10-2006

MANET protocol stack testing using wireless emulation platform

Amish Rughoonundon

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact rit scholarworks@rit.edu.
MANET Protocol Stack Testing Using Wireless Emulation Platform

by

Amish Rughoonundon

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Fei Hu

Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
October 2006

Approved By:

Fei Hu

Dr. Fei Hu
Primary Advisor – R.I.T. Dept. of Computer Engineering

Shanchieh Yang

Dr. Shanchieh Yang
Secondary Advisor – R.I.T. Dept. of Computer Engineering

James Minseok Kwon

Dr. James Minseok Kwon
Secondary Advisor – R.I.T. Dept. of Computer Science
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: MANET Protocol Stack Testing Using Wireless Networking

Emulation Platform

I, Amish Rughoonundon, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

A. Rughoonundon
Amish Rughoonundon

10/26/06
Date
Dedication

This thesis is dedicated to my mother and father, Sheila and Bhaskarnath Rughoonundon who have always believed that I would be able to achieve my goals in life and provided me with every means to do so.
Acknowledgements

I would like to thank my main advisor, Dr. Fei Hu for providing me with enough guidance to come up with this thesis topic as well as providing the financial support for obtaining the components needed to achieve my thesis goals. I would also like to thank him for taking the time to explain to me some of the intricacies related to working with wireless protocols. I would like to thank Dr. Shanchieh Yang and Dr. James Minseok Kwon for their help in analyzing the weaker part of my thesis and for providing suggestions in improving them. Many thanks to Mr. Richard Tolleson for helping out with obtaining the hardware needed for this thesis.
Abstract

Wireless networks have become more important in today’s world of hassle free communication. These networks have been used increasingly in critical applications in the medical field and in the army. The ability to remotely reprogram the devices has been especially important if a user did not have easy access to the device. Reprogramming required the ability to route the packet properly as well as reduce packet loss to a minimum. Such protocols would be used with critical applications and would need to be tested in an environment as close to the real one as possible. This thesis consisted of the creation of a wireless emulation platform on which a transport protocol Pump Slowly Fetch Quickly (PSFQ) would be tested and evaluated. A routing protocol, Ad-Hoc On-Demand Distance Vector (AODV), enhanced with the Skipjack cryptographic algorithm was used to test the emulator and ensure that the emulator could emulate the mobility of nodes virtually. This was to make sure that the emulator allowed real tests to be carried out in real time. The PSFQ transport protocol enhanced with AODV was tested with the emulator to observe the effect of using a real wireless device and movement emulation on the protocol performance. The results showed that PSFQ was very stable even with a high degree of emulated movement as well as high link error rate. The results also revealed that even though the emulator did not emulate all the aspects of a mobile wireless medium, the ability to use a real wireless device provided additional constraints that needed to be taken into consideration to attain the targeted end-to-end packet error rate.
# Table of Contents

Chapter 1  Introduction ........................................................................................................... 1

Chapter 2  Background ............................................................................................................ 4

2.1. Emulator ......................................................................................................................... 4

2.2. Hardware ........................................................................................................................ 6

2.3. Communication protocol ............................................................................................... 8

Chapter 3  Mobile Node Emulator (MoNoE) Design ............................................................. 16

3.1. Overview ........................................................................................................................ 16

3.2. Runtime DLL loading ..................................................................................................... 19

3.3. Distance filter design .................................................................................................... 20

3.4. Error Insertion Filter .................................................................................................... 23

3.5. Packet recognition and reconstruction design ............................................................. 24

3.6. Statistic collection ......................................................................................................... 29

3.7. Application transfer engine ......................................................................................... 29

Chapter 4  Skipjack-based MANET routing layer security .................................................. 33

4.1. AODV Outline ............................................................................................................... 33

4.2. AODV Implementation ................................................................................................. 34

4.3. AODV Modifications from original protocol ............................................................... 40

4.4. AODV PseudoCode ...................................................................................................... 41

4.5. Skipjack outline ............................................................................................................ 48

4.6. Skipjack implementation .............................................................................................. 48

Chapter 5  Transport layer design ........................................................................................ 52

5.1. Pump Slowly Fetch Quickly (PSFQ) outline ................................................................. 52
List of Figures

Figure 1: Packet delivery ratio of different network protocol on different simulators. .... 11
Figure 2: End-to-end delay of different network protocol on different simulators. ....... 12
Figure 3: Routing overhead of different network protocol on different simulators........ 13
Figure 4: MoNoE modules interconnection ......................................................... 18
Figure 5: Emulator packet content ..................................................................... 24
Figure 6: State machine for re-constructor function .............................................. 27
Figure 7: MoNoE program .................................................................................. 30
Figure 8: Fields of RREQ message ..................................................................... 34
Figure 9: Fields of RREP message ..................................................................... 36
Figure 10: Fields of HELLO message ................................................................. 37
Figure 11: Fields of DATA message ................................................................... 37
Figure 12: Rule A ................................................................................................. 48
Figure 13: Rule B ................................................................................................. 49
Figure 14: Rule A⁻¹ ............................................................................................ 49
Figure 15: Rule B⁻¹ ............................................................................................ 49
Figure 16: G-permutation diagram ..................................................................... 50
Figure 17: Cipher Block Chaining mode ............................................................ 51
Figure 18: Probability of end-to-end successful data delivery ......................... 53
Figure 19: Message implosion .......................................................................... 54
Figure 20: Number of packets lost versus size of packets ............................... 72
Figure 21: Overhead percentage versus packet error rate per link .................. 61
Figure 22: End-To-End Packet error rate versus packet error rate per link ........ 62
Figure 23: Destination node 4 moving closer to sender node 1.................................63
Figure 24: Packet error rate versus Node movement.............................................64
Figure 25: Skipjack encryption and decryption.......................................................67
Figure 26: End-To-End Packet error rate VS packet error rate per link 1 hop away....69
Figure 27: End-To-End Packet error rate VS packet error rate per link 2 hops away.....69
Figure 28: End-To-End Packet error rate VS packet error rate per link 3 hops away......70
Figure 30: Packet error rate VS medium error rate..................................................74
Figure 31: 4 nodes in diamond formation. ...............................................................75
Figure 32: Results for 4 nodes in diamond formation..............................................76
Figure 33: 4 nodes in diamond formation with interconnection...............................76
Figure 34: Results for 4 nodes in diamond formation with interconnection.............77
Figure 35: Comparison of packet error rate with different formations.......................78
Figure 36: Comparison of packet error rate with different formations.......................78
Figure 37: PSFQ over AODV.................................................................................80
List of Tables

Table 1: Effect of Skipjack on AODV packets.................................................66
Glossary

**AODV**
Ad-Hoc On-Demand Distance Vector Routing is a reactive routing protocol that uses a routing table to keep routes to different locations.

**PSFQ**
Pump Slowly Fetch Quickly is a transport protocol that uses hop-by-hop packet recovery to send packets across multiple hops.

**MANET**
Mobile Ad-Hoc Network consists of hardware devices that can form a network without any single controlling entity.

**Node**
A node is a device that forms part of a Mobile Ad-Hoc network.

**Testbed**
A testbed consists of multiple devices that form a realistic network.

**WSN**
Wireless Sensor network consists of multiple sensing nodes that observe and relay data to a main node regarding their environments.

**TCP/IP**
Transport control protocol/Internet protocol make up the transport and network protocol used to transfer Ethernet packets.

**Emulator**
A program that creates a real environment virtually is called an emulator.

**WLAN**
Wireless Local Area Network is a network consisting of wireless devices.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>Medium Access control is the protocol used by devices to access the communication medium.</td>
</tr>
<tr>
<td>DSSS</td>
<td>Direct Sequence Spread Spectrum is a way to transmit data whereby the data stream is combined with a higher data rate bit sequence. This increases the signal’s resistance to interference.</td>
</tr>
<tr>
<td>RS-232</td>
<td>RS-232 is a serial protocol used for communicating with external devices using a serial port.</td>
</tr>
<tr>
<td>DSR</td>
<td>Dynamic Source routing is an on-demand routing protocol that performs route discovery and route maintenance.</td>
</tr>
<tr>
<td>DSDV</td>
<td>Destination-Sequenced Distance-Vector is a proactive routing protocol where each node maintains routing information for all known destinations.</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library is a piece of external code that can be loaded at runtime by an application to perform certain tasks.</td>
</tr>
<tr>
<td>CBC</td>
<td>Cipher Block Chaining is used to make successive encrypted packets dependent on the previous encrypted packet.</td>
</tr>
<tr>
<td>IV</td>
<td>Initialization Vector is a string of bytes used in Cipher Block Chaining to initiate the encryption chain.</td>
</tr>
<tr>
<td>KEA</td>
<td>Key Exchange Algorithm is a protocol used by devices to exchange encryption keys for secure communication.</td>
</tr>
</tbody>
</table>
**Throughput**

Throughput is the number of packets that can be injected into the network.

**Goodput**

Goodput is the amount of data packets that are leaving the network.
Chapter 1  Introduction

Wireless networks have been used more and more in the past few years. Many wireless standards such as Bluetooth [17], ZigBee [14] and WiMax [16] have shown that wireless networks would be a common aspect of everyday use in the near future. With the use of wireless in critical applications and in remote areas, it has become increasingly important to have the ability to develop and test protocols using the real hardware devices. In this thesis one particular aspect of using hardware devices in remote areas and critical applications was evaluated; the ability to re-program the wireless devices remotely. The wireless network that was used in the evaluation was called a Mobile Ad hoc network (MANET). A MANET is formed whenever two or more mobile devices form a self-configuring network, without any single controlling entity. A communication protocol is used by each entity in a MANET to communicate with other entities and maintain the network. The communication protocol is usually described by the Open Systems Interconnection Reference Model (OSI) model. 7 separate layers make up the OSI model. Specific tasks such as communication or maintenance are usually assigned to each layer. Among the 7 layers, the network and transport layer play significant roles in providing efficient secure communication between devices sometimes across multiple hops.

Improvement in wireless routing and security protocols are constantly being proposed and thorough testing has been required to check that the new wireless protocol behaved better than its predecessors or at the very least was error free. The most common kind of experimentation was the simulation of wireless protocol using network simulation tools such as Opnet, ns-2 and OMNET++. Although these software-based simulations did
provide a good background on the possible advantages of wireless protocols, they did not cover all the problems that may be encountered in the real world. To complement these simulations, experiments on real devices needed to be carried out to obtain a correct idea of the advantages and disadvantages of different wireless solutions in different kind of environments using different wireless devices.

