picoNet II
a wireless ad hoc network for mobile handheld devices

by

Alex Song

Department of Information Technology and Electrical Engineering,

University of Queensland.

Submitted for the degree of

Bachelor of Engineering (Honours)

in the division of Electrical Engineering

October 2001

To Mum, Dad and Brian ...

15 Aurora Crescent
Kenmore
QLD 4069
Tel. (07) 3378 9764

October 19, 2001

The Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, QLD 4072

Dear Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Electrical Engineering, I present the following thesis entitled "picoNet II - A Wireless Ad Hoc Network for Mobile Handheld Devices". This work was performed under the supervision of Dr Mark Schulz.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Alex Song.

Acknowledgements

The picoNet II project was the result of many hours of hard work. It would not have been completed without the help of many people. The author sincerely wishes to thank:

Dr Mark Schulz for his guidance, insight and his unique way of telling me I am behind schedule.

Robert and Laura Song for being the most supportive parents one could ever ask for.

David McCullough for his advice with the Linux kernel routing code.

Yih-Chun Hu for his help with the DSR protocol acknowledgement timeouts.

Andrew Over for his advice on kernel debugging techniques.

Dr Aleks Rakic for his advice on how to use aluminium foil to reduce the RF signal strength.

Dr Vaughan Clarkson for lending me his iPAQ for this project.

and last but not the least, all my friends who proofread this thesis.

Abstract

This thesis describes the design and implementation of picoNet II - a wireless ad hoc network for mobile handheld devices. As users are increasingly mobile it is more and more common for users to meet and communicate without prior planning, and in environments where there is little or no networking infrastructure. Such a network is known as an ad hoc network, where the network is of a dynamic nature without centralised administration.

Current technologies, such as Bluetooth and IEEE 802.11b, can form ad hoc networks but is limited in that only single hop networks can be formed. This means that each node can only act as a host, whereas in a multi-hop ad hoc network all nodes act as routers. This thesis concentrated on the problem of adding multi-hop capabilities to existing ad hoc network platforms such as IEEE 802.11b. This capability will allow a network to be fully dynamic, self-organising and self-configuring.

The result was an implementation of the dynamic source routing protocol (DSR) for TCP/IP on Linux, for both the PC and the Compaq iPAQ. The implementation enabled the iPAQs and the PC to form a multi-hop ad hoc network with 802.11b wireless cards. A DSR to IP gateway was also implemented and it allowed nodes in the DSR network to access external IP networks. Existing unmodified TCP/IP applications were able to run seamlessly on the picoNet II network, providing a useful platform for future extensions.

Contents

Acknowledgements ii

Abstract iii

List of Figures ix

List of Tables x

1. Introduction 1

1.1. The picoNet II Vision 1

1.2. Problems with Current Mobile Networks 1

1.3. Proposed Solution 2

1.4. Thesis Structure 2

2. Background 4

2.1. Mobile Networks 4

2.1.1. Network Reference Models 4

2.1.2. The Routing Concept 7

2.1.3. Mobile Ad Hoc Network (MANET) Characteristics 8

2.2. Current MANET Research 8

2.3. Commercial Products 9

2.3.1. Bluetooth 9

2.3.2. IEEE 802.11b 10

2.3.3. Current Products Comparison 10

3. picoNet II Specifications **12**

3.1. Functional Overview of picoNet II 12

3.2. Routing Protocol Requirements 13

3.3. Software Requirements 14

3.4. Hardware Requirements 15

3.5. System Block Diagram 15

3.6. Summary 15

4. System Selection **17**

4.1. Operating System Selection 17

4.1.1. Windows Pocket PC 17

4.1.2. NetBSD 17

4.1.3. Linux 18

4.1.4. Operating System Choice 18

4.2. Mobile Handheld Selection 19

4.2.1. Palm OS based PDAs 20

4.2.2. Pocket PC based PDAs 20

4.2.3. Mobile Handheld Choice 20

4.3. Wireless Interface Selection 21

4.3.1. Bluetooth 22

4.3.2. IEEE 802.11b 22

4.3.3. Wireless Card Choice 23

4.4. Summary 23

5. Routing Protocol Implementation **24**

5.1. Routing Protocols 24

5.1.1. Ad hoc On-demand Distance Vector (AODV) Routing 24

5.1.2. Temporally Ordered Routing Algorithm (TORA) 25

5.1.3. Dynamic Source Routing (DSR) 25

5.1.4. Protocol Selection 25
5.2. DSR Protocol Implementation Details 26
5.2.1. Route Discovery 26
5.2.2. Packet Forwarding 27
5.2.3. Route Maintenance 27
5.2.4. Packet Formats 29
5.2.5. Protocol Optimisations 31
5.2.6. Protocol Modifications 31
5.3. System Design Choices 32
5.3.1. The Netfilter Architecture 33
5.3.2. Stack Partitioning 33
5.3.3. Kernel Interfacing 35
5.3.4. Development Environment 36
5.4. Summary 36

6. System Evaluation **37**
6.1. Comparison with Specifications 37
6.2. Routing Protocol Performance 38
6.2.1. Network Characteristics 38
6.2.2. Routing Performance Measures 38
6.2.3. Multi-hop Ad Hoc Capabilities 41
6.2.4. Gateway Capabilities 41
6.3. Personal Evaluation 42
6.4. Summary 43

7. Future Developments **44**
7.1. Implementation Improvements 44
7.2. Routing Protocol Extensions 44
7.3. Global Roaming 45

8. Conclusion **46**

A. Source Code Listings **47**

A.1. Software License 47
A.2. Makefile 47
A.3. dsr.h 48
A.4. dsr_header.h 49
A.5. dsr-kmodule.h 52
A.6. dsr-kmodule.c 53
A.7. dsr_debug.h 64
A.8. dsr_debug.c 65
A.9. dsr_input.h 66
A.10. dsr_input.c 66
A.11. dsr_output.h 71
A.12. dsr_output.c 72
A.13. dsr_queue.h 86
A.14. dsr_queue.c 87
A.15. dsr_route.h 89
A.16. dsr_route.c 89

References **97**

List of Figures

2.1. The OSI reference model and the Internet reference model 5

2.2. Plantronics M1000 Bluetooth headset 10

3.1. Network stack traversal in picoNet II 13

3.2. TCP/IP stack 13

3.3. picoNet II system block diagram 16

4.1. Pocket Internet Explorer running on Pocket PC 18

4.2. Linux running QPE on the Compaq iPAQ 19

4.3. Sony Clie running Palm OS 20

4.4. Compaq iPAQ with 802.11b network card running Linux 21

4.5. 3Com Bluetooth network interface card 22

4.6. Lucent Orinoco 802.11b network card 22

5.1. Route Discovery Process 27

5.2. Packet Forwarding Process 28

5.3. Route Maintenance Process 29

5.4. DSR and IP packet structure 30

5.5. IP Header Format 30

5.6. Fixed Portion of the DSR Header 30

5.7. Route Request Option Format 31

5.8. Linux networking layers for TCP/IP 34

5.9. Netfilter hooks for IPv4 34

6.1. Some tested network topologies 41

6.2. iPAQ with its wireless card wrapped in aluminium foil 42

List of Tables

2.1. Comparison of table-driven and on-demand protocols 9
2.2. Features of commercial ad hoc products 11

6.1. Network Characteristics 38
6.2. DSR Protocol Delays 39
6.3. DSR Packet Data Overhead 39
6.4. DSR Packet Overhead 40
6.5. DSR Protocol Parameters 40

1. Introduction

This chapter introduces the thesis project and the vision it has. It outlines the problem this thesis addressed and the solution developed. The chapter concludes with a description of the thesis document structure.

From the “II” in picoNet II one can correctly assume that this thesis was a continuation of a previous thesis project. picoNet II was indeed a continuation of picoNet ([1, 2]) but only in the sense of the goal of the thesis and not the design or implementation.

1.1. The picoNet II Vision

Pervasive computing is computing in an environment where users will be able to access information without going out of their way. To achieve this, the users must be surrounded by technology without knowing so. Pervasive computing is a trend that is currently driving, and will continue to drive many technologies.

The vision of picoNet II is to create a pervasive network where the underlying technology is invisible and transparent to the user. To achieve this, picoNet II will be designed to be compatible with existing networks and networking standards. It will also be designed to work on commonly available hardware and software platforms.

1.2. Problems with Current Mobile Networks

As users are increasingly mobile it is more and more common for users to meet and communicate without prior planning, and in environments where there is little or no networking infrastructure. For example, business meetings often require documents to be exchanged and it could happen in a cafe or at the airport. In such situations it is difficult

and inconvenient to set up a local area network (LAN) as the network will need to be created on the fly. Such a network is known as an ad hoc network where the network is of a dynamic nature without centralised administration.

Current technologies can form ad hoc networks but is limited in that only single hop networks can be formed. This means that each node can only act as a host sending directly to the destination. In a multi-hop ad hoc network, all nodes act as routers and neighbouring nodes will forward packets to the final destination.

This thesis concentrated on the problem of adding multi-hop capabilities to existing ad hoc network platforms.

1.3. Proposed Solution

An ad hoc network is a network that can be formed without the need for any preexisting networking infrastructure. Mobile ad hoc networks (MANET) [3] describes such a network. The IETF MANET Working Group specifies many routing protocols which will allow the formation of mobile ad hoc networks.

FIXME.

This thesis describes the design and implementation of picoNet II, a mobile ad hoc network that enables handheld devices to form a dynamic, self-organising and self-configuring network. To be in line with the picoNet II vision of being compatible with existing networks and networking standards, picoNet II will be designed to be compatible with TCP/IP[1] .

1.4. Thesis Structure

This thesis dissertation describes the design, implementation and testing of an ad hoc network, picoNet II.

Chapter 2 covers network models and theory relevant to mobile networks. It will also provide reviews on current MANET research and commercial ad hoc networking products.

Chapter 3 specifies the features and functionalities of the picoNet II system. The specification will state the requirements of a mobile ad hoc network and will also specify in detail, parts of the picoNet II system.

[1]Transmission Control Protocol/Internet Protocol

Chapter 4 will describe the hardware and software platform used and how that platform was chosen for the picoNet II project.

Chapter 5 will describe the software implementation of the picoNet II system, namely the routing protocol. It will also cover routing protocol selection, routing protocol operation and how the implementation interfaces with the chosen platform.

The picoNet II system is evaluated in Chapter 6 against the specifications outlined in Chapter 3. Performance issues and functionality of the implementation will be covered in the evaluation.

Chapter 7 will suggest improvements and extensions for future development and the thesis will conclude with a brief summary in Chapter 8.

2. Background

This chapter reviews theory and technologies which are related to the picoNet II project. Firstly, background information will be provided on network models and theory relevant to ad hoc networks. Then a review on current research in ad hoc networks will be presented. Finally the chapter will review and compare current ad hoc networking products.

