined to is broken. When the MAC on the source IEEE 802.11 wireless interface transmits a frame containing the packet, if an acknowledgment is not received from the destination MAC, confirming reception of the transmitted frame, then the source MAC will retry transmitting the frame again. After a set number of retries, the MAC assumes the link is broken, discards the failed packet and generates an interrupt, which is received by the driver software interfacing the wireless interface and the kernel on the node. The event of the transmit retry interrupt can be monitored by modifying the wireless interface driver software, and on the occurrence of such an event, a message can be sent to the AODV implementation notifying it of the broken link.

The proposed solution to improve throughput over multiple hop routes, is to use two or four IEEE 802.11 wireless network interfaces per node. Two interfaces per node were used in a way so that a node within a multiple hop route used one interface to connect with the "downstream" node in the route , while the other interface was used to connect to the "upstream" node. Four interfaces per node resulted in two interfaces being assigned to the "downstream" node while the other two interfaces were

used for the "upstream" node. But then within each pair one interface was used solely for incoming traffic (excluding traffic below the network layer) and likewise the other interface was used only for outgoing traffic, overall creating four non-interfering communication channels. Both interface configurations decreases the time consuming (and throughput reducing) effects of the IEEE 802.11 overhead, by allowing packets to be transmitted to the next node in a route without having to wait for MAC layer acknowledgments to be transmitted back to the previous node. The four interface configuration also creates a full duplex connection across the route, which TCP can then use to its own advantage, improving throughput even more.

The driver software of the IEEE 802.11 wireless interface used was modified to be able to monitor the transmit retry limit interrupt and generate an AODV route error to send up to AODV if the interrupt occurred. An original AODV-HM (hello message based AODV) implementation was also modified to remove the hello messages and make it compatible with multiple wireless interfaces, resulting in the AODV-LL (data link layer based AODV) implementation. To test the AODV-LL implementation, functionality tests were designed, where routes were created and then broken, allowing the route error detection function to be observed. To test the multiple wireless interfaces, as well as compare the AODV-HM and AODV-LL implementations, performance tests were designed, where throughput tests were run over routes with up to four hops. For each test, all nodes either had one, two or four interfaces each, and also during each test, all the nodes were either running the AODV-HM or AODV-LL implementation.

The functionality tests showed that monitoring the transmit retry limit interrupt was an effective technique to detect broken routes. It was found it was important to set a suitable transmit retry limit for the network as part of these tests, else simple network disruptions could cause the transmit retry limit to be exceeded, and links incorrectly reported as broken. Features of the data link layer based detection technique was that detecting a broken route was very fast, less then a second, whereas the AODV-HM implementation by default took at least two seconds. Another feature of using the transmit retry limit is that a broken link will go undetected until there is an attempt to transmit a frame across it. This behavior suits the on-demand nature of AODV and, as explained in the thesis, this behavior doesn't give the AODV-HM implementation any advantage over the AODV-LL implementation due to the default settings of the AODV standard.

The performance tests showed a definite throughput improvement when using two interfaces per node

and an even more impressive improvement when using four interfaces, even though the four interface tests could only be performed over routes of up to two hops due to the available test equipment. The tests showed that providing a full duplex connection increased throughput by allowing the TCP segments to travel across the route simultaneously in both directions and increased throughput performance by allowing a significant amount of the IEEE 802.11 overhead to operate without increasing the overall transaction time. The difference between the AODV-HM and AODV-LL implementations appeared only significant in the two wireless interface per node tests. The results for the AODV-HM implementation when using two interfaces per node, were only marginally better then when using a single interface per node, showing that the benefits from using two interfaces with AODV-HM was negated by the increased hello messages. The effect of hello messages on throughput is an area of future research, as hello messages could have a much greater effect in a network made up of a larger number of hello message based AODV nodes.

Overall it can be concluded that using the transmit retry limit is an effective alternative to hello messages and it is possible to implement a non hello message based AODV implementation using off-the-shelf hardware and without any modification to the IEEE 802.11 standard. The use of multiple wireless interfaces is also a technique that requires no modification to the IEEE 802.11 standard; can be easily implemented; and provides significant throughput improvements, especially for TCP based network connections.

AODV is a routing protocol well suited to multiple hop wireless mesh networks. The non-existence of alternatives to hello messages for detecting broken routes, as well as the dramatic throughput decrease over multiple hop routes in IEEE 802.11 wireless networks, may have been reasons contributing to the slow uptake of AODV in real world applications. This research has shown that alternatives to hello messages is easily possible and that the multiple hop route throughput decrease can be minimized by using multiple wireless interfaces. It is hoped that as a result, this research will help promote the use of AODV and wireless mesh networks in real world scenarios.