On top of simulators, other testing environments have been used such as emulators and testbeds. Real hardware testbeds used were expensive and required a lot of man power and man hours to run experiments on. Some of the experiments already done in the past used Remote controlled cars to move the nodes around [4] and some of them used humans carrying the nodes following a certain pre-planned route. These types of experiments were not feasible on a small scale. The next best thing was emulation. In this case, emulation would mean recreating the effect of moving wireless nodes. The emulator would also have to be inside a small enclosed area for convenience purposes. Since real movement cannot be replicated exactly, it would have to be simulated somehow in software. The real wireless nature of the communication could still be obtained by using a wireless radio.

This thesis was part of a National Science Foundation (NSF) funded project. It consisted of creating an emulator on which a protocol stack could be tested with real hardware devices and to observe the performance of the protocol with virtually moving nodes. The protocol stack would consist of a transport layer protocol based on the implementation of a modified Pump Slowly Fetch Quickly (PSFQ) [12] algorithm and a network layer protocol based on the Ad-hoc On-demand Distance Vector (AODV) [13] algorithm enhanced with a security algorithm. AODV was first tested with the emulator. This was to make sure that the emulator was bug free and that it’s performance was
adequate to work with communication protocols. The PSFQ protocol was then used by itself on the emulator to observe the effect of virtual node movement and real hardware device on the performance of the protocol. After performing some adjustment to the PSFQ protocol to allow it to perform better with the hardware, the AODV and PSFQ protocol were combined and tested to observe if there was any benefit to using the two together. A conclusion was then drawn as to whether the goals of this thesis were achieved. This was followed by a discussion of some of the improvements to the emulator that could be done in the future.

Chapter 2 will discuss some of the previous work done in creating emulators and low data rate network protocols. Chapter 3 will provide a more in-depth view of the design methodology, implementation and analysis of the Mobile Node Emulator (MoNoE). Chapter 4 will discuss the details of the AODV implementation and the possible security algorithms that were considered as well as provide more details on the implementation of the algorithm chosen. Chapter 5 will discuss the use of PSFQ as a transport layer protocol and the modifications that were required to achieve our goal of stability in the communication link. Chapter 6 will analyze the data obtained from using the protocol on the emulator and discuss the feasibility and advantages of using such a testing environment. Chapter 7 will provide a conclusion as to whether our goals were achieved or not. Finally in the future work section, an analysis of possible improvements that could be done on both the network emulator and communication protocol will be discussed.
Chapter 2  Background

2.1. Emulator

Testbeds and emulators are additional tools that can be used in testing a communication protocol after running it through a simulator. A testbed is defined as a platform on which a range of experimental products can be deployed and allowed to interact in real-time. Testbeds provide the best possible solution to real time testing of communication protocols but they do carry a heavy price. Testbeds that have been used in the past have shown that they were difficult to set up and some of the experiments could not be reproduced faithfully. Testbeds have also been expensive in terms of the monetary cost of setting them up. Most of the testbeds currently in use cost thousands of dollars and are so complex that they require a lot of man power to perform one experiment [4].

The wireless network’s performance has always been highly dependent on the location where the network has been used. It would therefore be advantageous if the tests on the communication protocol could be carried out in the environment where the protocol would be primarily used. This was especially important if the communication protocol would have been being developed for sensitive areas such as hospitals, disaster zones or for military applications. Moving a testbed to different locations was very difficult due to the vast amount of hardware used to perform the experiment.

Emulators although not the best solution did provide advantages as compared to testbeds. They cost less to setup and use, they take much smaller space than testbeds and the manpower needed to operate an emulator is much less than a testbed. They could
therefore be easily moved to other locations to check the effect of a real environment on the wireless communications. There have been a number of emulators created over the years for wireless network testing; JEMU [1] which is a radio replacing emulator and MobiEmu [2] which uses packet filtering to emulate mobility. Although all the emulators were different in their architecture, they all had the following things in common:

a) All of them used Linux as their base Operating System to run the emulation software on.

b) They all used WLAN 802.11 as their MAC and physical layer.

c) TCP/IP was the transport and network protocol of choice when designing the emulator.

d) They all used IP chains to virtually kill the link between different nodes.

IP Chains have been a set of rules in Linux that allowed the operating system to filter IP packets based on the content of the packet such as MAC address, source IP address or destination IP address. The two issues with this kind of architecture was that 802.11 implements a high data rate wireless protocol which might not exist for all Ad-Hoc networks and the emulator software was restricted to being used under the Linux operating system.

For this thesis, an emulator was designed that would be highly reconfigurable and provide a user with the ability to modify the emulator to their specific needs. The emulator was created to run on the Windows operating system to differentiate it from the common trait of past emulators. Since ip chains are not available in Windows, a new type of packet filter was created to virtually kill the wireless links between the hardware
devices. The emulator was not created to be used with a single hardware device. This would provide the user with the ability to test a protocol with multiple hardware devices.

In any design, a balance is needed with respect to research cost and realistic performance. Using an emulator would be much cheaper than a testbed and but it would still offer the ability to perform experiments using real hardware devices. Since the focus of this thesis was two-fold; Protocol implementation and Emulator Design, not all aspects of a real wireless environment would be emulated. Even with this partial emulation capability, the emulator would still be more effective for certain tests than just using a simulator. Testing of the code with different real hardware devices would be possible at different stages of the application development. It would also allow testing the protocol in different locations to observe the environmental effects on the protocol performance.

2.2. **Hardware**

Since this thesis would deal with low data rate network, two low data rate radio devices were considered to be used in this research:

a) TelosB motes from XBow.

b) XBee-Pro RS-232 RF modem from MaxStream.

TelosB motes are highly reconfigurable devices that consist of processor and low data rate radio boards. The board could also be customized with different sensory devices. A more in depth specification of TelosB can be found below:

- IEEE 802.15.4 compliant.
- 250 kbps, High Data Rate Radio.
- TI MSP430 microcontroller with 10kB RAM.
- Integrated onboard antenna.
• Data collection and programming via USB.
• Open-source operating system.

XBee-Pro is an off the shelf RF modem. Much of the modem’s settings are easily reconfigurable for different power levels as well as different channel frequencies by using a user-friendly software that comes with the device. A more in depth specification of XBee-Pro can be found below:

• Zigbee/802.15.4 compliant RF modem using ISM 2.4 GHz.
• RF maximum data rate of 250,000 bps.
• Uses Direct Sequence Spread Spectrum for modulation.
• RS-232 interface.
• Interface maximum data rate of 115,200 bps.
• Range of a 100 m indoor.

Both radio devices had more or less the same hardware settings. Having used the TelosB motes in the past, I knew that it was tricky for a user to reconfigure the programs running on the motes. Any little change to the hardware needed such as having the radio run at different power levels or changing the channel frequency would have required a very good understanding of the underlying code and considerable testing. The XBee-Pro provided a more easily reconfigurable device. Since this thesis’s main focus was on the emulation and protocol testing, it was not necessary to use a device that would require considerable effort to customize. This would allow more time to be able to focus on the emulator and communication protocol design. The only downside to using the XBee-Pro modem was that the MAC layer was fixed. In the future it might be advantageous to use
the TelosB mote and therefore have the ability to modify the MAC layer to comply with different kind of tests. Another factor influencing this decision is that I had used RS-232 a lot in the past and the learning curve when writing a program to communicate with the device would be very small as compared to learning how to communicate using USB.

2.3. Communication protocol

The main part of this research was to determine how communication protocols behave in low data rate environments and if it was possible to use a secure and stable protocol in noisy environments using the hardware described above. It was critical that these protocols be tested in an environment as close to the real thing as possible and therefore an emulator was chosen to be the testing environment for the communication protocol. There has been multitude communication protocols developed in the past. At the network layer, protocols such as Destination-Sequenced Distance-Vector (DSDV), Ad hoc On Demand Distance Vector (AODV) and Dynamic Source Routing (DSR) have been used regularly.

DSDV used a hop-by-hop distance vector routing protocol where each node maintained a routing table with information about the next hop to a destination, the number of hops and the sequence number of the current route. This sequence number provided the advantage that DSDV guaranteed loop-freedom when sending packets. DSR used source routing as a means to transfer data. Each packet contained the complete route towards a particular destination. The main advantage of this protocol was that it incurred less overhead since nodes do not need to keep and maintain routing tables. The disadvantage was that packet size was greatly increased with the inclusion of the
complete routing information. AODV was then designed and contained the best of both of the previous routing protocols. It used DSDV’s hop-by-hop routing, sequence numbers and periodic beacons with DSR’s on-demand mechanism for route discovery and maintenance.

There have been many research papers comparing these protocols to each other in different scenarios. Since results using different simulators differ greatly as stated by Cavin D. et al [21], it was only fair to compare results from different papers that use different simulators but with more or less the same scenario. In this case, 4 papers [8,9,10,11] were considered. Altogether they provided a good background on choosing the appropriate network protocol for stability.

When dealing with stability, packet delivery ratio became of utmost importance. In critical medical applications, it would be more beneficial that all packets sent reach their destination without needing to retransmit the information. The results obtained from the different papers are showed in figure 1. In the case of DSDV and DSR, the throughput actually decreased greatly with an increase in topology change. It can be observed that AODV does not have the best performance in all cases but its performance is very stable, oscillating very little with increase in movement speed of the nodes.

The second important aspect that needs to be considered is to minimize the time the protocols took to locate routes towards a particular destination and minimize the time it took for a packet to reach its destination. The packet delay for different simulation results are shown in figure 2. As expected DSDV and DSR showed varying end-to-end delay as the speed of the nodes was increased. AODV had a more or less stable end-to-end delay although it looks as if it is highly dependent on the simulator used. The Qualnet
simulator for example actually showed that AODV had the worst end-to-end delay among the three.