2.1. Mobile Networks

A mobile network consists of mobile devices, herein simply referred to as "nodes", which are free to move about [3]. The way in which mobile networks operate is fundamentally different to traditional fixed networks. In order to understand these differences, and the challenges of designing and implementing a mobile network, some background information needs to be presented. Network models and the concept of routing will be presented first. Then the characteristics of mobile ad hoc networks will be compared to fixed wire networks.

2.1.1. Network Reference Models

A computer network is a collection of computers connected by some link which supports data transfer. Designing a computer network to provide various types of connectivity across large numbers of hosts imposes many challenges to the designer. Network reference models help designers deal with these design challenges by abstracting functionality into a layered architecture. Two important network architectures, the OSI reference model and the Internet reference model, will be discussed in the following sections.

OSI	Internet
Application	Application
Presentation	
Session	
Transport	Transport
Network	Internet
Link	Host-to-network
Physical	

Figure 2.1.: The OSI reference model and the Internet reference model

2.1.1.1. The OSI[1] Reference Model

The OSI model is shown in Figure 2.1 and it is based on a proposal developed by the International Standards Organisation (ISO). The model is called ISO Open Systems Interconnection Reference Model because it deals with connecting open systems - that is, systems that are open for communication with other systems [4].

There are seven layers in the OSI model and they are described below.

Physical Layer The physical layer forms the lowest layer in the OSI model and it is concerned with transmitting raw bits over a communications channel. The design issues in this layer are mostly to do with mechanical, electrical and procedural interfaces of the underlying transmission medium.

Link Layer The task of the link layer is to group raw bits into frames and provide a channel which is free of undetected errors to the network layer. This is accomplished via error detection and correction schemes, and acknowledgements. The link layer also regulates access to the physical layer and this is sometimes described as a sublayer called the MAC[2] layer.

[1] Open Systems Interconnection
[2] Medium Access Control

Network Layer The network layer deals with how one packet gets from the source to the destination. This process is known as routing. This layer is also responsible for connecting networks of different types together, providing a host-to-host connectivity to the upper layers.

Transport Layer The main purpose of the transport layer is to provide an end-to-end communication channel to the upper layers. It deals with segmenting data streams, flow control and the reliability of data transfers.

Session Layer The session layer provides users on different machines with the ability to establish a session between them. The layer manages connections within the session, which may be one way or two way.

Presentation Layer This layer performs functions which are often required in network communications. These functions include integer conversions, data compression and encoding, and encryption.

Application Layer The application layer sits between the users and the network resources and it provides users with network functionality via applications.

2.1.1.2. The Internet Reference Model

The Internet reference model, also known as the TCP/IP reference model, is a four layer network model. Figure 2.1 shows the Internet model and the OSI reference model side by side.

The four layers of the Internet model are described below:

Host-to-network Layer This layer provides a standard interface of the underlying hardware to the upper layers. It's functionality is similar to the combination of the physical and link layers of the OSI model.

Internet Layer The Internet layer is analogous to the OSI network layer and provides host-to-host connectivity and routing between multiple network technologies.

Transport Layer The transport layer provides end-to-end connectivity like its OSI counter part. This layer defines a reliable connection oriented protocol TCP (Transmission Control Protocol), which has flow control, and another protocol UDP (User Datagram Protocol), which is an unreliable connection-less protocol.

Application Layer The application layer's function is similar to the combination of the application, presentation and session layers of the OSI model. The session and presentation functionality is still present in this but but it is not clearly defined.

2.1.1.3. The OSI model vs The Internet Model

As seen from previous sections, both the OSI model and the Internet model are abstractions of networking functionality. The models differ in the way the abstraction is done, as the Internet model has less layers yet describing the same functionality. The OSI model is more general as it can describe any network, but due to many reasons, both technical and non-technical, it was never implemented. The Internet model on the other hand is used widely today. Since most of the networking technology is based on the Internet model, it will be used to define the picoNet II system.

2.1.2. The Routing Concept

Routing is defined as the process of finding a path from a source to some arbitrary destination on the network. This process operates in the Internet layer so packets can be forwarded across networks with different transmission mediums. The Internet is a collection of numerous networks and subnetworks, which are interconnected by computers which perform routing. Computers which perform routing are known as routers, and a router which routes between a subnet and an external network is known as a gateway.

Traditional routing assumes that all computers on the network are static or semi-static. Therefore the router only need to react to changes caused by failure of network links or other routers. So routers conventionally exchange routing information with other routers by periodically sending out routing specific control messages. Due to the fact that the topology of a fixed network are semi-static, the amount of control messages can be kept to a minimum.

Many assumptions for fixed networks are not valid for mobile networks as there are fundamental differences in how they operate. These differences are discussed in the next section.

2.1.3. Mobile Ad Hoc Network (MANET) Characteristics

“A "mobile ad hoc network" (MANET) is an autonomous system of mobile routers (and associated hosts) connected by wireless links–the union of which form an arbitrary graph. The routers are free to move randomly and organise themselves arbitrarily; thus, the network’s wireless topology may change rapidly and unpredictably. Such a network may operate in a stand alone fashion, or may be connected to the larger Internet.” [5]

The fundamental difference between fixed networks and MANET is that the computers in a MANET are mobile. Due to the mobility of these nodes, there are some characteristics that are only applicable to MANET. Some of the key characteristics are described below [3] :

1. Dynamic Network Topologies: Nodes are free to move arbitrarily, meaning that the network topology, which is typically multi-hop, may change randomly and rapidly at unpredictable times.
2. Bandwidth constrained links: Wireless links have significantly lower capacity than their hardwired counterparts. They are also less reliable due to the nature of signal propagation.
3. Energy constrained operation: Devices in a mobile network may rely on batteries or other exhaustible means as their power source. For these nodes, the conservation and efficient use of energy may be the most important system design criteria.

The MANET characteristics described above imply different assumptions for routing algorithms as the routing protocol must be able to adapt to rapid changes in the network topology. They also present different optimisation parameters such as bandwidth overhead and energy usage. A considerable amount of research has been done in the area of MANET, and this is presented below.

2.2. Current MANET Research

Mobile ad hoc networks, or MANET, are fundamentally different to traditional wired networks as wired networks are assumed to be stationary and static. This imposes different design requirement and constraints on routing protocols for MANET.

There are two categories of routing protocols: table-driven and on-demand routing. In table-driven routing protocols, routing information is periodically advertised to all nodes

	Table-driven	On-demand
Availablilty of Routing Information	Immediately from route table	After a route discovery
Route Updates	Periodic Advertisments	When requested
Routing Overhead	Proportional to the size of the network regardless of network traffic	Proportional to the number of communicating nodes and increases with increased node mobility

Table 2.1.: Comparison of table-driven and on-demand protocols

so all nodes have an up to date view of the network. Alternatively, on-demand routing protocols only discovers a new route when it is required. Hybrid routing protocols also exist and they try to achieve an efficient balance between both categories of protocols [2, 6]. Table 2.1 shows a comparison between the two methodologies.

It is clear that on-demand protocols are more suited for mobile handheld devices as network bandwidth and battery power is limited. This saving in network bandwidth and energy consumption is a tradeoff for up to date routing information. Generally speaking, on-demand routing protocols have longer route discovery delays than table-driven protocols. On-demand routing protocols will be discussed in detail in chapter 5.

2.3. Commercial Products

There are a number of wireless products available, but only a few technologies have ad hoc capabilities. Products based of these technologies will be discussed below.

2.3.1. Bluetooth

Bluetooth is a technology that promises fast, secure, point-to-point wireless communications over short distances (approximately 10 metres) for devices as diverse as mobile phones, consumer electronics appliances and desktop computers [1, 7, 8]. It uses spectrum in the unlicensed ISM[3] band of 2.4 to 2.48GHz. Besides being a hardware standard, Bluetooth defines a protocol stack that allows for hierarchical ad hoc networking in the form of “piconets”, in which Bluetooth devices form themselves into point-to-multipoint picocells of seven slaves under the control of one master. Multiple piconets in overlapping coverage areas form “scatternets”.

[3]Industrial, Scientific and Medical

Figure 2.2.: Plantronics M1000 Bluetooth headset

Although Bluetooth has been standardised for quite some time, the devices are just beginning to become available. The Bluetooth devices which are currently available are only single hop devices as the formation of "scatternets" is not specified in the current version of the Bluetooth standard. Figure 2.2 shows a wireless headset based on Bluetooth technology from Plantronics.

2.3.2. IEEE 802.11b

IEEE 802.11b is wireless local area network communications standards that operates in the 2.4GHz band at data rates of 1 to 11Mbps and distances of 25 to 550 metres [9]. In an IEEE 802.11 network, there are two possible modes: ad hoc mode, where all nodes in the network must be within range of each other, and the infrastructure mode, in which all inter-node communication must pass via access points.

The ad hoc mode allows nodes to form an ad hoc network, but the communication is limited to single hop, with no multi-hop capabilities. Since the IEEE 802.11 standard only defines the host-to-network layer, it is up to upper layer protocols to incorporate multi-hop capabilities. Unlike Bluetooth, IEEE 802.11b products are widely available and are currently used by many corporations and institutions.

2.3.3. Current Products Comparison

From the product comparison shown in Figure 2.2, it can be seen that current ad hoc networking solutions are limited to single hop operation. The network range can be slightly extended with the use of access points but that requires preexisting networking infrastructure to be present.

picoNet II provides a solution to this problem by adding multi-hop routing capabilities to existing ad hoc networks. This will allow a multi-hop network to be deployed with no

	IEEE 802.11b	Bluetooth
Bit Rate	1 - 11 Mbps	1 Mbps
Range	25 - 550 m	10 m
Ad Hoc Capabilities?	Only single hop, not multi hop	Only single hop (Multi hop not specified)
Cost	~AUD$300	~AUD$300

Table 2.2.: Features of commercial ad hoc products

preexisting networking infrastructure. It will also allow a multi-hop ad hoc network to form as an extension to an existing network. Chapter 3 will outline the specifications for the picoNet II system.

3. picoNet II Specifications

This chapter specifies the features and functionality of picoNet II, a wireless ad hoc network for mobile devices. The overall functionality will be outlined first followed by specifications of the routing protocol, hardware, and software platform. Finally the system block diagram will be presented.

3.1. Functional Overview of picoNet II

The function of the picoNet II system is to provide multi-hop capabilities to existing ad hoc networks. For compatibility purposes this functionality should be implemented on the TCP/IP network standard. Any two nodes in the system should be able to communicate across a wireless medium, with end-to-end connectivity achieved by point-to-point packet forwarding at intermediate router nodes [1]. The system should be able to dynamically adapt to node mobility, and nodes entering and leaving the network. Figure 3.1 shows how packets traverses through the TCP/IP stack in the picoNet II system. The multi-hop routing protocol operates at the Internet layer (shaded in grey) of the stack.

The picoNet II systems assumes that the network is relatively small, with a network diameter[1] of around 15. This assumption eases the scalability requirement as the system is just a proof of concept.