10 Future Research

There a number of areas that could provide interesting future research and help provide even more conclusive results. One major area is testing with a much larger number of nodes, in a more realistic environment, such as outdoors within a university campus. Testing with more nodes and outdoors, will not only represent real world scenarios much more then the tests preformed in this research, but it may also be easier to observe any negative effects caused by hello messages and highlight the advantages of using a non-hello message based AODV implementation.

It would be interesting also to measure more metrics then simply throughput, including processor utilization by the AODV implementation, and also power usage. Both of these areas require more sophisticated hardware and software to measure then was available for this research.

Another area of future research is investigating whether the significant increase of the AODV-LL implementation over the AODV-HM implementation when using two interfaces per node is caused by the presence of hello messages or whether it is caused by other factors.

11 References

1 M. Sinclair, F. Al-Ali and G. Punchihewa. "Wireless Mesh Networking: an Investigation into AODV Routing". In *Projects*, College of Sciences, Massey University, New Zealand. November, 2003.

2 C. Perkins, E. Belding-Royer and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561. July, 2003.

3 C. Tschudin, H. Lundgren and E. Nordstrom. "Embedding MANETs in the Real World". In *Proc. of the 8th Int'l Conference on Personal Wireless Communications (PWC 2003)*, , Venice, Italy. September, 2003.

4 Locust World. Locust World: The Information Revolution. Available at http://www.locustworld.com. October, 2005.

5 L. Klien-Berndt. Kernel_AODV v 2.2.2. NIST. Available at http://w3.antd.nist.gov/wctg/aodv_kernel/. July, 2004.

6 E. Nordström, B. Wiberg and H. Lundgren. AODV-UU v 0.8.1. Uppsala University. Available at http://user.it.uu.se/~henrikl/aodv/. July, 2004.

7 Intel Corporation. AODV for Microsoft Windows v 0.1.14. Intel Corporation. Available at http://moment.cs.ucsb.edu/AODV/aodv-windows.html. July, 2004.

8 B. Gupta. AODV-UIUC v 0.5. University of Illinois, Urbana-Champaign. Available at http://sourceforge.net/projects/aslib/. July, 2004.

9 I. Chakeres and E. Belding-Royer. "AODV Routing Protocol Implementation Design". In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, IEEE Computer Society, Washington, DC, USA, Vol.7, pp.698-703. 2004.

10 H.Lundgren, E.Nordström and C.Tshudin. "The Gray Zone Problem in IEEE 802.11b based Ad hoc Networks". In *ACM SIGMOBILE Mobile Computing and Communications Review*, ACM Press, New York, USA, Vol.6. July, 2002.

11 I.Chakeres and E Belding-Royer. "The Utility of Hello Messages for Determining Link Connectivity". In *The 5th International Symposium on Wireless Personal Multimedia Communications*, Vol.2, pp.504-508. October, 2002.

12 J. Broch, D. Maltz, D. Johnson, Y. Hu and J. Jetcheva. "A performance comparison of multi-hop wireless ad hoc network routing protocols". In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, ACM Press, New York, USA, pp.85 - 97. 1998.

13 E. Belding-Royer, L Sung-Ju and C Perkins. "The effects of MAC protocols on ad hoc network communication". In *Wireless Communications and Networking Conference, 2000 (WCNC 2000)*, Vol.2, pp.543-548. 2000.

14 J. Novatnack, L. Greenwald and H. Arora. "Evaluating Ad hoc Routing Protocols With Respect to Quality of Service". In *Under review*.

15 J. Novatnack, L. Greenwald and H. Arora. "Impact of Ad Hoc Routing Protocols on Pervasive Computing". In *Under Review*.

16 L. Klein-Berndt and I. Chakeres. "AODVjr, AODV Simplified". In *ACM SIGMOBILE Mobile Computing and Communications Review*, ACM Press, New York, USA, Vol.6, No.3. July, 2002.

17 J. Postel. Transmission Control Protocol. RFC 793. September, 1981.

18 R. Braden. Requirements for Internet Hosts -- Communication Layers. RFC 1122. October, 1989.

19 W. Stevens. TCP Slow Start, Congestion Avoidance,Fast Retransmit, and Fast Recovery Algorithms. RFC 2001. January, 1997.

20 D. Clark. Window and Acknowledgment Strategy in TCP. RFC 813. July, 1982.

12 Appendices

Included in the appendices is the AODV-LL implementation code and the code for the modified ADM8211 chipset driver. All code is written C and intended for the Linux v 2.4 kernel. Due to the size of the code, rather then being included in this thesis, it is instead included on a CD-ROM. The following outlines what is included on the CD-ROM.

Appendix A

The modified ADM8211 chipset driver code. Areas of code that are modified are outlined by comments, the main section being lines 850 to 971 in the adm8211.c file.

Appendix B

The AODV-LL implementation based on the original Kernel AODV from NIST by L.Klien-Berndt. Areas of the original code that has been modified are commented.