Since some of the latency in transferring data can be accounted for by the routing overhead, a network protocol with a minimum routing overhead would be better suited for our purpose. Routing overhead simulation results are showed in figure 3. DSR had the lowest amount of routing overhead. This was expected since the nodes are not required to maintain any routing tables. DSDV's and AODV's routing overhead increased as movement speed increased. This was unavoidable since more links were broken and more requests were sent on the network for new link information.
Using ns2 simulator. [11]

Using ns simulator. [8]

Using ns2 and glomosim simulators. [10]

Using Qualnet simulator. [9]

*Figure 1: Packet delivery ratio of different network protocol on different simulators.*
Figure 2: End-to-end delay of different network protocol on different simulators.
Using ns2 simulator. [11]

Using ns simulator. [8]

Using Qualnet simulator. [9]

**Figure 3: Routing overhead of different network protocol on different simulators.**

Figure 1, 2 and 3 were only a small fraction of the amount of information available but they did provide enough supporting evidence to show that AODV matched most of the criteria required in our case. The throughput was more or less constant and the end to end delay was not the worst among the three and did not change much in some cases. The only issue that would have to be addressed was the routing overhead.
Protocols such as Pump Slowly Fetch Quickly (PSFQ) [12], Reliable Multi-Segment Transport (RMST) [18] and Sensor Transmission Control Protocol (STCP) [22] were relatively new protocols and were mostly geared towards specialized applications. Some were more specialized in optimizing upstream data reliability (RMST and STCP) and some optimized downstream data reliability (PSFQ). In our case though, it was more beneficial to use a protocol that was specialized to achieve a certain low end-to-end error rate rather than a general protocol.

Although a lot of research was available on all the communication protocols above, most of them had been carried out using wireless sensor networks. On top of that, an accurate comparison of the different protocols could not be found in all the literature I reviewed so far. In the case of this thesis, PSFQ was chosen as the transport protocol that was tested on the emulator. The main reason for choosing this protocol was that in some applications, it might be critical that once a device is deployed, the software running on the device can be upgraded with ease remotely. PSFQ was optimized for downstream data reliability. This meant that the protocol provided some kind of assurance that if data was sent from a master node to slave nodes, the data would reach all the required slave nodes with a minimal number of retransmissions. Reliability was especially important if some algorithm or code was broken down into multiple chunks of data and sent to the slave nodes. The reconstruction of the algorithm would require that the chunks are used in order. Any missed data might cause the master to have to send all the chunks again. Since this protocol was mostly optimized for downstream data reliability, a node would also require some other kind of protocol for upstream data reliability. This would be left as future work of this thesis.
Two types of security algorithms have been symmetric and asymmetric key algorithms. In symmetric key algorithms, both sender and receiver shared the same key whereas in asymmetric key algorithm, the sender's key was secret whereas any receiver with the appropriate public key could decrypt the data. One of the main advantages of symmetric key algorithms was that it was not as computationally intensive as asymmetric key algorithms. This was the primary reason why such algorithms are widely used in WSN for example. One example was TinySec which used Skipjack as one of the encryption and decryption algorithm it offered. The creators of TinySec evaluated both Skipjack and RC5 and their results showed that Skipjack was less resource greedy and still offered the same amount of security as RC5 [24]. On top of that other research have shown that the Skipjack algorithm will not be broken any time soon as stated in a review of the security algorithm; “A more speculative attack using a future, hypothetical, massively parallel machine with 100,000 RISC processors, each of which was capable of 100,000 encryptions per second, would still take about 4 million years.” [25]. To this day, only 31 of the 32 rounds of the algorithm have been broken. This showed that this security algorithm was very stable and that it was worthwhile to implement it in our design since most of the communication protocol nowadays comes with some security enhancement.
Chapter 3  Mobile Node Emulator (MoNoE) Design

3.1. Overview

There are many emulators that have already been created. Many people would argue that there was no need for one more. Most emulators used common features such as ip chains and all of them assumed that WLAN 802.11 would be used at the physical level. Since they all used ip chains, the emulators were restricted to be run under the Linux operating system or through a Linux emulator under the Windows operating system.

The goal of this design was to create an emulator that can work with any device at the physical level. As long as the communication between the device and the emulator was kept the same, the emulator could be able to function with any current wireless devices. The emulator would run under the windows operating system which would differentiate it from other emulators created before. The final product would be simple and self sufficient such that it could be moved between different computers with different configurations. The emulator was set up to model the transmission range between different nodes as well as the error rate per link in a network. The MAC and Physical layer of the communication protocol would be made up by the real RF modem used.

Since the design of an emulator was to be as modular as possible, the emulator was created as a Dynamic Link Library (DLL). In this way if any part was changed, a whole re-write of the emulator was not needed. It would also provide some kind of security against accidental tampering of the main code by users of the emulator.

The breakdown of the emulator modules is showed in figure 4. The filter modules as well as the serial communication module were designed to all be DLLs. This was done
so that they can be interchanged without the need to modify the emulator code that held everything together. The emulator itself was another DLL. Any application that would use the DLL would need to instantiate, initialize and then use the DLL as is. In this case, the RS232 communication protocol was used to transfer data from the emulator to the wireless devices. In the future USB or I²C protocols could be used instead to transfer the information much faster. The only requirement would be to create a new module for them and replace the serial communication module. The filters shown in figure 4 are special modules that were designed to perform various tasks on the packets being passed through them. For example, the distance filter would compute the distance between two nodes using information in the packet and decide if the packet should be silently dropped or passed onto the next filter. An additional filter called errorInsertion was also created that allowed a user to inject a certain percentage of incoming packets with errors. If a real wireless environment was tested, error injection would not have been necessary but it did provide a means to test the protocols under varying errors in the incoming packets. Note that the speed filter was not created in this thesis and was only shown in figure 4 as an example.
Higher layers: Link, Network, Transport and application layers

EMULATOR

Send Packet

Other filter

Send Packet

ErrorInsertion filter

Send Packet

(Speed Filter)

Send Packet

Distance Filter

Send Packet

Packet reconstruction

Packet tagging

Send Packet tagged with header and footer

Serial communication module

Send Bytes

Send Packet

Read Bytes

Write Bytes

Communication Hardware

Figure 4: MoNoE modules interconnection
3.2. Runtime DLL loading

More filters can be created and added without the need to modify the emulator. This was done by using a text file that would tell the emulator which DLL needed to be loaded at runtime. The text file named `incomingFilters.data` contained lines with the name of each filter on one line. The filter names should be entered in the order that they would filter the data. For example `incomingFilters.data` could contain the following lines:

- `errorInsertion`
- `distanceFilter`
- `speedFilter`

This would tell the emulator to look for three DLL files named `errorInsertion.dll`, `distanceFilter.dll` and `speedFilter.dll` in the same folder as the emulator DLL. It would then try to initialize each DLL. If a DLL failed it would be unloaded without terminating the emulator. If no DLL was loaded, the emulator would basically pass through any data from the physical device to the application.

If all three DLLs were loaded, whenever a packet was received from the physical device, it would first be sent to the first filter that was loaded. In this case it would be the `errorInsertion.dll`. In this case, the filter would modify the packet inserting errors in random places. It would then be passed to the `distanceFilter.dll`. This filter would calculate the distance between the sender and receiver node. If the distance was greater than the range of the sender node, the packet would be silently dropped. If not, it would be passed to the `speedFilter.dll`. This process would go on until the packet was passed to all filters. If the packet was not dropped in the meantime, it would then be passed to the
application. Although these slots should be used by filters, they could be used as statistic collectors as well if any special statistics needed to be collected by the application.

3.3. **Distance filter design**

The distance filter used a file called distanceFilter.data similar to the ns2 simulation file to determine the position of each node in the virtual landscape. The content of one such file is shown below:

```
Number of nodes = 4
Simulation time = 60
1,0,0=5,0,60,0,0
2,10,0=5,0,5,270,1=5,6,10,90,1=5,11,15,270,1=5,16,20,90,1=5,21,30,270,1=5,31,40,90,1=5,41,50,270,1=5,51,60,90,1
3,0,10=5,0,5,180,1=5,6,10,0,1=5,11,15,180,1=5,16,20,0,1=5,21,30,180,1=5,31,40,0,1=5,41,50,180,1=5,51,60,0,1
4,0,-10=5,0,5,0,1=5,6,10,180,1=5,11,15,0,1=5,16,20,180,1=5,21,30,0,1=5,31,40,180,1=5,41,50,0,1=5,51,60,180,1
```

The first line was added to allow a human user to understand the content of the file. The filter would first bypass the first line and then use the second line to know how many nodes would be in use during the current emulation. The filter would use the third line to obtain the total simulation time. Note that all the times were set in seconds.

The next 4 lines contained the information for each node in the emulation. The line for each node contained information on the node identification number followed by the start position of the node in x and y coordinates. Alas only positive x and y coordinates could be used as the start position. The first number after the equal sign was the virtual transmission range of the node. This had to be greater or equal to 0. The number could be used to emulate real wireless communication where the transmission range of 1 node might be greater than the transmission range of another node.
The transmission range was followed by the duration for which the range was valid. The duration consisted of a start and stop time. The start time should always be 1 second more than the previous stop time except for the first one in which case it was 0. The corresponding stop time should at least be 1 second more than the current start time. The next two numbers were the direction in degrees ranging from 0° to 360° and speed of movement.

In this case there were 4 nodes running. The distance filter would use this information to calculate the position of each node for each second of the simulation. The calculation was done before the simulation was started so that the filter does not take too long to determine if the packet was valid or not during runtime. The information was then kept in an array for use during runtime. The downside of such an algorithm was that the memory usage would increase proportionately to the simulation time. It should not cause any problems if the program was run on personal computers but if the program was ported to smaller devices such as PDAs, it might run out of memory very quickly. In this case it might be better suited to have the algorithm compute the data during runtime to save memory. The algorithm for calculating the position is as follows:

Note: The bold words are inputs from the simulation file.

For each second in the duration input {

If the speed is between 0° and 89° {

Distance units moved in x direction = speed*cos((direction-0°)* \(\frac{PI}{180}\))

Distance units moved in y direction = speed*sin((direction-0°)* \(\frac{PI}{180}\))

Add the distances to the value for the previous second and store in the array

}
If the speed is between 90° and 179° {

Distance units moved in x direction = speed * cos((direction - 90°) * \( \frac{\pi}{180} \))

Distance units moved in y direction = speed * -1 * sin((direction - 90°) * \( \frac{\pi}{180} \))

Add the distances to the value for the previous second and store in the array
}

If the speed is between 180° and 269° {

Distance units moved in x direction = speed * -1 * cos((direction - 180°) * \( \frac{\pi}{180} \))

Distance units moved in y direction = speed * -1 * sin((direction - 180°) * \( \frac{\pi}{180} \))

Add the distances to the value for the previous second and store in the array
}

If the speed is between 270° and 360° {

Distance units moved in x direction = speed * -1 * cos((direction - 270°) * \( \frac{\pi}{180} \))

Distance units moved in y direction = speed * sin((direction - 270°) * \( \frac{\pi}{180} \))

Add the distances to the value for the previous second and store in the array
}

Since the actual emulation might not start at the exact time the filter was loaded by the emulator, a start time variable can be set so that the filter knows the current emulation time as compared to the real clock time. The filter would also log any errors that occurred
into a file called distanceFilter_log. This log file could be viewed later by a user to determine what if anything went wrong. An example is showed below:

4) Input string passed to charToInt was NULL.  
4) Input string passed to charToInt was NULL.  
4) Input string passed to charToInt was NULL.  
4) Input string passed to charToInt was NULL.