There are three major sections of the system that needs to be specified and they are routing protocol, hardware and software. The hardware and software specifications will depend on the routing protocol requirements and will also be interdependent. The following sections will describe the specifications of the routing protocol, hardware, and software in detail.

[1] Network diameter is the maximum number of hops from one end of the network to the other.

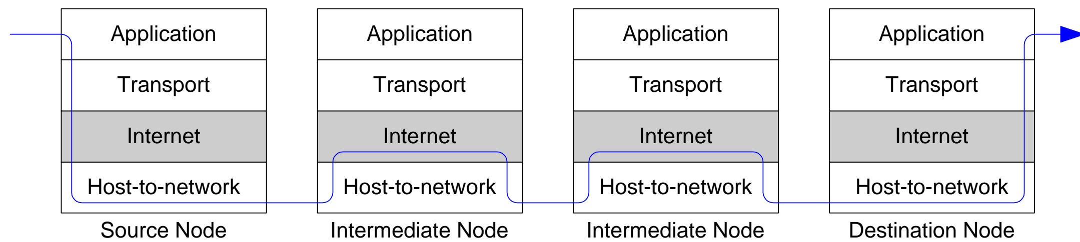

Figure 3.1.: Network stack traversal in picoNet II

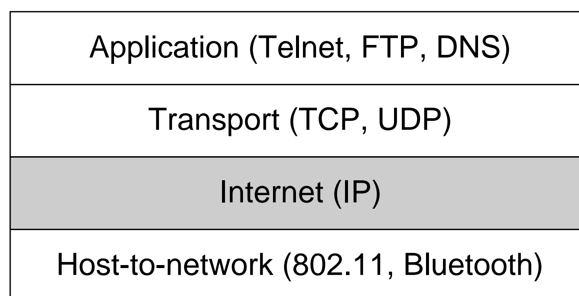

Figure 3.2.: TCP/IP stack

3.2. Routing Protocol Requirements

The function of the routing protocol is to provide multi-hop capabilities. The routing protocol and its implementation must be fully compatible with existing TCP/IP networks. To achieve this compatibility, the routing protocol will need to operate in the Internet layer maintaining compatibility with the transport layer and the host-to-network layer. Figure 3.2 shows this structure with the Internet layer shaded in grey. This compatibility will allow existing applications, transport protocols and network interfaces to operate without modifications.

There are some key requirements in mobile ad hoc networks (MANET) and they are listed below [3] :

- Distributed operation: This is an obvious and necessary property of MANET.
- Loop-freedom: Not a required property but desirable for any routing protocol, as it avoids packets spinning around in the network causing performance to degrade.
- On demand operation: Instead of maintaining routing information between all nodes at all times, the routing information is generated on a demand or need basis. Although it may increase the route discovery delay, it utilises network energy and

bandwidth resources more efficiently. This efficiency is especially important for mobile devices where bandwidth and energy resources are limited.

There are also some performance measures of ad hoc routing protocols and they are described below:

- Route acquisition time: A measure of the time taken to discover a new route.
- Packet data overhead: The amount of extra data exchanged for ad hoc routing to operate.
- Packet overhead: The number of extra packets sent by the ad hoc routing protocol.

These performance measures ultimately affect the end-to-end delay experienced by users. The routing protocol should be selected to be meet the requirements of MANET with minimised end-to-end delay.

3.3. Software Requirements

The key design philosophy behind picoNet II is compatibility, and the software requirements will reflect that philosophy. Routing protocols are implemented as a part o the operating systems (OS) and often inside the kernel for increased performance. Therefore the software requirements will be in the context of the operating system.

The following is a list of requirements for the operating system:

- Network enabled: This is an obvious and needed requirement. It means the OS must have support for networking devices and more specifically a TCP/IP implementation.
- Network applications: This is required for the system to demonstrate seamless multi-hop operation.
- Hardware compatibility: The OS must be available for the hardware platform and must have driver support for the wireless network interfaces.
- Access to source code: The routing protocol will be implemented inside the operating system, so it is essential to have access to the operating system source code. Without the source code the routing protocol will need to be implemented in user space, which will result in incompatibilities with existing applications.

3.4. Hardware Requirements

The hardware requirements can be derived from the software specifications as their requirements are interdependent. These requirements are listed below:

- Mobile and portable.
- Support for wireless network interfaces.
- Compatible with chosen operating system.
- Widely available.

The wireless network interface should be based on radio frequency (RF) technology as they do not require line-of-sight. Other wireless technologies like infrared, which require line-of-sight, are not well suited to ad hoc applications as communication depends on the orientation of the user and the device.

3.5. System Block Diagram

The requirements outlined in previous sections can summarised in a system block diagram shown in Figure 3.3. The system block diagram shows the structure of the TCP/IP stack in relation to the operating system and the underlying hardware. Various parts of the TCP/IP stack will already have been implemented in the operating system. The IP[2] layer, which is shaded in grey, will be where the picoNet II implementation will reside. It is logical for picoNet II to be at the IP layer since all routing is performed there. No changes will be made to the interface between the IP layer and the other layers, as all changes will be internal to the IP layer. This design will allow picoNet II to be completely compatible with existing systems, allowing it to operate with existing network technologies and applications seamlessly.

3.6. Summary

The specifications for the routing protocol, hardware and software of the picoNet II system was outlined in this chapter. It was specified that the routing algorithm should be

[2] Internet Protocol

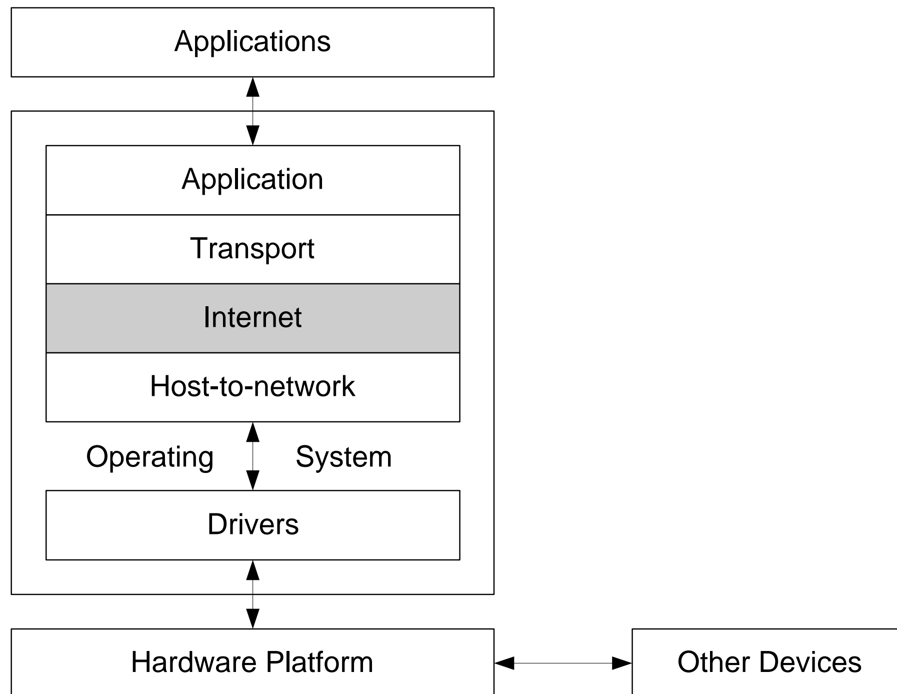

Figure 3.3.: picoNet II system block diagram

implemented for TCP/IP in the Internet layer within the operating system. The system should also be implemented on mobile and portable hardware platforms. The selection of the hardware and software platform will be described in the following chapter.

4. System Selection

This chapter will describe the hardware and software used and how that platform was chosen for the picoNet II project. Firstly, operating system selection will be described in detail followed by mobile handheld and wireless interface selection.

4.1. Operating System Selection

This section will describe a few operating systems which meet the software requirements outlined in the previous chapter. The advantages and disadvantages of each operating system are detailed below.

4.1.1. Windows Pocket PC

Pocket PC is a commercial operating system based on Windows which is pre-installed on many handheld computers on the market today. The operating system is network enabled with many network applications available. One such application is Pocket Internet Explorer and it is shown in Figure 4.1. The Pocket PC OS also has driver support for wireless network interfaces.

Pocket PC, being designed for the consumer market, does not have network routing support built into the OS. The source code to the Pocket PC operating system is available for developers, but due to the commercial nature of the OS, at a cost.

4.1.2. NetBSD[1]

NetBSD is a UNIX operating system derived from BSD and it is available on many hardware platforms including handheld computers and embedded systems [10]. Being a UNIX

[1] Berkeley Software Distribution

Figure 4.1.: Pocket Internet Explorer running on Pocket PC

operating system, NetBSD is a network enabled OS with full TCP/IP support and network routing capabilities. It also has driver support for wireless network interfaces as well as many networking applications. The source code for NetBSD is freely available under the BSD license ([11]). The GNU[2] development environment is used for NetBSD.

4.1.3. Linux

Linux is another UNIX operating system and, like NetBSD, it is available on many different hardware platforms from PCs to handheld computers. Like other UNIX operating systems, Linux supports TCP/IP, routing and network applications. There is also support for various wireless network interfaces. An environment called the QT Palmtop Environment (QPE) is designed for handheld computers running Linux and a screen shot is shown in Figure 4.2. The source code is fully available under the GPL licence ([12]) and the GNU programming tools are used for kernel development.

4.1.4. Operating System Choice

All three operating systems met the requirements outlined in chapter3. The selection process was based on two criterion, ease of implementation and cost. This rules out the Pocket PC operating system straight away as the source code and development tools are not freely available.

[2]GNU is Not UNIX

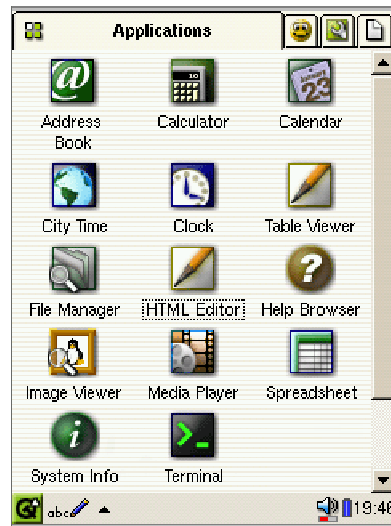

Figure 4.2.: Linux running QPE on the Compaq iPAQ

Of the remaining choices, NetBSD and Linux, there were not great differences as both operating systems are open source UNIX operating systems. Linux was chosen in the end for reasons listed below:

- A new packet mangling framework called netfilter, a new feature for Linux 2.4, which will allow the routing protocol to be implemented with more ease. The netfilter architecture will be discussed in detail in section 5.3.1.
- I have had previouse experience with the Linux OS and no experience with NetBSD. Choosing Linux will result in a smoother learning curve, reducing development time.
- On handheld platforms, Linux is actively developed by Compaq and their research labs, and this active development will result in a more stable and better supported OS.