In most cases, unless a critical error had occurred, the program would continue running after logging the error.

3.4. Error Insertion Filter

This filter was created to insert errors in the incoming packets so that a user could emulate different kind of packet error rate per link. The filter would use a file called errorInsertion.data to determine the error rate of the virtual link. The content of the file is shown below:

Please only modify the value after the = sign; Do not add or remove lines from this document; Do not modify the name of the data

A percentage of packets will get errors in them. Packets will be chosen at random
PercentageError=50;

The filter would use the percentage from the file to figure out the number of packets that should contain errors. If the data file specified to insert errors in 50% of the packets, the filter would count each packet coming in and would insert errors in 50 out of every 100 packets. A random number generator was used to determine which packets out of the 50 would contain errors. A second number generator was used to determine which byte of the packet would be modified. Finally a third number generator was used to determine which substitution byte would be used.
3.5. Packet recognition and reconstruction design

Since the emulator was supposed to recognize any kind of data stream used by the physical device, it was necessary to add some overhead to create a packet that will be recognized on the receiving end. The overhead consisted of header and footer bytes. The header bytes consisted of the byte 0xFE followed by the Identity of the sender twice followed by 0xFE again. The footer bytes consisted of the byte 0xEF followed by the Identity of the sender twice followed by 0xEF again. This is shown in figure 5. The size of ID is a byte. This means that at this time at most 251 nodes could be used altogether. ID 0x00, 0xFF, 0xFE and 0xEF were reserved with 0xFF used for broadcast.

This sequence of characters was to ensure that the packet re-constructor on the receiver side knew the boundary of the packet. Although this added 8 bytes to the total packet length, it was necessary so that the receiver could create the packet again from the byte stream of the physical device. There was no error correction at this level. If either the header or footer was changed during transport, the packet would not be recognized. Error correction would be a nice addition if ever the emulator was upgraded in the future. This issue may cause the re-constructor to drop the next packet as well while it tried to locate the start of a new packet. This shortcoming is unavoidable due to the fact that the re-constructor has to look at each byte coming in one at a time.
The sender function is used by the application layer to send packets to different part of the network. On the receiving end, the serial buffer is continuously monitored by a thread for new data. As the new data arrived one by one, the data was fed to a re-constructor function that placed it in the appropriate location in a packet object. If a packet was reconstructed correctly, it was then sent to the filters.

To prevent any bottleneck from occurring and the possibility that the serial buffer might fill up faster than the re-constructor can create packets, intermediate circular buffers had been added with independent threads on each side of the buffer. The thread on one side would be responsible for grabbing data from the serial buffer and sending the data to the re-constructor thread on the other side. The re-constructor will create a packet and put it in another buffer. On the other side of the buffer, another thread would take care of sending the packets to the filters.

Since the receiving thread was always monitoring the serial port, the sender and receiver thread had to be synchronized appropriately. In this emulator, the sender thread had priority over the receiver thread. This meant that if the application needed to send any data, the emulator would stop the receiving thread, send out the data and then resume the receiving thread.

The re-constructor function was created as a state machine whose purpose was to insert bytes in the right position in a packet depending on which state it was in. This is shown in figure 6. Three different packet objects are kept at all times. This was because the header and footer sequence might appear inside the body of a packet. For example, if a complete header sequence was found, a new packet would be created and the data section would be filled until a corresponding footer sequence was found. If another header sequence was detected while the data section of 1 packet was being filled, a new
packet would be created with the new header. In most cases, there would not be any footer for the second packet or the ID might be out of bounds. This was done in case, some bytes were lost in communication and the new header was actually the start of another packet whereas the footer for the previous packet was lost. The first packet would therefore continue to be filled until the maximum number of bytes was reached. When this happened, the data packet content would be reset. In the rare cases where there was both a header sequence and a corresponding footer sequence inside a valid data packet, a new packet would be created and sent to the filters. This was unavoidable due to the nature of the re-constructor.

If the packet was passed to the application, it was the application’s duty to perform additional tests on the packet to know if it was valid or not. Some of the possible scenarios are shown below:

NOTE: Header and footer are in bold, data section is in italic and fake header is underlined.

Case 1: Header inside data section.

**FE 01 01 FE 02 38 48 FE 02 02 FE 46 78 92 EF 01 01 EF**

In this case, the fake header would cause another packet header to be created just in case.

Case 2: Header and footer inside data section.

**FE 01 01 FE 02 38 48 FE 02 02 FE 46 EF 02 02 EF 78 92 EF 01 01 EF**

In this case, the fake header would cause another packet to be created just in case. This packet would actually be sent to the filters.
Figure 6: State machine for re-constructor function
Case 3: Header inside data section followed by valid packet with same ID as the fake header.

**FE 01 01 FE 02 FE 02 02 FE 92 EF 01 01 EF FE 02 02 FE 48 EF 02 02 EF**

In this case, the fake header would cause two packets to be created. One packet being **FE 02 02 FE 92 EF 01 01 EF FE 02 02 FE 48 EF 02 02 EF** and another one **FE 02 02 FE 48 EF 02 02 EF**. The first packet created was a fake but the re-constructor has no way of knowing that.

Note that this would not affect the good packets which would still be sent to the application layer. The only downside was that extra packets might slip through. The chance of this happening is very remote.

The state machine used by the packet-reconstructor simply placed the byte received on the serial interface in the appropriate place in a packet. If the byte was a header, it would be inserted at the front of a packet. It would then wait for three successive bytes containing the same node ID twice followed by another header. Once these four bytes were obtained, any bytes that were not footer bytes would be added to the packet. If a footer byte was obtained, the reconstructor would then wait for three successive bytes containing the same node ID twice followed by another footer byte. The footer node ID would be checked against the header node ID to make sure they were from the same packet. Once these four bytes were obtained, the packet reconstruction would be done and the packet would be forwarded to the filters. If the footer ID did not match a header ID, it would be inserted into the packet body as regular data bytes. This process would continue until the maximum packet size was reached by the reconstructor.
After that the packet would be discarded and the re-constructor would wait on a header byte before constructing a new packet.

3.6. **Statistic collection**

Since the emulator was created to be transparent to the higher layers, it could not be set to know in advance the type of packets that the higher layers would be sending. Therefore collecting statistics about routing or transport layer packets could not be done at the emulator level. Data collection would therefore have to be done at the user level. The programmer would need to insert code to collect specific data regarding his/her protocol. The same thing would apply to the filters used in the emulator. Each filter used can have coded instructions to collect the data that it would be filtering.

3.7. **Application transfer engine**

The emulator program would need to be installed on each node individually. Since the application that ran using the emulator may differ depending on the user, a file transfer engine was developed to allow the rapid propagation of the program before starting the emulation. The transfer engine consisted of server controller programs that would always be running in the background on the slave nodes. The master node would consist of the client that would connect to the servers and transfer the information. Since it was critical that the application sent was exactly the same as the application received, the integrity of the file on the receiving end would need to be checked using 32-Bits CRC. This meant that the sending end would compute the CRC of the file and send it to the receiving end. After receiving all the data from the master, the slave would compute the CRC of the file it just received and compare it to the CRC it received from the master.
If they are not the same, the slave would inform the master that it would need to resend the data. If data got corrupted 10 times, the master would abort the operation. The file transfer engine could also be used to transfer any sort of file such as filter files, new modules for the emulator and to retrieve the statistic files from each node. In essence a user would only need to use 1 machine to control the whole emulation. The transfer engine used tcp/ip to perform the transfer operation. This provided a better quality of service ensuring that critical information was not lost in the sending process. Figure 7 below shows the interface of the application.

![Mobile Nodes Emulator interface](image)

*Figure 7: MoNoE program*

The program can perform multiple functions as outlined below:

1. **Create a simulation file.**
   
The user would go through a wizard to create the simulation file used by distanceFilter.dll

2. **Send a file to all other nodes in the network**
   
The user could choose to send a file to 1 particular node or to as many nodes in the simulation as needed

3. **Retrieve a file from all other nodes in the network.**
   
   Files would be retrieved from all nodes in the current emulation.

4. **Start a simulation**
The simulation would be started on all slave nodes in the current emulation. This was used as a time synchronizer also. All the independent programs would be synchronized to the master time.

5. **End a simulation**

The receiving program controller would first send the “ENTER” keystroke to the program twice with a time interval of 10 seconds in between. After that a kill request would be sent to the program and the controller will wait a certain amount of time for the program to terminate.

This program could be used by a user to perform a simulation from only 1 computer. This considerably reduced the time needed to perform the emulation. The user could create the simulation file needed, upload that file to all the nodes that will run the simulation. He could then upload his own program and then start the simulation. When the simulation was done, a user could choose to manually terminate the simulation if his/her program did not terminate automatically. If his program had any output written to files on the slave computer, a user could retrieve those files for later analysis. Sending the files and retrieving the files were done automatically for all nodes in a network. In effect the user only had to click the button once and the program would do the rest.
3.8. **Limitations of the emulator**

The emulator did not support the emulation of all physical effects of a real wireless environment. Even though the nodes were not in the same broadcast domain in the virtual landscape, any transmission would affect all other nodes in the real physical broadcast domain. This could cause an increase in collision and error rate. A second issue was that out of order packet delivery would never occur in the real wireless domain because all the physical nodes could communicate with each other. Since all nodes could hear each other’s transmission, the effect of the hidden terminal problem would not be felt in the real network. Although the emulation of these effects was not done, new filters could be created to process the packets and introduce these effects in the future.
Chapter 4  Skipjack-based MANET routing layer security

4.1. AODV Outline

Ad-Hoc On-Demand Distance Vector routing has been a very popular protocol that already has had many different flavors in the industry. There has been Kernel AODV provided by the National Institute of Standard and Technology, AODV-UU by the Uppsala University and UoB JAdhoc by the University of Bremen just to name a few. Some were created to be exclusively used on the Linux platform whereas others created in JAVA could be used anywhere the JAVA virtual machine is running. A lot of AODV implementation assumed that packet loss would be minimal as viewed from the routing protocol since the Link layer would take care of packet consistency and retransmission. In this case though, the Link layer used does not provide any of those services. Therefore the AODV protocol that was implemented did not assume anything about the lower layers. Another point was that the AODV protocol would need to work with the emulator created. The emulator had certain restrictions due to the size of the node id and the total size of the network. Therefore a new AODV protocol was implemented called AODV_RIT that did not assume that any underlying protocol was performing any packet consistency check. The original specification by Perkins and Royer [13] was used to create the protocol. It seemed though that the protocol specification did not take into account some more intermittent extreme conditions that can sometimes occur in some wireless networks. It was therefore modified somewhat to be better suited to the conditions that it would encounter. These modifications have been outlined below.
4.2. AODV Implementation

AODV is an on-demand protocol meaning that it will find a route only if requested by the higher level protocols. If the higher level protocols were not sending any information, then AODV would not maintain any routing information for too long. This was very important so that congestion was reduced in a wireless network where the available bandwidth is limited. If a node that used a route once was to send messages all the time checking if that route was still present or not, then other nodes that might have critical information to transmit might be delayed or in the worst case scenario not be allowed to transmit the data at all.

AODV relies heavily on control messages to perform its work correctly. These messages are:

a) RREQ – Route REQuest information

This message was sent whenever the higher level protocols sent a packet to a destination and the node does not know the next hop to the final destination. The packet sent contained the following fields for a total of 19 bytes.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td>(6)</td>
<td>(7)</td>
<td>(8)</td>
<td>(9)</td>
<td>(10)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Fields of RREQ message

(1) RREQ CODE
(2) SOURCE ID
(3) DESTINATION ID
(4) BROADCAST NUMBER
(5) SOURCE SEQUENCE NUMBER
(6) DESTINATION SEQUENCE NUMBER
(7) TIME TO LIVE
(8) HOP COUNT
(9) FROM
(10) TO

Note that the source and destination id was only 1 byte. Due to the restriction of the emulator, the maximum number of nodes that could be used is 254. ID 0x00 was reserved for a master controller and ID 0xFF was reserved for message broadcast. The sequence numbers were 4 bytes in length, this meant that $4 \times 10^9$ messages would need to be sent before the sequence numbers wrap around. Just to give an idea, it would take 49 days of sending messages every millisecond for the sequence numbers to wrap around. The FROM and TO field were special fields that were not in the original protocols. These were used to know which neighbor the packet came from. It was also used to unicast messages to the neighbor nodes. For broadcasting, the TO field was set to 0xFF. The Time-To-Live was usually set to 0xFD since this was the longest possible route if every node was aligned in a straight line with a node only being able to communicate with 1 other node.

b) **RREP – Route REPly information**
This message is sent whenever a node receives a route request message for which it has a destination in its routing table or if the node itself is the destination of the route request. The packet sent contains the following fields for a total of 14 bytes.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

(Figure 9: Fields of RREP message)

(1) RREP CODE
(2) SOURCE ID
(3) DESTINATION ID
(4) BROADCAST NUMBER
(5) DESTINATION SEQUENCE NUMBER
(6) HOP COUNT
(7) FROM
(8) TO

This packet was unicasted back to the source of the route request message by following the route the request took. This was possible since all the intermediate nodes that passed the RREQ message also created a route back to the source of the message for the RREP to take.

c) HELLO – Local connectivity messages

HELLO messages were very important in a wireless environment due to the lossy nature of the wireless connections. Hello messages were used to know which
nodes were neighbors and to determine whether a neighbor had moved out of wireless range or not. The packet sent contained the following fields for a total of 7 bytes.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

(Figure 10: Fields of HELLO message)

(1) HELLO CODE  
(2) SOURCE SEQUENCE NUMBER  
(3) FROM  
(4) TO

This packet was usually broadcasted out and was not re-broadcasted by the nodes receiving the HELLO message.

d) **DATA – The data packet**

DATA packets contain the data that the higher levels want to transfer to another node. They also contain some information about the sending node for local node management purposes.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>...</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

(Figure 11: Fields of DATA message)

(1) DATA CODE  
(2) SOURCE ID
The protocol relied heavily on routing tables to keep track of which route was valid or not. The main routing table contained the following entries:

- **Destination:**
  This was the destination node identification number.

- **Number of hops:**
  This was the number of hops it would take to reach the destination.

- **The next hop:**
  This was the neighbor node that the data would be sent to.

- **The current destination sequence number:**
  The latest sequence number for this particular destination.

- **Time when route would expire if not used:**
  The time the route would expire if not used by the higher protocol levels. Usually this element was reset every time the route was used.

- **Time when route would expire if next hop neighbor does not reply to hello messages:**
  The time the route would expire if the next hop neighbor does not reply to a certain number of hello messages. This usually indicated that the neighbor moved
and was out of reach of this node. It could also be caused by a very poor wireless link even though the neighbor was within wireless range.

There was also a back pointer routing table that contained back pointers for RREQ received by a node:

- **TO:**
  This was the node to which any RREP received that matched this back pointer would be sent to.

- **Source:**
  The source field in the RREQ message that created this back pointer.

- **Destination:**
  The destination field in the RREQ message that created this back pointer.

- **Broadcast number:**
  The broadcast number field in the RREQ message that created this back pointer.

- **Destination Sequence number:**
  The destination sequence number of the RREQ that created this back pointer.

- **Expiration time:**
  The time this back pointer would become invalid. This was used so that the nodes that were not on a route don’t keep back pointers forever. Back pointer entries were usually deleted whenever a node received a RREP or the back pointer expired.
The code for AODV_RIT relied heavily on threads to perform all the concurrent actions that were required by the protocol. These actions were:

1. Periodic check of the routing table to check if any pending request had a matching route.
2. Periodic check of the routing table to invalidate old routes.
3. Periodic check of the list of request so that AODV knows when to give up if a route had not been found after a certain number of trials.
4. Hello messages needed to be sent periodically.
5. Receiving messages from other nodes and responding appropriately to them.
6. Outputting packets sent by different sections of the protocol.

The main problem with that many threads was that synchronization became critical to prevent memory corruption.

4.3. **AODV Modifications from original protocol**

The original protocol did not work very well when used in a very lossy environment with this emulator since the wireless nodes broadcasted every message to all other nodes in the network. Some changes needed to be made as follows:

AODV_RIT did not keep track of the active neighbors using a route. In the original protocol, if a route was invalidated, only the active neighbors were notified of the change. During testing, it was found that unicasting messages to all the active neighbors did not provide the best solution. Due to the lossy environment, a lot of messages got lost and many neighbors still sent messages to the node even after an “inactive route”
notification was sent. On top of that, since the hardware nodes always broadcasted their
data, even unicasting really meant broadcasting the data to all the nodes. Since the same
message was sent multiple times to each active neighbor, the overhead was greatly
increased and the bandwidth taken by all the messages was more than was really
required. The solution was to broadcast the message once and let the receiving nodes take
care of deciding whether the message should be used or not.

The second modification was to use local connectivity management on all packets
received by the node and not only hello messages. This was due to the fact that many
messages did not actually reach their destination and it would happen that although a
node was still a neighbor, none of its hello messages reached another neighbor node.
Since other messages such as RREQ and RREP were still heard by neighbors, these were
also used to perform local connectivity management.

4.4. **AODV PseudoCode**

This chapter will go over the code and explain what is happening in all the
different functions related to the AODV protocol.

**sendData:** This function was responsible for sending the data obtained from the higher
layers to the appropriate node.

\[
\text{IF the ID is 0xFF \{ \}
  \text{AODV will send out the packet immediately. The higher layer is notified that the packet}
  \text{has been sent.}
\]

\[
\text{\}}
\text{IF NOT{}
\]

41
The routing table is searched for the ID of the destination node. Neighbor nodes destinations are searched first then nodes that are further away.

IF route is found {
    Packet is sent immediately and the route entry is updated to show that it was used recently. The higher layer is notified that the packet has been sent.
}

ELSE IF route is not found /
    IF route is located in the old routes table {
        Send a RREQ message with sequence number 1 greater than in old route entry and increase the broadcast ID by 1
    }
    ELSE {
        Send a RREQ message with sequence number 0 and increase the broadcast ID by 1
    }

recvPacket_aodv:- This function was responsible for receiving all the control and data packets and putting them in the right buffer.

IF packet is a RREQ packet {
    IF size is correct AND IF TTL is not equal to 0 {
        IF packet source is not equal to this node’s ID {
            Create a RREQ object and put it in the RREQ incoming buffer.
        }
    }
    ELSE IF packet source is equal to this node’s ID {
        Since the packet was sent by a neighbor, update the neighbor table.
    }
    ELSE {
        Report BAD packet.
    }
}

ELSE IF packet is a RREP packet {
    IF size is correct {
        Since the packet was sent by a neighbor node, update the routing table
        IF the packet is meant for this node {

42
IF the number of hops is infinite {

   IF sequence number of packet is higher than sequence number of destination in routing table {
      The destination in the routing table is invalidated.
   } 
}

ELSE IF the number of hops is not infinite {
   Create a RREP object and put it in the RREP incoming buffer.
}

ELSE IF size if not correct{
   Report BAD packet.
}

ELSE IF packet is a HELLO packet {

   IF size is correct {
      Create a HELLO object and put it in the HELLO incoming buffer. 
      Since the packet was sent by a neighbor, update the neighbor table. 
   }

   ELSE {
      Report BAD packet.
   }

ELSE IF packet is a DATA packet {

   IF size is correct {

      IF the packet is meant (TO field) for this node or packet was broadcasted {

         Since the packet was sent by a neighbor, update the neighbor table.

         Delete any backpointer that was waiting for an RREP that matches the destination, source and FROM fields of the DATA packet.

         Delete any previous RREQ received that matches the destination, source and FROM fields of the DATA packet. The RREQ was kept to prevent the node from sending multiple RREPs if receiving the same RREQ from other nodes.

   }

   ELSE {

      
   }

}
IF packet’s destination (DESTINATION field) is this node or it was broadcasted {

Update the routing table entry for the source of the packet only if it is already present and the sequence number in the table is less or equal than the sequence number in the data packet

Send the data to the higher level.
}

ELSE {

IF the routing table has the next hop for the destination.{

Forward the packet with the FROM and TO modified to this node and next hop respectively
}

ELSE IF destination was an old routing entry {

Send an RREP packet with the number of hops infinite and sequence number 1 greater than the content of the old routing entry.
}

ELSE {

Send an RREP packet with the number of hops and sequence number infinite.
}
}
}
}

ELSE {

Report BAD packet.
}

**copyRREQ:** This thread was responsible for checking the RREQ buffer and either sending an RREQ out to other nodes if the RREQ was new and the destination was not in the routing table or sending back an RREP if the destination was in the routing table.