4.2. Mobile Handheld Selection

This section will describe handheld computers which meet the hardware requirements outlined previously and has support for Linux. The advantages and disadvantages of each handheld platform are detailed below.

Figure 4.3.: Sony Clie running Palm OS

4.2.1. Palm OS based PDA[3]s

Palm OS based handheld computers such as the Sony Clie shown in Figure 4.3are widely available [13]. A variant of Linux called uClinux has been successfully ported to a few Palm OS devices such as Palm IIIe and the Handspring Visor [14]. uClinux or micro-controller Linux is a derivative of the Linux 2.0 kernel for micro-controllers without a MMU[4] [15]. There are currently many wireless network interfaces available for Palm OS based devices, including IEEE 802.11b and Bluetooth. At the time of hardware selection these wireless products were only announced, but not available.

4.2.2. Pocket PC based PDAs

Pocket PC based PDAs such as the Compaq iPAQ, shown in Figure 4.4, are also widely available. The Pocket PC devices are more powerful than the Palm OS devices as they have faster CPUs and have more storage space, but shorter battery life. There are many different wireless network solutions available for Pocket PC based handheld computers. Some handhelds, like the Compaq iPAQ, have PCMCIA[5] adaptors which allows any PCMCIA wireless network card to be used, so long as drivers exist. Full featured Linux is available on most Pocket PC based devices at handhelds.org ([16]).

4.2.3. Mobile Handheld Choice

Both types of handheld computers had the functionality described in chapter 3. The selection criteria was again, ease of implementation and cost. There are no major differences

[3]Personal Digital Assistant
[4]Memory Management Unit
[5]Personal Computer Memory Card International Association

Figure 4.4.: Compaq iPAQ with 802.11b network card running Linux

in the functionality between the two families of handheld computers. Despite the higher cost of Pocket PC PDAs, it was chosen over the Palm OS devices for its ease of implementation. The reasons are discussed below:

- Palm OS devices only supported uClinux which lacked many features in Linux 2.4 such as netfilter.
- Pocket PC devices had PCMCIA support which provided a wider range of wireless network interfaces.
- Linux for Pocket PC devices was more actively developed than uClinux for Palm OS devices.

There are many different Pocket PC devices and the Compaq iPAQ, which is ARM[6] based, was chosen over the other devices because of its PCMCIA support and its extensive support for the Linux operating system. It must be noted that Compaq Australia sponsored this thesis project by providing two iPAQs.

4.3. Wireless Interface Selection

The wireless interface selection is narrowed down to PCMCIA cards with driver support for Linux running on the Compaq iPAQ.

[6] Advanced RISC Machines

Figure 4.5.: 3Com Bluetooth network interface card

Figure 4.6.: Lucent Orinoco 802.11b network card

4.3.1. Bluetooth

Bluetooth network cards are relatively new to the market and at the time of selection there was only driver support for Windows running on PCs. There was also compatibility issues between Bluetooth network cards from different vendors. Choosing Bluetooth as the wireless network interface for picoNet II could have introduced unnecessary difficulties. A newly released Bluetooth network card from 3Com is shown in Figure 4.5.

4.3.2. IEEE 802.11b

In contrast to Bluetooth, IEEE 802.11b is a more mature technology with driver support for a variety of operating system. There is strong support for 802.11b network cards in Linux running on PCs and the support is more limited on Linux for the iPAQ as only two chipsets are supported. The supported chipsets are from Lucent and Intersil. The Lucent IEEE 802.11b network card is shown in Figure 4.6. At the time of research the Lucent drivers were more mature and had ad hoc (ad hoc in the 802.11 sense, single hop only) support where the Intersil driver was only functional in infrastructure mode.

4.3.3. Wireless Card Choice

The Bluetooth network cards were ruled out due to its lack of Linux driver support. Out of the two IEEE 802.11b cards, the Lucent network card was chosen as it had the functionality required at the time of selection.

4.4. Summary

This chapter covered the selection process for the picoNet II system. The chosen platform was the Compaq iPAQ handheld computer running the Linux operating system. The Lucent IEEE 802.11b network card was chosen to provide wireless connectivity. The selection and implementation of the routing protocol will be discussed in the next chapter.

5. Routing Protocol Implementation

This chapter represents the core of the thesis project as it describes the implementation of the routing protocol. Firstly, several MANET routing protocols will be discussed and a protocol will be chosen to be implemented. Then the chosen routing protocol will be discussed in detail and any changes made to the routing protocol will also be presented. Finally, the chapter will describe the choices made in implementing this routing protocol, and more specifically how the implementation interfaces with the Linux operating system.

5.1. Routing Protocols

As discussed earlier in section 2.2, on-demand routing protocols are better suited to ad hoc networks formed by handheld devices. In the following section, several on-demand routing protocols will be compared.

5.1.1. Ad hoc On-demand Distance Vector (AODV) Routing

Ad hoc On-demand Distance Vector Routing (AODV) is an on-demand version of the table-driven Dynamic Destination-Sequenced Distance-Vector (DSDV) protocol [6]. To find a route to the destination, the source broadcasts a route request packet. This broadcast message propagates through the network until it reaches an intermediate node that has recent route information about the destination or until it reaches the destination.

When intermediate nodes forwards the route request packet it records in its own tables which node the route request came from. This information is used to form the reply path for the route reply packet as AODV uses only symmetric links. As the route reply packet traverses back to the source, the nodes along the reverse path enter the routing information into their tables. Whenever a link failure occurs, the source is notified and a route discovery can be requested again if needed.

5.1.2. Temporally Ordered Routing Algorithm (TORA)

The Temporally Ordered Routing Algorithm (TORA) is a highly adaptive, efficient and scalable routing algorithm [6]. It is a source-initiated on-demand protocol and it finds multiple routes between the source and the destination. TORA is a fairly complicated protocol but its main feature is that when a link fails the control messages only propagates around the point of failure. While other protocols need to re-initiate a route discovery when a link fails, TORA would be able to patch itself up around the point of failure. This feature allows TORA to scale up to larger networks but has higher overhead for smaller networks.

5.1.3. Dynamic Source Routing (DSR)

The Dynamic Source Routing (DSR) protocol is a source-routed on-demand protocol [6, 17]. There are two major phases for the protocol: route discovery and route maintenance. The key difference between DSR and other protocols is the routing information is contained in the packet header. Since the routing information is contained in the packet header, the intermediate nodes do not need to maintain routing information. An intermediate node may wish to record the routing information in its tables to improve performance, but this is not mandatory.

Another feature of DSR is that it supports asymmetric links as a route reply can be piggybacked onto a new route request packet. DSR is suited for small to medium sized networks as its packet overhead (not packet data overhead) can scale all the way down to zero when all nodes are relatively stationary. The packet data overhead will increase significantly for networks with larger hop diameters as more routing information will need to be contained in the packet headers.

5.1.4. Protocol Selection

Out of all the routing protocols, TORA was the most complex and also the most scalable. These properties of TORA may be ideal for an ad hoc routing protocol, but it was not preferred for this thesis as ease of implementation was one of the key selection factors.

The main difference between AODV and DSR was the way the routing information was exchanged. In AODV the information was stored at each node where as in DSR the routing information was included in each packet. Simulation results have shown that

AODV and DSR have similar performance with DSR being more efficient with higher node mobility and AODV more efficient at lower node mobility [18].

All three protocols met the criteria outlined in section 3.2, and DSR was chosen because it was the most efficient in terms of bandwidth requirements and hence energy consumption. The implementation of the DSR protocol will be detailed in the next section.

5.2. DSR Protocol Implementation Details

This section describes the implementation details of the DSR protocol, the operating system specific detail will be described next, in section 5.3. The DSR protocol is based on the Internet-draft from the MANET Working Group ([19]).

Firstly, the operation of the protocol, namely route discovery, packet forwarding and route maintenance, will be described followed by the description of the packet formats and protocol optimisations. Lastly the modifications made to the protocol will be discussed, and this will include extensions and changes made to the protocol.

5.2.1. Route Discovery

Route discovery is the process in which a source node discovers a route through the network to some arbitrary destination node. Every node has a route cache which contains recent routes to other nodes on the network. If a node needs to send information to some destination and a route is found in the route cache then the node will use that route. Otherwise the source node will initiate a route discovery process by sending a route request packet across the network. Figure 5.1 illustrates the route discovery process and the propagation of route request packets.

Every route request packet has a unique identification number. Nodes cache this identification number and discards subsequent route request packets with the same identification number. In the example shown in Figure 5.1, node D received the route request from node C first and it discarded the route request from node B. As the route request propagates through each node, each node adds its own address to the route request if it is not already present. This ensures loop-free routes.

When the route request reaches the final destination, a route reply packet is returned to the source node from the destination node. For asymmetric links the route reply may be

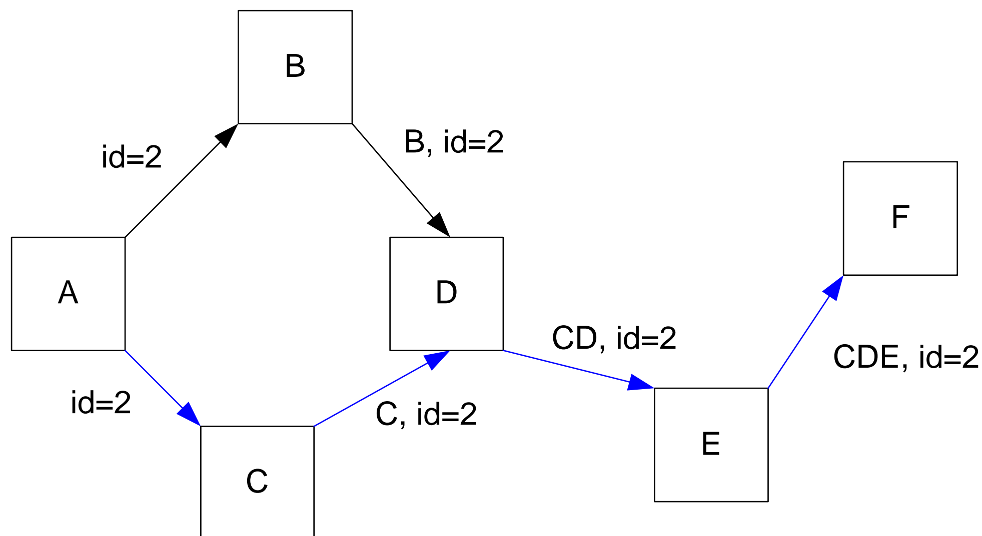

Figure 5.1.: Route Discovery Process

piggybacked onto a new route request. In our system where 802.11b links are bidirectional, the route reply will simply contain the route recorded in the route request packet in reverse order. In the event that the final destination is not present or completely out of range, route requests will be resent by the source node after a timeout which backs off exponentially.