**WHILE RREQ buffer is empty {**
Sleep for 5 milliseconds
}

For each element in RREQ buffer {

Since the packet was sent by a neighbor, update the neighbor table.

If destination contained in RREQ is found in routing table or this node is the destination {

Send RREP back to the neighbor which sent the RREQ
}

ELSE if destination cannot be located and this RREQ was not obtained previously {

Create or update the routing entry for the source of the RREQ

Create a back pointer pointing to the neighbor which sent the RREQ.

Broadcast RREQ after decrementing the TTL.
}
}

expireRREQ: This thread is responsible for removing old backpointers after a certain time has elapsed.

processRREP: This thread was responsible for processing the RREP obtained and creating or updating routing table entries.

WHILE RREQ buffer is empty {

Sleep for 5 milliseconds
}

For each element in RREQ buffer {

Since the packet was sent by a neighbor, update the neighbor table.

IF RREP matches a RREQ requested by this node {

Create a routing entry for the destination in the routing table.
ELSE IF RREP matches a back pointer {

Update the routing entry for the destination in the routing table.

Send RREP back to the node that the back pointer points to.
}

ELSE IF RREP is unsolicited {

Update the routing entry for bad routes.
}

expireROUTING:- This thread removed the routing entry from the routing table after certain conditions have been met.

IF routing entry is for a neighbor node {

IF a certain number of hello messages have not been heard {

Remove the neighbor routing entry

Remove any destination that depends on that neighbor for their next hop.
}

ELSE IF routing entry is for a destination 2 or more hop away {

IF the routing entry has not been used for a certain amount of time {

Remove the routing entry.
}

expireROUTESEQ:- This thread removed the old routing entries after a certain amount of time. Old routing entries were used to determine if the routing table previously contained a route or not. The sequence number of the destination would be used with the RREP with an infinite number of hops.
checkRREQRequest: This thread is used to check the RREQ requested by this node. It will check the same RREQRequest for a certain amount of time.

IF a route has been obtained {
    Send the data corresponding to the RREQRequest.
    Notify the higher layers that the data was sent.
}

ELSE If route has not been found {
    IF a certain amount of RREQ have been sent {
        Notify the higher levels that a route to the destination could not be located.
    }

    Increment number of times RREQ was sent.
    Send the RREQ again with a new broadcast ID
}

outputPackets:- This thread was responsible for sending packets to the lower layers under the routing layer.

sendHello:- This thread is used to send a hello message every x seconds with some randomization to prevent all nodes from broadcasting hello messages at the same time.

processHello:- This thread was responsible for updating the neighbor entries in the routing table when receiving hello messages.
4.5. **Skipjack outline**

In today’s world, the threat of hackers and malicious users has been a constant reminder that security is of utmost importance in any wireless network. Any data sent over a wireless network had to be at the very least encrypted to prevent tampering. Since wireless devices could not be power hungry, simple but reliable encryption algorithms needed to be used to make it an efficient and viable solution. The encryption needed to be at the lowest level of the OSI layers possible to allow only the final bytes that would actually be sent on the wireless medium to be secured. Multiple encryptions at different layers might be more secure but it would also consume much more power in terms of processing the data multiple times. It was decided to use the Skipjack Encryption algorithm. The algorithm was very small and as of yet the full 32 rounds have not been fully broken.

4.6. **Skipjack implementation**

Skipjack was a symmetric encryption/decryption algorithm that used two stepping rules to secure 8 bytes data blocks. The stepping rule was defined as follows

**Encryption**

Rule A:

\[
\begin{align*}
  w_1^{k+1} &= G^{k}(w_1^k) \oplus w_4^k \oplus counter^k \\
  w_2^{k+1} &= G^{k}(w_1^k) \\
  w_3^{k+1} &= w_2^k \\
  w_4^{k+1} &= w_3^k
\end{align*}
\]

*Figure 12: Rule A [15]*
Rule B:

\[
\begin{align*}
    w_{1}^{k+1} &= w_{4}^{k} \\
    w_{2}^{k+1} &= G^{k}(w_{1}^{k}) \\
    w_{3}^{k+1} &= w_{1}^{k} \oplus w_{2}^{k} \oplus \text{counter}^{k} \\
    w_{4}^{k+1} &= w_{3}^{k}
\end{align*}
\]

Figure 13: Rule B [15]

The total number of steps was 32. First step A was done 8 times, then step B was done 8 times, this was followed by step A 8 times and step B 8 times again.

Decryption

Rule A:

\[
\begin{align*}
    w_{1}^{k-1} &= [G^{k-1}]^{-1}(w_{2}^{k}) \\
    w_{2}^{k-1} &= w_{3}^{k} \\
    w_{3}^{k-1} &= w_{4}^{k} \\
    w_{4}^{k-1} &= w_{1}^{k} \oplus w_{2}^{k} \oplus \text{counter}^{k-1}
\end{align*}
\]

Figure 14: Rule A\(^{-1}\) [15]

Rule B:

\[
\begin{align*}
    w_{1}^{k-1} &= [G^{k-1}]^{-1}(w_{2}^{k}) \\
    w_{2}^{k-1} &= [G^{k-1}]^{-1}(w_{2}^{k}) \oplus w_{3}^{k} \oplus \text{counter}^{k-1} \\
    w_{3}^{k-1} &= w_{4}^{k} \\
    w_{4}^{k-1} &= w_{1}^{k}
\end{align*}
\]

Figure 15: Rule B\(^{-1}\) [15]
For the decryption, the algorithm was the same as for encryption but in reverse. The total number of steps was 32. First step B was done 8 times, then step A was done 8 times, this was followed by step B 8 times and step A 8 times. The permutation used in the encryption and decryption was a four round feistel structure. Each round consisted of a byte substitution using a substitution table. Each round also included using a specific private cryptovariable. The cryptovariable was 128 Bytes long and each byte was used in 1 round of the byte substitution. The permutations are shown below

\[
G^k
\]

\[
[G^k]^{-1}
\]

Figure 16: G-permutation diagram [15]

The algorithm for skipjack encryption/decryption was then incorporated into the AODV protocol. Since this algorithm required that the data to be encrypted is of length modulo 8, for any size data packet, at most 8 bytes needed to be added to the packet for encryption. Since the encryption algorithm was to be used in Cipher Block Chaining (CBC) mode, the Initialization Vector (IV) with length 8 bytes used in the encryption
would need to be added to the packet so that the receiving node can decrypt the data. The CBC mode is shown below. The IV is a block of bytes that is used to start up the cipher block chaining.

![Diagram of Cipher Block Chaining mode]

*Figure 17: Cipher Block Chaining mode*

No key exchange algorithm was incorporated into the protocol. The main reason was a lack of time. Therefore all nodes would need to possess the same key to be able to encrypt and decrypt the data. In real life this type of private key could be some code that a user needed to enter manually into a device before the device would allow the user to enter a certain network.
Chapter 5  Transport layer design

5.1. **Pump Slowly Fetch Quickly (PSFQ) outline**

Wireless devices are everywhere we can think of. They could be inside home appliances relaying data regarding the state of the device, used as sensing applications in security devices or to monitor hazard areas and even used inside the human body to monitor the health of a patient. One of the problems has been that the code used in the machines might need an upgrade from time to time. It could be due to an error that needed to be fixed or maybe the function of the device needed to be modified. This might pose a problem if the wireless device was found in a hazardous or unreachable environment. It was therefore important to have a system that would allow the devices to be reprogrammed remotely. Reprogramming wireless devices involved many different tasks [23] that cannot all be discussed here. This section will therefore concentrate on the aspect of code dissemination at the transport layer.

Existing transport protocols fall into two categories; Congestion Control Protocol and Reliability Guarantee protocol [19]. PSFQ falls into the second category. It is a transport protocol used for the propagation of downstream data. It has been advertised as being able to deliver data from a master node to slave nodes across multi-hops with close to a 100% reliability factor. PSFQ did not take a traditional end-to-end approach to error recovery. Since reliability decreased quickly with increasing number of hops [12] as shown in figure 18, PSFQ was designed to be a hop by hop transport protocol. This meant that the data would need to be consistent across 1 hop before PSFQ would allow it to be sent to the next hop.
Figure 18: Probability of successful delivery of a message using an end-to-end model across a multi-hop network [12].

5.2. **PSFQ implementation**

PSFQ was designed to send data packets at regular interval to neighbor nodes (PUMP slowly). This allowed neighbor nodes to request any missed packets in between the data packets (FETCH quickly). The request was done using Negative Acknowledgement (NACK). This meant that a node would only request for a packet resend after getting a new packet and if there was a sequence gap between packets. The protocol relied heavily on timers to keep track of when to send the regular packets and how fast to request for packet resend.
Since PSFQ usually broadcasted its data to all neighboring nodes, one important drawback that needed to be considered was the problem of message implosion. To prevent message implosion, the protocol would not send a certain packet if it heard the packet being broadcasted by other nodes a certain number of times. This was also true for NACKs. Each regular packet was sent at time intervals between T_MAX and T_MIN whereas the fetch operation was done a certain number of times before T_MIN. This was to ensure that the missed packets would be obtained before the next regular packet was sent as well as to prevent message implosion by allowing nodes to hear other broadcasted packets. T_MAX and T_MIN can be adjusted depending on the state of the wireless medium.

Another important feature of PSFQ was that only packets with continuous sequence numbers were forwarded to the next node. This was to prevent a missed packet error to propagate to other nodes which could cause unnecessary NACKs transmissions. This is shown in figure 19.

*Figure 19: Node A, B and C would all send NACKs. B would send to A and C would send to B. But B and A would not have the packets so the NACKs would be wasted.*
A third important function in PSFQ was the report operation. The master node needed a way to know that the end nodes had received the packet properly. PSFQ therefore used a report CODE to request that the slave nodes tell the master node if they received all packets or not. To prevent message implosion, the furthest node from the master node was the only one to reply and all intermediate nodes piggybacked their answer on that reply.

5.3. **PSFQ modification**

The paper describing PSFQ protocol stated that any packet that had the most significant bit of the TTL high would act as a report request. Instead of using 1 bit in the TTL field, a separate REPORT packet was sent to request a report from the node. This was done since the maximum number of nodes allowed in the emulator was 254. Therefore the TTL field was set to use 8 bits and 1 bit could not be used as a report bit.

This PSFQ implementation did not really guarantee 100% packet delivery since the protocol description did not give any details what happened when the master node received a report that said that a node had not received all packets. In this case, it was decided to only measure the number of data that actually reached all the nodes. Further implementation might use the report as a mechanism to start sending the missing packets again.

Although PSFQ could be used to send any kind of data, this particular implementation of PSFQ was used to send a large file to other nodes for demonstration purposes.
5.4. PSFQ pseudocode

This chapter will go over the code and explain what is happening in all the
different functions related to the PSFQ protocol.

sendMessage: This function was used by the master node to send a file to slave nodes.

Split the data passed to this function into segments of equal lengths except the last
segment which can be less or equal than the others.

Schedule the packets to be sent at intervals between T_MIN and T_MAX with some randomization.

Wait until all packets have reached the last node.

Start the report thread.

recvPacket_psfq: This function was responsible for receiving all the control and data
packets and putting them in the right buffer.

IF packet is a DATA packet and size and CRC are correct {

IF DATA packet has not already been received {

IF this is the first DATA packet {

Create an array of size equal to the number of bytes in the file.

Store the first packet.

Start the resend thread.

Set the time for proactive fetch to start.
}

ELSE {

IF DATA packet is new {

Store the packet.

Start the resend thread.
}
Update the time for proactive fetch to start.

ELSE IF DATA packet has been received before {

Start the NACK thread.
}

IF data sequence number does not follow the last in sequence packet {

Start the NACK thread.
}

IF packet is a NACK packet and size and CRC are correct {

Extract the windows and store in a NACK object for the processNack thread to work on.
}

ELSE IF packet is a REPLY and size and CRC are correct {

Store the reply in the appropriate location in the buffer.

Update the time for proactive fetch to start.

Start the NACK thread.
}

ELSE IF packet is a REPORT and size and CRC are correct {

IF REPORT request was not sent by this node and this report has not been heard before {

IF TTL is equal to 1 {

Send a reply with the latest in sequence packet received.
}

ELSE {

Store packet

Broadcast the packet with TTL minus 1.

Set a timer equal to T_MAX * (TTL - 1) to wait for a reply.
ELSE {
    Report BAD packet.
}

**dataOutput**: This thread was responsible for sending DATA packets to the lower layers under the transport layer.

**checkGaps**: This thread would check for out of sequence packets in the input buffer.

IF a gap is located {
    Remove any outstanding NACKS waiting to be outputted.
    Create a new NACK and store in output buffer.
}

ELSE IF there is no gap{
    empty the NACK output buffer.
}

**nackOutput**: This thread was responsible for sending NACK packets to the lower layers under the transport layer.

**processNack**: This thread was responsible for checking if the windows found in NACKs received corresponded to any packets that the node had in its input buffer. If it did the packets would be scheduled for output.
Resend: This thread only forwarded packets if they had continuous sequence numbers.
For example if the input buffer contained segment 1,2,3,5,6,7; only packets 1,2 and 3
would be forwarded. Once packet 4 was received, the rest of the packets would then be
forwarded.

proactiveFetch: This thread would send out a NACK if the last segment was not
received in a certain time frame.

askReport: This thread sends out a report request a certain number of times. It can
usually only used by a master node.

checkReportReply: This thread checks the replies obtained. If it matched a pending
report request, the reply would be forwarded to the node that sent the report request.

IF report reply matches report request {
  IF size of report reply is greater or equal than the maximum size for report replies {
    Create a new reply packet and send out.
    Send out the old packet.
  }
  ELSE {
    Add this node’s information to the packet and send out.
  }
}
waitForReportReply: - This thread check the wait time for reports in the report buffer. If their wait time expires, this thread will send a reply message to the node that sent the report request.
Chapter 6  Analysis

6.1.  AODV_RIT

The number of available nodes to perform the experiment was limited to 4 nodes. Therefore very limited testing was possible. These tests might be only valid with the particular hardware device used. If another wireless device would be used, different results might be obtained. The tests were carried out at different times of the day. The tests were done as follows:

1. The first experiment used 4 nodes to send 1000 packets of 50 bytes from node 1 to node 4 with varying packet error rate per link. This was done to obtain results regarding end-to-end error rate and control overhead percentage of the AODV.

![Control Overhead versus Packet Error rate per link](image)

Figure 21: Overhead percentage versus packet error rate per link.
The percentage of overhead packets versus data packets increased with the Packet error rate per link. Since the overhead took into consideration RREQ, RREP and HELLO messages, it was expected that as the number of nodes increased, the number of HELLO messages would increase proportionately. The RREQ and RREP mostly contributed to the additional overhead packets at high error rates where a lot of control packets were lost and routes became invalidated frequently. This caused the sending node to request a route over and over again.

![End-to-End Packet error rate VS Packet error rate per link](image)

*Figure 22: End-To-End Packet error rate versus packet error rate per link.*

The end to end packet lost increased with an increase in Packet error rate per link although at very low and very high error rates, the number of lost packets seemed to congregate to the same amount. This was probably due to most of the
packets being lost on the first hop with only a few packets making it to the next hop.

2. 4 nodes were then used with 3 nodes in a linear fashion. Node 1 was within range of node 2 only and node 3 was within range of node 2 only. Node 4 was initially within the range of node 3 only. Node 1 was the sender and node 4 was the receiver. Once a route had been set up, node 4 then started moving slowly towards node 1 stopping within range of the middle node. This was done to force node 1 to reconstruct the route. The pattern of the nodes is shown in figure 23.

![Diagram of node movement](image)

*Figure 23: Destination node 4 moving closer to sender node 1.*

The simulation file for such a movement was as follows. In this case node 4 would move back to its original position at the end:

```
NodeId, x start position, y start position, range, start time, end time, direction, speed = range, start time, end time, direction, speed,...
Number of nodes = 4
Simulation time = 7200
1, 0, 0 = 3, 0, 7200, 0, 0
2, 3, 0 = 3, 0, 7200, 0, 0
```
In figure 24, the number of rounds indicates how many times node 4 has moved towards node 1 and back to its original position. For example; for 4 rounds, node 4 moved towards node 1 then back to its original position 4 times. This particular experiment did not take into account the speed of movement but the number of times the node was moving. This experiment was run for a fixed amount of time. The graph represents the amount of end-to-end error induced with increasing topology change. From the data collected it was observed that stopping for a longer time within range of the intermediate node had no effect on the number of packets received. The movement of the node was the major cause
of packet lost in this case. This was probably because the nodes have some delay before sending hello messages and there was the possibility that some hello messages were lost. While the nodes were trying to reconfigure the paths, packets would definitely be lost in the meantime. Decreasing the time between hello messages would certainly improve data loss due to movement. The downside would be that it would increase the interference and packet collisions. Another side effect would be that since only 1 function can access the serial port at any one time, sending a lot of hello messages could actually increase the time it took to send other packets out.

Altogether these 3 experiments proved that the emulator was working correctly and reconstructing the packets with very little packet loss due to software malfunction. The trend line of the results in figure 22 seemed to be reasonable for an increasing number of hops with the same packet error rate per link. On top of that, experiment 3 showed that if a node is moved virtually, the protocol will actually observe a greater error rate in the link between each node which is the effect that movement would cause in real environments

6.2. SKIPJACK_RIT

Emulation of AODV_RIT using the skipjack encryption did not produce any significant change in the error rate as compared to sending the packet without any encryption. It only introduced a slight delay for the packet to travel from the routing layer to the transport layer. This is shown in table 1.
This was most probably due to the fact that Skipjack only added at most 15 bytes to the outgoing data. This would not have very much impact on the error induced due to packet size or the time it took to process each packet.

<table>
<thead>
<tr>
<th></th>
<th>AODV</th>
<th>AODV with SKIPJACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average increase in Packet overhead (%)</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Average increase in end-to-end error rate (%)</td>
<td>0</td>
<td>0.13</td>
</tr>
<tr>
<td>Average Packet processing time (µs)</td>
<td>57</td>
<td>81</td>
</tr>
</tbody>
</table>

*Table 1: Effect of Skipjack on AODV packets*

An example of an encryption and decryption procedure is shown in figure 25. The first window was the sender requesting a route to destination 4. Since the route was not found, an RREQ was sent. As shown in the figure, both nodes used the same encryption key since no Key Exchange Algorithm was implemented. The original RREQ packet was 19 bytes long. Therefore 5 more bytes with value 0x00 were added as padding. On top of that 8 more 0x00 bytes were added to store the IV used to create the encrypted packet using CBC. The RREQ was then forwarded by the neighbor node, received by the first node and decrypted as shown on line 6 of the first window.
Figure 25: Skipjack encryption and decryption
6.3. **PSFQ_RIT**

The number of available nodes to perform the experiment was limited to 4 nodes. Therefore very limited testing was possible. These tests might be only valid with the particular hardware device used. If another wireless device was used, different results might be obtained. The tests were carried out at different times of the day. Each experiment was stopped after the master sent the last packet to the 4\(^{th}\) node, requested a report and obtained a report reply message. The amount of packets received was then tabulated. The tests were done as follows:

1. 4 static nodes were used and the higher levels would request data of a certain size be sent with constant packet size of 50 bytes to node number 4. Three tests were done with increasing T\_MIN, T\_MAX and constant T\_NACK values. The number of end-to-end packets lost with respect to number of hops and packet error rate per link was measured. The results are shown in figure 26 to 28.
**Figure 26:** End-To-End Packet error rate VS packet error rate per link for node 1

**Figure 27:** End-To-End Packet error rate VS packet error rate per link for node 2
From the experimental results, it would seem that PSFQ’s performance is much better than that of sending packets without any hop-to-hop recovery up to a certain error rate of about 65%. After that there is a significant lost of data. Since the Timing used in PSFQ was critical to the proper use of the protocol, different intervals were used to send packets out. As T_MIN was increased and T_NACK stayed the same, more NACKs were sent before the next valid packet thus allowing the next node more chances to recover lost packets. The number of NACKs was approximately equal to T_MIN/T_NACK. The downside was that with more overhead data, the chance of packet collision and interference increased. It can be observed from figure 28 that the most improvement was seen...
for the node 3 hops away from the sender. The end-to-end packet error rate actually decreased from 15% to about 1% at a packet error rate per link of 60%. This showed that this protocol was highly dependent on the timing of its packet departure time and recovery time. It may be beneficial to modify these times for different kind of wireless medium to obtain an optimal solution.

During experiment 1, it was observed that NACKs were not being received once the NACK packets became too large. This is shown in figure 26, 27 and 28 above with high packet error rates per link above 65%.