5.2.2. Packet Forwarding

When a node wishes to send packets and has a route to the destination, it adds the full source route to the each of the packets. Along with the source route the number of segments left is also added and source node will initialise this number to the length of the source route. The number of segments left is the number of hops left for the packet to reach its destination and it gets decremented each hop. It is used for intermediate nodes to index the next hop address from the source route so the packet can be forwarded to the next node. The packet forwarding process is illustrated in Figure 5.2.

5.2.3. Route Maintenance

As nodes in a MANET move around, links will be down and routes need to be maintained. This is called route maintenance.

During packet forwarding every node is responsible confirming that the packet was received by the next hop. There are three ways to get this acknowledgement and they are

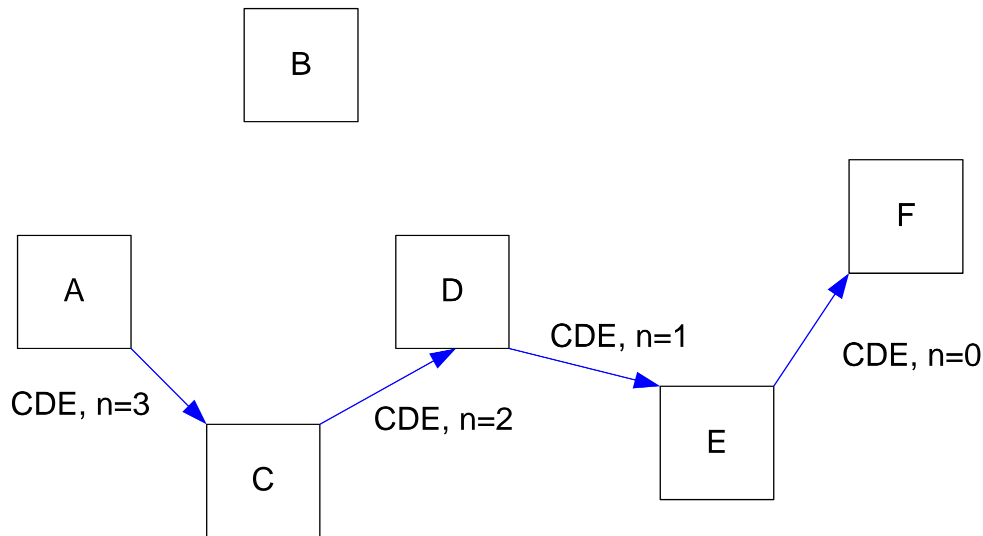

Figure 5.2.: Packet Forwarding Process

listed below:

- MAC layer acknowledgement: this is supplied by the underlying MAC layer and technologies like IEEE 802.11b support it.
- Passive acknowledgement: this confirmation comes from nodes overhearing the next node forwarding the packet. It can be used on every hop except the last hop. This requires the network interfaces to work in promiscuous mode so it can overhear packets sent to other nodes.
- Network layer acknowledgement: this is when the node explicitly request a DSR specific acknowledgement to be returned by the next hop.

Although the most inefficient, network layer acknowledgements were used for picoNet II as it was the easiest to implement. MAC layer acknowledgements require interfacing the routing code with the network interface driver which would introduce unnecessary work. Passive acknowledgements were not feasible as there was little support for promiscuous mode from the network drivers.

When no acknowledgement has been received by the node sending a packet after a set timeout, the packet is resent after a timeout a set number of times. If no acknowledgement is received after the retransmission,then a route error packet will be sent back to the source node to indicate that the link is broken. In the example shown in Figure 5.3, the link from node D to node E is broken so node D will send a route error packet back to node A

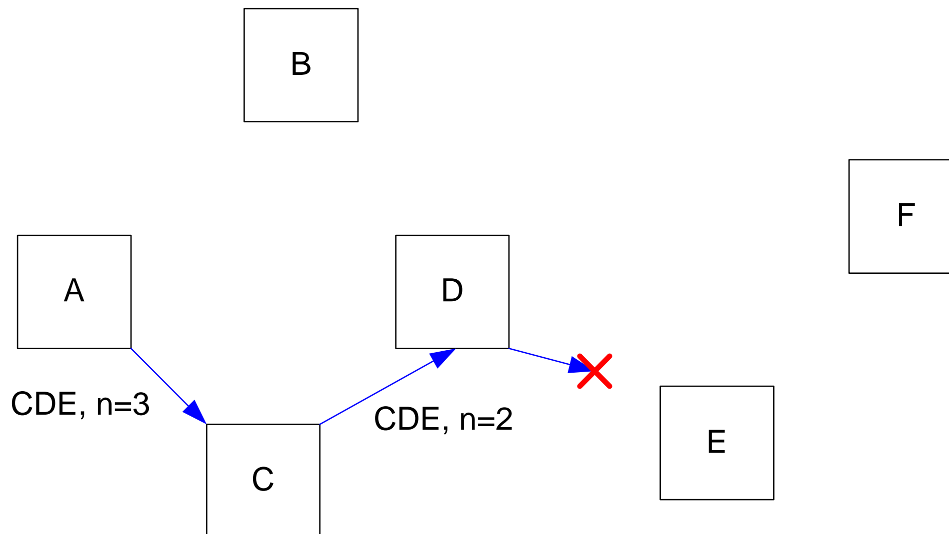

Figure 5.3.: Route Maintenance Process

indicating that link D-E is broken. Upon receipt the route error packet, the source node will update the route cache accordingly. The source node will use another route if it is present in the route cache, otherwise a new route discovery process is initiated.

5.2.4. Packet Formats

The Dynamic Source Routing protocol makes use of a special header, which carries control information, that can be included in any IP packet. The DSR header in a packet contains a fixed sized 4 byte section which is followed by a sequence of zero or more DSR options carrying optional information [19]. The total length of the DSR options is stored in the DSR header. The DSR header is inserted in the packet following the IP header and before any transport layer information. Figure 5.4 illustrates this.

The format of the IP header will not be modified but some fields in the IP header will need to be changed to differentiate a DSR packet from a normal IP packet. Figure 5.5 shows the IP header with the modified fields shaded in grey. The protocol field is changed to a unique number indicating that the packet is a DSR packet. As DSR information is inserted to the packet, the total length of the packet must also be changed. The destination is changed to a broadcast address for route request packets and when any field in the IP header changes, the checksum must be recalculated.

The fixed portion of the DSR header shown in Figure 5.6 and it contains three fields of which two are currently used. The next header field is used to record the IP protocol

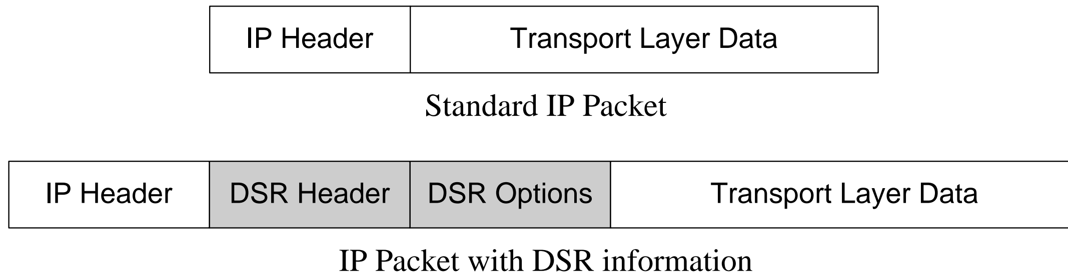

Figure 5.4.: DSR and IP packet structure

<table>
<tr><td>Version</td><td>IHL</td><td>Type of Service</td><td colspan="3">Total Length</td></tr>
<tr><td colspan="3">Identification</td><td>D
F</td><td>M
F</td><td>Fragment Offset</td></tr>
<tr><td colspan="2">Time To Live</td><td>Protocol</td><td colspan="3">Header Checksum</td></tr>
<tr><td colspan="6">Source Address</td></tr>
<tr><td colspan="6">Destination Address</td></tr>
</table>

Figure 5.5.: IP Header Format

number of the original packet, so when the DSR information is removed at the final destination, the original packet can be constructed. This method is completely transparent to the upper protocol layers. The payload length field defines the total length of all the DSR options carried in this packet.

Every DSR option has an option type field and an option length field. The option type field indicates the type of the option which will determine the format of the option. The option length field indicates the size of the option. Figure 5.7 shows and example of a DSR option, the route request option. The option type and option length fields are shaded in grey.

Other DSR options such as, route reply, route error, acknowledgement request, acknowledgement, source route and pad options are specified and described in the DSR Internet-draft ([19]).

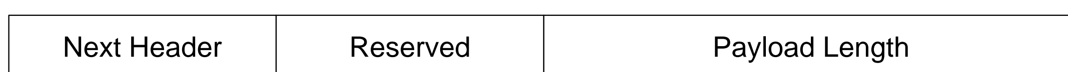

Figure 5.6.: Fixed Portion of the DSR Header

Option Type	Opt Data Len	Identification
Target Address		
Address[1]		
...		
Address[n]		

Figure 5.7.: Route Request Option Format

5.2.5. Protocol Optimisations

Many routing optimisations were outlined in the DSR Internet-draft, but only a few basic optimisations were implemented due to time constraints, and that optimisations were not necessary for the demonstration of the picoNet II proof-of-concept. Many optimisations also required the wireless links to be in promiscuous mode so nodes can cache and process overheard information.

The few basic optimisations that were implemented involved nodes caching routes from packets they received or forwarded. This occurs whenever a node propagates a route request or a route error, and also occurs when a node forwards a packet as they contain route information. These optimisations reduce the number of route discoveries initiated.

Other protocol optimisations will not be discussed in detail as they are simply more sophisticated techniques for reducing routing overhead and they are specified in the DSR Internet-draft ([19]).

5.2.6. Protocol Modifications

There were two modifications made to the DSR protocol. The first one changed the way route discoveries were made to nodes outside of a DSR network. The second change was an extension to the protocol, dealing with IP packet fragmentation. Both of these changes are detailed below.

5.2.6.1. External Node Route Discovery

The specified method for conducting a route discovery to external nodes (nodes outside the DSR network) was to initiate a normal route request with the route reply indicating

the last hop is external. This meant that a route entry was required for every external destination.

In conventional network setups, there is usually one gateway machine per subnet to route between the subnet and external networks. This subnet idea has been applied to the DSR implementation and means that nodes would simply initiate a route discovery to the gateway machine if the destined node is external to the DSR network. This results in one route entry for all external nodes. The subnet method is less flexible than the method specified in the draft, as all DSR nodes have to be within the same subnet and only one gateway can exist, but it is easier to implement.

5.2.6.2. IP Packet Fragmentation

During a packet's traversal through a network it may go through different MAC technologies with different maximum frame sizes. IP fragmentation deals with this by splitting packets which are too big, into smaller fragments and reassembling them again when the destinations is reached.

The current draft for the DSR protocol ([19]) does not support packet fragmentation. In order to demonstrate compatibility with existing applications, fragmentation is needed. Thus fragmentation support was added to the DSR protocol. This extension involved duplicating the DSR information during packet fragmentation and requesting a separate network layer acknowledgement for each fragment.

5.3. System Design Choices

This section describes how the protocol implementation interacts with the rest of the system and more importantly, the choices made in the design process. As mentioned before, the netfilter architecture was one of the deciding factors for operating system selection and it will be presented in detail. Secondly, how the routing protocol is situated in the operating system will be covered followed by a description of how the protocol interfaces with the operating system kernel. The final part of this section will describe the choices made for the development environment.

5.3.1. The Netfilter Architecture

Before going into the details of netfilter, we must first look at the networking structure in Linux. Figure 5.8 shows the Linux networking layers for TCP/IP and the IP layer (shaded in grey) is where the DSR protocol is implemented. The figure also illustrates that applications in user space access network functionality in the kernel via the use of sockets. As outlined in the previous section, the DSR protocol will need to manipulate packets in order to operate correctly.

Netfilter is a framework for packet manipulation at the IP layer and it was introduced in version 2.4 of the Linux kernel [20]. A set of well defined points in a packet's traversal of a protocol stack, called hooks, is defined for each protocol. When a packet passes any of these hooks, the netfilter framework will be called.

Sections of the kernel can register to listen at these hooks for a particular protocol, so when the netfilter framework is called, it checks if any functions have been registered for that particular hook and protocol. If functions are registered at that hook then netfilter will call the functions in order of priority, and if there are no functions registered to listen at a hook, the packet continues its normal traversal.

When the functions are called, it may do anything it wishes to that packet. The packet may be examined, altered, discarded, stolen, queued for user space or continue to traverse through the stack. These functionalities make netfilter a very flexible and extensible packet manipulation framework. Figure 5.9 shows the netfilter hooks defined for IPv4 and the path of packet traversal [21].

The PRE_ROUTE hook is called if the packet received is not a promiscuous receive and the IP checksum was verified. Next the packet is routed to see if the packet is destined for another host or the local host. If the packet is destined for the local host then the LOCAL_IN hook is called before passing the packet to upper layers in the TCP/IP stack. If the packet is to be forwarded onto another host then the FORWARD hook is called. After going through the FORWARD hook, the POST_ROUTE hook is called before the packet is finally sent out. The LOCAL_OUT hook is called for packets created by the local host.

How picoNet II uses netfilter will be covered in the next few sections.

5.3.2. Stack Partitioning

Stack partitioning refers to how a protocol stack is divided with respect to the underlying operating system. In traditional fixed wired routing, the packet forwarding resided in

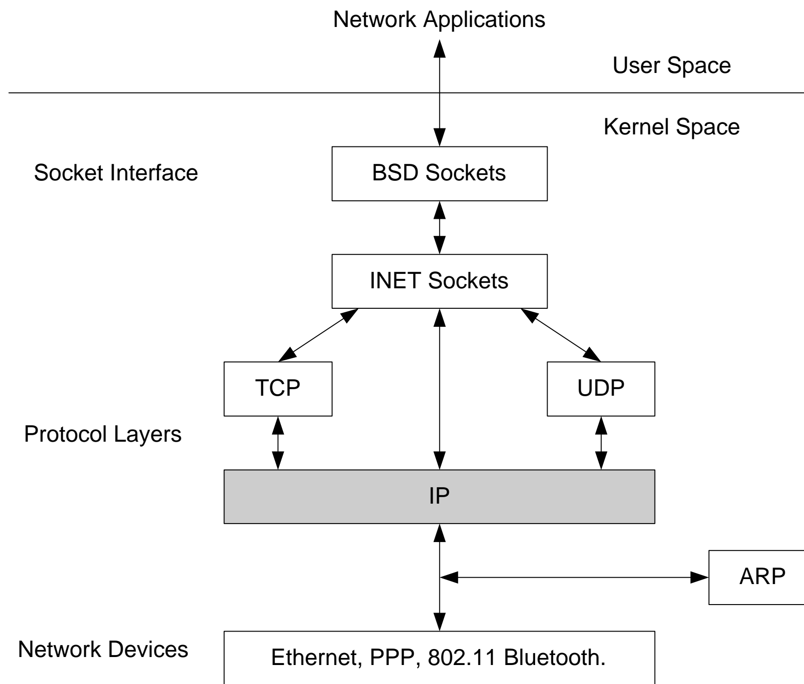

Figure 5.8.: Linux networking layers for TCP/IP

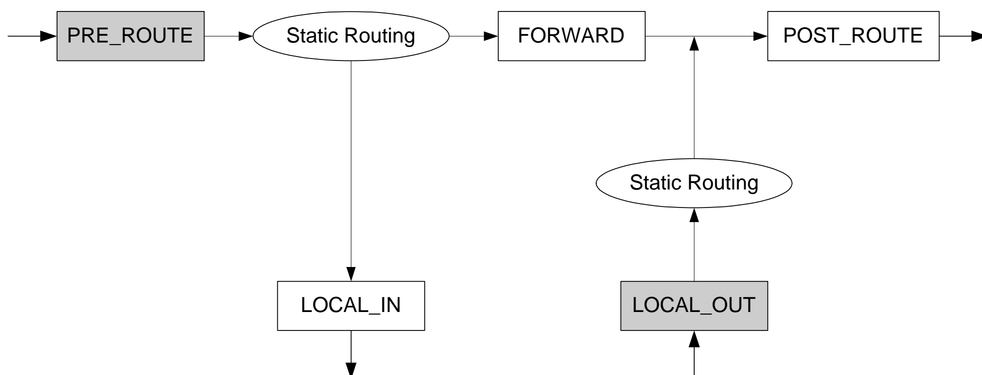

Figure 5.9.: Netfilter hooks for IPv4

kernel space for increased performance and routing table updates ran as a user process so it was easy for system administrators to manage.

picoNet II was originally based on this model with packet forwarding running as a kernel process while route discovery and route maintenance ran as a user process. This would have resulted in the kernel process using netfilter user space queues to communicate with the user process. Early on during the development process, it was realised that having a kernel space and user space process introduced unnecessary work and problems as netfilter user space queueing was buggy and under heavy development. As there are no system administrators for handheld computers, there was no real need to have a user space process.

The stack partitioning was modified so the whole DSR protocol was implemented in kernel space using the netfilter architecture. Referring back to Figure 5.9, the PRE_ROUTE and LOCAL_OUT hooks (shaded in grey) were used for the DSR protocol implementation. An additional hook, the POST_ROUTE hook, was used for the DSR to IP gateway.

The gateway functionality was not specified in the system specifications but it was implemented as it required little extra work. This was due to the fact that static routing capabilities was a part of the Linux kernel, and that the netfilter architecture was built around the static routing. Care had to be taken when developing the DSR protocol implementation to make sure it interacted with the static routing and other netfilter modules correctly.

5.3.3. Kernel Interfacing

There are two ways to add functionality to a Linux kernel, one is to build the functionality into the kernel and the other is to build the functionality as a kernel module. If the functionality is built into the kernel then it will be present every time the machine boots, and a reboot will be required if the kernel is modified. A kernel module on the other hand, can be inserted into and removed from the kernel at runtime, without a reboot. This is one of the most attractive features of the Linux kernel.

The DSR protocol was implemented using kernel modules for two main reasons, to provide users with more flexibility and to reduce development time. It provides users with flexibility by allowing users to enable or disable the DSR module at anytime, users can even change networks and re-enable DSR for the new network without rebooting their machine.

The development time was cut down greatly by using kernel modules. As the DSR module was developed, the test cycle would be to copy the kernel module to the iPAQ, insert the new module and test. If the DSR protocol was built into the kernel then the development cycle would be time consuming as the whole kernel will need to be compiled, downloaded onto the iPAQ, rebooted and then tested.

5.3.4. Development Environment

The development environment used was the Linux kernel development environment. Since the Linux kernel was written in C, the DSR protocol was also implemented in C. The development tools consisted of GNU C compiler tools for both the ARM and the Intel x86 architectures and various tools for the iPAQ from handhelds.org ([16]). These tools were chosen because they were freely available, and they were well supported by the open source community.

The setup consisted of one x86 PC and three Compaq iPAQs, all running Linux 2.4. The code, which was developed on the PC, was tested on both the iPAQs and the PC. The PC was part of a departmental LAN so it acted as the gateway for the DSR network. The DSR gateway module was developed and tested on the PC.

The DSR network used private IP addresses so in the event of buggy development code, the departmental LAN would not be affected. The setup was subject to budget limitations but it was more than adequate for a proof-of-concept project.

5.4. Summary

This chapter reviewed several MANET routing protocol and the Dynamic Source Routing (DSR) protocol was chosen to be the implemented. The operation of the DSR protocol and the modifications made were discussed in detail. Next, the chapter described the Linux networking structure, the netfilter architecture and how netfilter was used for the implementation of the DSR protocol. The chapter then explained how implementing the DSR protocol as a Linux kernel module, provided user flexibility and reduced development time. Finally, reasons for using the GNU development tools, and a private network were described. The performance of the system will be evaluated in the next chapter.

6. System Evaluation

The resulting solution was able to provide multi-hop capabilities and this chapter evaluates the final system both qualitatively and quantitatively. Firstly, the system will be compared with the specifications outlined in chapter 3 followed by an evaluation of the routing protocol performance. The chapter will finish off with a personal evaluation that reviews the design and development process.

6.1. Comparison with Specifications

picoNet II met the specification of providing multi-hop capabilities to existing ad hoc networks. The chosen platform, Linux on the Compaq iPAQ with IEEE 802.11b wireless interfaces, is a widely available platform featuring many network applications. The routing protocol implementation was fully compatible with TCP/IP allowing existing network applications and protocols to operate seamlessly without modifications. Tested applications include ping, traceroute, ssh, ftp, nfs and web browsers.

Some unspecified but desirable features such as the ability to enable and disable the routing protocol on the run, and DSR to IP gateway functionality were developed with little extra work.

One of the reasons the DSR protocol was chosen to be implemented was its efficiency and low overhead. As most of the protocol's optimisations were not implemented, the resulting DSR module was fully functional but not the most efficient. Performance improvements of the DSR module will be discussed in chapter 7 and the performance analysis of the DSR module will be discussed in the next section.

Network Size	Not limited by the routing protocol
Network Diameter	15 hops
Hop Range at 11Mbps	25m indoor, 160m outdoor
Hop Range at 1Mbps	50m indoor, 550m outdoor
Maximum Network Range at 11Mbps	375m indoor, 2.4km outdoor
Maximum Network Range at 1Mbps	750m indoor, 8.25km outdoor

Table 6.1.: Network Characteristics

6.2. Routing Protocol Performance

This section will describe the performance of the implemented DSR protocol. The network characteristics will be outlined followed by details of routing performance measurements. Then the multi-hop capabilities of the network will be evaluated followed by a description of the DSR gateway capabilities.