2. 4 static nodes were used. First data with increasing packet size were sent to a node 1 hop away, then 2 hops away and finally 3 hops away. The packet error rate per link was set to 0%.
It was observed that there was a sharp increase in the number of packets lost after the size of the packets exceeded 100 bytes. I believe that this might be due to the hardware trying to cope with an increasing number of incoming bytes while its internal buffer remained fixed at 100 bytes. Some bytes might have been dropped thus corrupting packets. If the byte dropped was an emulator header or footer, then the whole packet would have been dropped by the emulator. This was
an unavoidable issue of the emulator and might have contributed to the increase in packet loss after the packet size went over 100 bytes.

From experiment 1 and 2, it was found that the NACK messages were becoming very large at high packet error rates per link. This was causing the hardware to drop a lot of incoming bytes since its internal buffer could not accommodate the influx of data. To remediate this issue, the protocol was modified so that NACKs would not contain all the gaps in the current file. Instead, multiple different NACKs would be sent with different segment gaps. The size of the segment was dependent on the size of the hardware buffer. In this case, the size was constrained to about 50 bytes because this was the optimal packet size as found in experiment 1 for AODV_RIT. This was to ensure that the NACK packets do not become too big. Packets with too many bytes might overload the hardware buffer of the receiving module and cause packets to be dropped unnecessarily.

3. After the packet size constraint was added to the program, another set of experiment was run with the PSFQ properties as follows:

    T_MIN=1
    T_MAX=2
    T_NACK=1
Figure 30: Packet error rate VS medium error rate with $T_{\text{MIN}}=1$ sec, $T_{\text{MAX}}=2$ sec, $T_{\text{NACK}}=1$ sec

This experiment should have provided results that are worst than those on figure 26, 27 and 28 for $T_{\text{MIN}}=2$ sec and $T_{\text{MAX}}=3$ seconds. Instead it showed an improvement from 25% end-to-end error rate to close to 1% for a packet error rate per link of 65%. This showed that even simple modifications could have huge impact on the overall performance of the protocol when the limitation of the hardware was taken into account.

These types of issues were only noticeable when real hardware was used. A programmer or tester could have overlooked such an issue if only using a simulation. The most probable case would have been that the programmer would think that there was a bug in the software that was overlooked or that the interference was just too great or even that the hardware might be malfunctioning. When using the emulator, a programmer would immediately notice that the issue was not the program since he/she could test the program as they were writing it.
using the hardware. In this case using an emulator on top of simulation would be more cost effective in the long run.

4. This experiment was done to compare the use of multiple intermediate nodes to help recovery. 4 static nodes were used with 2 nodes in between node 1 and 4. The 2 middle nodes would be set not to communicating with each other. The timing of the protocol was as follows:

- $T_{\text{MIN}} = 1$
- $T_{\text{MAX}} = 2$
- $T_{\text{NACK}} = 1$

![4 nodes in diamond formation](image)

*Figure 31: 4 nodes in diamond formation.*
End-To-End Packet error rate (%) VS Packet Error rate per link (%)

T_MIN=1 sec, T_MAX=2 sec, T_NACK=1 sec

Figure 32: Results for 4 nodes in diamond formation.

5. Experiment 3 was then repeated with the difference that the node 2 and 3 were set to communicate with each other. The timing of the protocol was as follows:

T_MIN=4
T_MAX=5
T_NACK=1

Figure 33: 4 nodes in diamond formation with interconnection.
Figure 34: Results for 4 nodes in diamond formation with interconnection.

Both graphs from experiment 3 and 4 started at 70% since any packet error rate per link less than that showed a 0% packet loss. Experiment 3 and 4 showed that additional intermediate nodes did in fact help in packet recovery albeit very little. The interconnection between node 2 and 3 complemented each other's packet loss causing both nodes to retrieve missing packets better. There were a couple of outliers in both graphs that was due to the hardware used and random nature of the wireless medium. The results from all the experiments were then compared side by side. This is shown in figure 35 and 36.
End-To-End Packet error rate (%) VS Packet Error rate per link (%)
T_MIN=1 sec, T_MAX=2 sec, T_NACK=1 sec

Figure 35: Comparison of packet error rate with different formations.

End-To-End Packet error rate (%) VS Packet Error rate per link (%)
T_MIN=1 sec, T_MAX=2 sec, T_NACK=1 sec

Figure 36: Comparison of packet error rate with different formations.

It was found that PSFQ worked well with small data under good wireless conditions. Once the packet error rate went beyond 80 percent per link, PSFQ had a tendency to perform poorly.
One critical observation was that whenever a packet was never recovered by a middle node, that node would never send any additional packets obtained to the next node. The next node would have to wait until the proactive fetch kicked in before starting to send NACKs and obtaining the data. This did not cause nay issues for small files but once the files became too large, the proactive fetch mechanism would take a long time before starting up. In the meantime, the nodes would be left to wait and do nothing.

It would also seem that as the number of nodes inter-communicating increased, the end-to-end packet error rate increased slightly for other nodes with the same number of hops from the sender. This might have been caused by an increase in overhead packets which prevented valid data packets from being sent out or an increase in interference due to other surrounding nodes or even buffer overflow occurring more often causing good packets to be dropped. I do believe that this might have been the major cause of the diamond formation behaving worst than the linear formation.

6.4. Integrated Communication Protocol

This experiment was done to compare the use of PSFQ on top of AODV to ascertain if there is any advantage or disadvantage. The nodes were set up to observe a common effect due to mobility whereby in the middle of a transmission, a link would break and AODV needed to find another route to the destination. Some modifications had to be done to AODV for this experiment. This was because the nature of a routing protocol was to hide the intermediate nodes from the transport layer between sender and
receiver nodes. Therefore packets sent to the intermediate nodes were never sent to the transport layer but sent directly to the next node until it reached its destination. If that was the case, PSFQ would not be able to buffer the data in the intermediate nodes and the hop to hop recovery scheme would not work. To remediate this, the routing layer had to be integrated into the transport layer as one big communication layer. All packets obtained by the routing layer had to be passed to the transport layer. The timing of the protocol was as follows:

T_MIN=4
T_MAX=5
T_NACK=1
RREQ_TRIAL = 12
RREQ_REQUEST every 1 sec
HELLO_INTERVAL = 1 sec

![Graph showing packet error rate per link](image)

*Figure 37: PSFQ over AODV.*
AODV provided the PSFQ with the ability to target nodes for reprogramming instead of being a general broadcast transport protocol. Surprisingly, although AODV added additional overhead packets that blocked the wireless medium causing the wireless devices to wait more, the difference in error rate was only minimal. Upon further analysis, it was found that the new intermediate node used after a link was broken recovered lost packets twice as fast as the other nodes. This was because the NACKs sent by this node for initial packets that it did not have were heard by both the sender and receiver of these packets. Therefore both were sending data and complementing each others packet lost. This caused the middle node to recover packets at twice the speed of regular node until the packet sequence number was back to where it was suppose to be. The only reason for the difference in error rate was actually because the emulation was stopped after a constant time interval. If it was allowed to run for some more time, the error rate would actually be more or less similar in the two cases. This showed that PSFQ was a very reliable transport protocol and could be used in very lossy environment with a lot of link breakage. As long as AODV was able to recover quickly, I believe that 100% packet delivery rate would be achieved in most cases.
6.5. Discussion

As observed from the results, the integration of PSFQ, AODV and SKIPJACK provided a very good transport protocol that allowed the transfer of packets to specific nodes in a network even without the help of the link layer performing any kind of packet checking and correction. Integrating the security algorithm into the routing layer provided the advantage that this protocol could be used with any MAC and Physical layer without having to worry if the link is secured or not. Finally the results showed that the simple integration of already created protocol with minimal modification could provide a developer with considerable reduction in the time taken to introduce new functionality into a network.
Chapter 7  Conclusion and future work

7.1. Conclusion

The objective of this thesis was achieved. An emulator was created that was easy to use and modular enough so that it was fully expandable by allowing users to swap different part of the emulator without the need to re-compile the original software. The AODV routing protocol was tested on the emulator and the performance showed that the emulator could be used as a testing environment. The PSFQ transport protocol was then tested on the emulator and the results analyzed. After experimentation and analysis of the results, a possible improvement to the PSFQ protocol was inferred and implemented which made the protocol more reliable particularly when used with the XBee-PRO wireless modem. This showed that it could be more advantageous for an implementer or tester to use an emulator to test a protocol on real hardware than just using a simulator. The PSFQ protocol was then combined with AODV and the results showed that there was no real change in end-to-end error rate. A routing protocol might be used with PSFQ to allow the transport protocol to target specific nodes but the routing protocol would need to be fully integrated with the PSFQ protocol to allow data to be transferred to the transport layer for buffering in intermediate nodes.
7.2. Future work

There is a lot that could be done to improve the overall usage of the emulator. New filter modules could be built that would emulate the wireless environment more accurately. For example a speed filter could be added that would increase the packet error rate due to the movement of the nodes.

One of the main flaws of this emulator was that it was not able to emulate the hidden terminal issue that is prevalent in wireless networks. This was due to the fact that the hardware used broadcasted all its messages and all the other nodes in the emulation could hear every message sent. Possibly if the channel frequency or antennae power used by each node could be controlled during runtime, the hidden terminal issue might be emulated.

Additional tests could be carried out using a simulator to test the written PSFQ and AODV code and compare that to the emulator. This would help to ascertain that the emulator worked correctly and the results were at least close to what a simulator might predict. Other protocols could be created to be tested on the emulator.

Since the emulator was created to function with multiple devices, different wireless modules could be used and tested with the emulator. A new communication module might be required for usb or other kind of communications. Test could be carried out to check if protocols created in other languages such as JAVA, C or C# could be used with the emulator DLL.

PSFQ itself was only a downstream packet transfer protocol. This meant that it specialized on delivering packets from a master node to other nodes. Additional research
could be done on communication protocol for upstream data delivery. This protocol could be added to the PSFQ_RIT protocol to create a self contained two way secure communication protocol. The skipjack part of the protocol did not use a Key Exchange Algorithm for additional security. This part could be implemented and added to the protocol.

Finally the emulator code could be ported to be used on other platforms such as Unix or Windows CE. The code could then be run on portable devices such as PDAs for added portability.
Bibliography


http://csrc.nist.gov/CryptoToolkit/skipjack/skipjack-kea.htm,

[16] Intel, “Understanding WIFI and WIMAX as Metro-Access solutions”,
http://www.intel.com/netcomms/technologies/wimax/304471.pdf,