6.2.1. Network Characteristics

Table 6.1 shows picoNet II network characteristics. The network size is the number of nodes inside the network and this is limited by underlying technologies such as the addressing range of IPv4 and not limited by the routing protocol. As specified in section 3.1 the network diameter was limited to 15 hops so scalability problems would not arise. The hop ranges shown in the table is based on the Lucent IEEE 802.11b wireless interfaces at two particular transmission rates [9]. Different transmission rates and wireless interfaces will result in different ranges obtained. The maximum network range is simply the product of the network diameter and the hop range.

From the maximum network ranges, it is feasible to deploy a campus-wide ad hoc network without additional networking infrastructure. This is quite a good result considering that 3G[1] cellular technologies are yet to deliver its promises.

6.2.2. Routing Performance Measures

Table 6.2 shows the delays introduced by this particular implementation of the DSR protocol. The route acquisition time is the time taken for a node to discover a route and it takes 10 ms for a destination that is only one hop away. Destinations further away will

[1] 3rd Generation

Route Acquisition Time	10+ ms
PRE_ROUTE Hook Delay (200MHz ARM)	20 us - 145 us
LOCAL_OUT Hook Delay (200MHz ARM)	55 us - 180 us
PRE_ROUTE Hook Delay (500MHz Pentium III)	8 us - 140 us
LOCAL_OUT Hook Delay (500MHz Pentium III)	15 us - 150 us

Table 6.2.: DSR Protocol Delays

Data Overhead (n is the number of hops)	4 + 24n bytes per packet
Minimum Data Overhead (n = 1)	28 bytes per packet
Maximum Data Overhead (n = 15)	364 bytes per packet
Percent Data Overhead (n=1, total size of packet=1500)	1.87%
Percent Data Overhead (n=15, total size of packet=1500)	24.27%

Table 6.3.: DSR Packet Data Overhead

have longer acquisition times and it will depend on the network topology at the time of the route discovery.

The table also shows the delay introduced by the extra processing of the packets from the functions hooked at PRE_ROUTE and LOCAL_OUT. A range of times were produced as the functions perform various amounts of packet processing.

Table 6.3 shows the packet data overhead introduced by the DSR protocol implementation. These measurements assume that the nodes are relatively stationary and only looks at overheads introduced by packet forwarding. Overheads introduced by node mobility are not included here and it will be look at below.

The data overhead, which is 4 + 24n bytes per packet, increases as the packet is forwarded over longer paths. The rate of increase, which is 24 bytes per hop, is quite high, resulting in 1.87% to 24.27% of overhead per packet. This inefficiency is due to the use of network layer acknowledgements. If MAC layer acknowledgements were used, the data overhead will become 4 + 4n bytes per packet. The overhead per hop will be reduced to 4 bytes per hop which will translatesto 0.53% to 4.27% of overhead per packet.

Table 6.4 shows the relationship between DSR packet overhead and network dynamics. Qualitative analysis is provided here as quantitative analysis can be difficult. The difficulty comes from the fact that the analysis depends on many factors such as user movement and network usage, not mentioning environmental effects on wireless links. Quantitative performance analysis is more suited to network simulations, which is beyond the scope of this thesis.

The number of route requests sent, heavily depends node mobility and network traffic as DSR discovers routes on a demand basis. The number of route errors sent is also heavily

Route Request Overhead	Increases with node mobility and network traffic
Route Reply Overhead	Increases with the number of route requests sent
Route Error Overhead	Increases with node mobility

Table 6.4.: DSR Packet Overhead

Maximum number of route request resends	16
Maximum route request timeout	10 s
Initial route request timeout	250 ms
Acknowledgement timeout	300 ms
Maximum number of retransmissions	0

Table 6.5.: DSR Protocol Parameters

dependent on node mobility and the number of route replies is proportional to the number of route requests sent. Complications arises from the fact that route errors can cause route discoveries to be initiated. For detailed network simulation results, refer to [18].

Table 6.5 shows the DSR parameters used to obtain optimum balance between network performance, and network adaptability to node mobility. Route requests will be resent a maximum of 16 times with an initial timeout of 250 ms. The timeout doubles each time until it reaches 10 s. This provides a good balance between route acquisition time and bandwidth used.

The network layer acknowledgement timeout is 300 ms with no retransmissions. The reason that the timeout is so long and that there are no retransmissions is due to the 802.11b MAC layer performing retransmissions at the link layer. If retransmissions were conducted at the network layer on top of 802.11b MAC layer retransmissions, performance will be significantly degraded as the send queue will be held up.

Performance problems were observed with TCP connections, as standard TCP is not wireless aware. TCP assumes that lost packets are caused by congestion, and will slow down unnecessarily when packets have been lost due to the wireless environment. This slow down occurs when packets are lost during a TCP connection, which can be caused by users temporarily moving out of range. The slow down is quite severe and very noticeable to the user. This problem is caused by TCP, not DSR and users can reopen the TCP connection to get around this problem. A proper solution would be to have a wireless aware TCP, which is an area of research that is beyond the scope of this thesis.

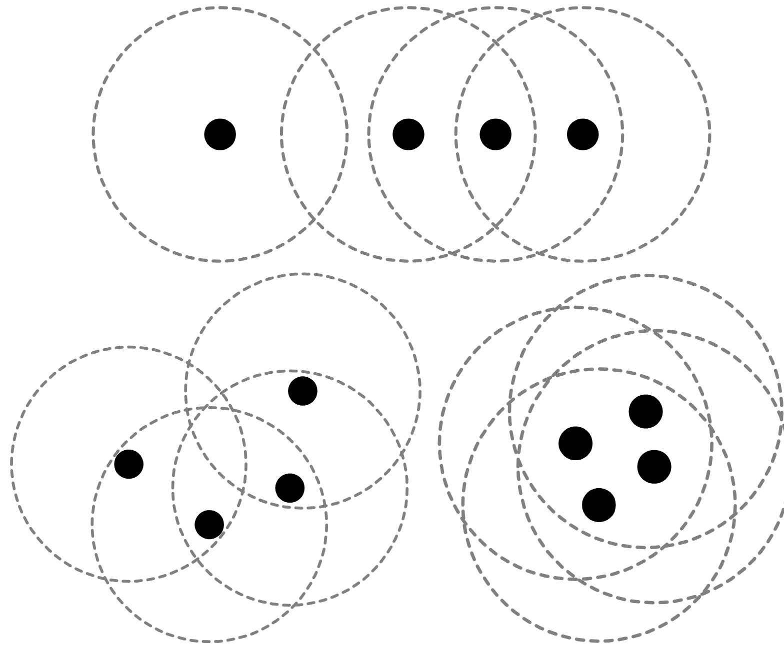

Figure 6.1.: Some tested network topologies

6.2.3. Multi-hop Ad Hoc Capabilities

A total of 4 nodes were available for testing, a PC acting as a gateway and 3 Compaq iPAQs. Only the iPAQs were completely mobile, but relative node mobility was still able to be achieved. Figure 6.1 shows some of the tested network topologies. For each of the tested topologies and the transitions between them, the DSR implementation was able to adapt and reconfigure itself to the network topology changes. The tests were conducted with the nodes moving at walking speeds.

Given that the smallest hop range of the 802.11b wireless interfaces was 25 m, difficulties were introduced to the testing process as a lot of time was wasted walking around placing the nodes. An improvement to the testing process was needed and the solution was to wrap a layer of aluminium foil around the antennas of the wireless cards. This reduced the hop range to around 2 m as metals are reflectors for RF signals, and a tightly wrapped aluminium foil around the antenna will greatly reduce the signal strength. Figure 6.2 shows an iPAQ with its wireless card wrapped in aluminium foil.

6.2.4. Gateway Capabilities

Gateway capabilities were shown to be functional as nodes inside the DSR network were able to access external non-DSR networks at the IP level. Ping, traceroute and other

Figure 6.2.: iPAQ with its wireless card wrapped in aluminium foil

TCP/IP applications were tested and they were able to access external networks. Due to the fact that the DSR network was using private IP addresses, network address translation (NAT) needed to be used with the DSR gateway to enable access to external networks,. The gateway was not tested on a DSR network with real IP addresses, as it involved significant changes to the departmental network, but it should be functional. The design philosophy was followed with the DSR gateway implementation, allowing the NAT module to work with the DSR gateway without modifications.

6.3. Personal Evaluation

Many experiences were gained during the design and development of the picoNet II system. The experiences were gained in many areas ranging from programming and debugging skills to problem solving and project management skills.

In the area of technical skills, many skills gained from my studies were put into practice and new techniques were developed as the project progressed. Working with the Linux kernel confirmed and extended my understanding of operating systems.

An important lesson in problem solving was learned during the development of the system. The lesson was that using source code to solve a problem should be the last resort and not the first. There was a tendency for me to go to the source code first whenever a technical problem arose, as the source code was freely available. In one situation, I came

across some documentation on the Internet that described the section of source code that I was looking at. A great deal of time would have been saved if the documentation was looked at first instead of the source code. That said, there were situations where the source code was clearer, as documentation can get out of date.

The practice of using a workbook to record design details was developed and it proved to be very a useful tool. Various other planning and time management skills were also developed.

Overall, I was satisfied with the outcome of the project as the final result was functional. The process of researching, designing and implementing a system has developed many of my skills and I am sure they will prove to be very useful in the future.

6.4. Summary

This chapter evaluated the system against the specifications outlined in chapter 3. The system met the specifications in a qualitative sense. Quantitatively, the overheads introduced by the DSR implementation was not optimal as many DSR protocol optimisations were not implemented. The multi-hop and gateway capabilities of the system was fully operational and compatible with TCP/IP. Personally, many skills were developed during the course of the thesis project, and they should prove to be useful in the future. Future developments for the thesis project will be discussed in the next chapter.

7. Future Developments

This chapter outlines some improvements to the picoNet II implementation and discusses possible future extensions.

7.1. Implementation Improvements

There are many improvements which can be made to the DSR protocol implementation. As mentioned in section 6.2.2, MAC layer acknowledgements can be implemented to reduce the packet data overhead.

The route cache is currently implemented as a simple table storing the full route to each destination. The route cache data structure can be improved to use a link state organisation where individual hops are stored in the cache. This can result in a more efficient use of the routing information, possibly reducing the number or route discoveries.

Currently, the DSR module uses brute force searching and this can be improved by implementing more efficient search techniques such as hashing. Other unimplemented optimisations specified in the DSR draft ([19]) can also be implemented to improve various performance aspects of the DSR module.

7.2. Routing Protocol Extensions

An extension to the DSR protocol called the “flow state” has been specified in a separate MANET draft standard ([22]). The DSR flow state extension allows the routing of most packets in a DSR network without adding source route information to each packet. The extension reduces the overhead of the DSR protocol without changing the fundamental operations of DSR.

Alternative routing protocols, such as AODV or TORA, may be used instead of DSR or in conjunction with DSR, as extensions.

7.3. Global Roaming

A mobile ad hoc network allows nodes to move around freely in a network without losing network connectivity. MANET technologies do not address the problems with node mobility on fixed networks such as the Internet. Mobile IP ([23]) is a technology that addresses this problem and allows nodes to roam within and between fixed networks. Mobile IP enables nodes to roam without changing the operations of the underlying fixed network.

Coupling MANET and Mobile IP technologies will enable nodes to roam anywhere where network connectivity exists. MANET technologies will be utilised for roaming within a subnet and when users roam to a different subnet, MANET technologies will work in conjunction with Mobile IP to provide transparent network connectivity. This “Global Roaming” ability can take us one step closer to true pervasive computing.

8. Conclusion

picoNet II was designed to provide multi-hop capabilities to existing ad hoc networks. The system was able to create a dynamic, self-organising, and self-configuring network on the fly without the aid of any networking infrastructure.

This thesis has demonstrated that it is possible to create extensions to existing technologies transparently, maintaining full compatibility. Although the implementation was not the most efficient, it was fully functional and can be used in many real world applications.

The current design presents a useful platform for future extensions, and may take us closer to realising the vision of pervasive computing, where technology blends seamlessly with everyday life.

A. Source Code Listings

A.1. Software License

Copyright (C) 2001 Alex Song

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

The author can be contacted via email at s369677@student.uq.edu.au

A.2. Makefile

```
#Makefile for the dsr kernel module
CCx86=gcc
CCarm=arm-linux-gcc

MODCFLAGS := -O3 -Wall -DCONFIG_KERNELD -DMODULE -D__KERNEL__ -DLINUX

HPATHx86 := /usr/src/linux/include
HPATHarm := /home/alex/linux/kernel/include

COMPILEARM := $(CCarm) $(MODCFLAGS) -I$(HPATHarm)
COMPILEX86 := $(CCx86) $(MODCFLAGS) -I$(HPATHx86)

LINKARM := arm-linux-ld -m armelf -r
LINKX86 := ld -m elf_i386 -r

COMMONOBJ:= dsr_route.o dsr_debug.o dsr_input.o dsr_output.o dsr_queue.o
LINKOBJ := dsr-kmodule.o $(COMMONOBJ)
LINKOBJDBG := dsr-kdbg.o $(COMMONOBJ)

SRC := $(wildcard *.c)
ARMOBJ := $(SRC:.c=-arm.o)
```

```
X86OBJ := $(SRC:.c=-x86.o)

TARGET := dsrarm.o dsrx86.o dsrarmdbg.o dsrx86dbg.o

all:            $(TARGET)

clean:
                rm -f *.o *.d

###arm build
%-arm.d:        %.c
                $(CCarm) -I$(HPATHarm) -MM $< 2> /dev/null \
                | sed "s/\.o/-arm\.o/" > $@

include $(ARMOBJ:.o=.d)

%-arm.o:        %.c
                $(COMPILEARM) -c $< -o $@

dsrarm.o:       $(LINKOBJ:.o=-arm.o)
                $(LINKARM) -o dsrarm.o $(LINKOBJ:.o=-arm.o)

dsrarmdbg.o:    $(LINKOBJDBG:.o=-arm.o)
                $(LINKARM) -o dsrarmdbg.o $(LINKOBJDBG:.o=-arm.o)

###x86 build
%-x86.d:        %.c
                $(CCx86) -I$(HPATHx86) -MM $< 2> /dev/null \
                | sed "s/\.o/-x86\.o/" > $@

include $(X86OBJ:.o=.d)

%-x86.o:        %.c
                $(COMPILEX86) -c $< -o $@

dsrx86.o:       $(LINKOBJ:.o=-x86.o)
                $(LINKX86) -o dsrx86.o $(LINKOBJ:.o=-x86.o)

dsrx86dbg.o:    $(LINKOBJDBG:.o=-x86.o)
                $(LINKX86) -o dsrx86dbg.o $(LINKOBJDBG:.o=-x86.o)
```

A.3. dsr.h

```
#ifndef DSR_H
#define DSR_H

/* option types */
#define PAD1        0
#define PADN        1
#define ROUTE_REQ   2
#define ROUTE_REPLY 3
```

```
#define ROUTE_ERROR 4
#define ACK_REQ     5
#define ACK         6
#define SRC_ROUTE   7

/* ietf draft definitions */
#define BCAST_JITTER         20  /* milliseconds default 10 */
#define MAX_ROUTE_LEN        15  /* nodes default 15 */
#define ROUTE_CACHE_TIMEOUT  300 /* seconds */

#define SEND_BUFFER_TIMEOUT  30  /* seconds */
#define REQ_TABLE_SIZE       12  /* nodes default 64 */
#define REQ_TABLE_IDS        16
#define MAX_REQ_RXMT         16
#define MAX_REQ_PERIOD       10  /* seconds default 10 */
#define REQ_PERIOD           250 /* milliseconds default 500 */

#define DSR_RXMT_BUFFER_SIZE 300 /* packets default 50 */
#define DSR_MAXRXTSHIFT      0   /* default twice */

/* own definitions */
#define ROUTE_CACHE_SIZE 20      /* entries */
#define SEND_BUFFER_SIZE 50      /* packets */
#define NO_NEXT_HEADER   59      /* ipv6 no next header */
#define DSR_PROTOCOL     168     /* protocol number */
#define ACK_TIMEOUT_MS   300     /* milliseconds 300-500 is good */
#define ACK_TIMEOUT_JF   ((ACK_TIMEOUT_MS * HZ) / 1000)

#define DSR_SUBNET(x)     ((x & NETMASK) == NETWORK)
#define NOT_DSR_SUBNET(x) ((x & NETMASK) != NETWORK)

#endif /* DSR_H */
```

A.4. dsr_header.h

```
#ifndef DSR_HEADER_H
#define DSR_HEADER_H

#include <asm/byteorder.h>
#include <linux/types.h>
#include <linux/timer.h>
#include <linux/skbuff.h>
#include "dsr.h"

struct dsr_hdr {
  __u8  nexthdr;
  __u8  reserved;
  __u16 length;
  /* options start here */
};
```

```
struct dsr_opt_hdr {
  __u8  type;
  __u8  len;
};

struct dsr_rt_req_opt {
  __u8  type;
  __u8  len;
  __u16 ident;
  __u32 taddr;
  __u32 addr[MAX_ROUTE_LEN];
};

struct dsr_rt_reply_opt {
  __u8  type;
  __u8  len;
#if defined(__LITTLE_ENDIAN_BITFIELD)
  __u8  reserved:7, lasthopx:1;
#elif defined (__BIG_ENDIAN_BITFIELD)
  __u8  lasthopx:1, reserved:7;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
  __u16 ident;
  __u32 addr[MAX_ROUTE_LEN];
} __attribute__ ((packed));

struct dsr_rt_err_opt {
  __u8  type;
  __u8  len;
  __u8  errtype;
#if defined(__LITTLE_ENDIAN_BITFIELD)
  __u8  salvage:4, reserved:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
  __u8  reserved:4, salvage:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
  __u32 errsaddr;
  __u32 errdaddr;
  __u32 typeinfo;  /* need to change for future route errors */
};

struct dsr_ack_req_opt {
  __u8  type;
  __u8  len;
  __u16 ident;
  __u32 saddr;
};

struct dsr_ack_opt {
  __u8  type;
  __u8  len;
  __u16 ident;
```

```
  __u32 saddr;
  __u32 daddr;
};

struct dsr_src_rt_opt {
  __u8  type;
  __u8  len;
#if defined(__LITTLE_ENDIAN_BITFIELD)
  __u16 segs_left:6, salvage:4, reserved:4, lasthopx:1, firsthopx:1;
#elif defined (__BIG_ENDIAN_BITFIELD)
  __u16 firsthopx:1, lasthopx:1, reserved:4, salvage:4, segs_left:6;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
  __u32 addr[MAX_ROUTE_LEN];
};

struct dsr_pad1_opt {
  __u8 type;
};

struct dsr_padn_opt {
  __u8  type;
  __u8  len;
  /* zero filled data */
};

/* route cache structures */
struct rt_entry {
  time_t        time;   /* in seconds */
  unsigned char segs_left;
  __u32         addr[MAX_ROUTE_LEN];
  __u32         dst;
};

struct id_fifo {
  __u32         saddr;
  time_t        time;   /* in seconds */
  __u16         id[REQ_TABLE_IDS];
  int           head;
  int           tail;
  int           len;
};

struct rt_req_entry {
  __u32             taddr;
  unsigned long     time;    /* in jiffies */
  unsigned long     timeout;
  int               n_req;
  /* timer_list is 20 bytes */
  struct timer_list timer;
};

/* the following structs has to be smaller than 48 bytes */
```

```
struct rxmt_info {
  __u16                 ident;
  unsigned int           n_rxmt;
  struct dsr_src_rt_opt * srcrt;
  /* timer_list is 20 bytes */
  struct timer_list       timer;
};

struct jitter_send {
  /* timer_list is 20 bytes */
  struct timer_list timer;
};

struct send_info {
  __u32 fwdaddr;
  int (*output)(struct sk_buff *);
  /* timer_list is 20 bytes */
  struct timer_list timer;
};
#endif /* DSR_HEADER_H */
```

A.5. dsr-kmodule.h

```
#ifndef DSR_KMODULE_H
#define DSR_KMODULE_H

#include <linux/kernel.h>
#include <linux/module.h>

#ifdef __KERNEL__

#include <linux/netfilter_ipv4.h>
#include <linux/netdevice.h>
#include <linux/proc_fs.h>
#include <linux/ip.h>
#include <linux/inet.h>
#include <net/checksum.h>
#include <net/ip.h>

#include "dsr_debug.h"
#include "dsr_header.h"
#include "dsr_route.h"
#include "dsr_input.h"
#include "dsr_output.h"
#include "dsr_queue.h"

nf_hookfn pre_route_handler;
nf_hookfn local_out_handler;
nf_hookfn gw_post_route_handler;
nf_hookfn gw_post_route_out_handler;
nf_hookfn post_route_handler;
```

```
nf_hookfn local_in_handler;
int dsr_read_proc(char * page, char ** start, off_t off, int count,
      int * eof, void * data);

#endif /* _KERNEL_ */
#endif /* DSR_KMODULE_H */
```

A.6. dsr-kmodule.c

```
#include "dsr-kmodule.h"

MODULE_AUTHOR("Alex Song <s369677@student.uq.edu.au>");
MODULE_DESCRIPTION("Dynamic Source Routing Kernel Module");
static char * ifname = "eth0";
MODULE_PARM(ifname, "s");
MODULE_PARM_DESC(ifname, "Network interface name. default is eth0.");
static int gw = 0;
MODULE_PARM(gw, "i");
MODULE_PARM_DESC(gw, "Gateway mode 0 is off 1 is on. default is off.");

/* variables */
static __u32 IPADDR;
static __u32 NETMASK;
static __u32 BCAST;
static __u32 NETWORK;
static int hhl;
static struct rt_entry rt_cache[ROUTE_CACHE_SIZE];
static int rt_cache_size = 0;
static struct id_fifo rt_req_ids[REQ_TABLE_SIZE];
static int rt_req_ids_size = 0;
static struct rt_req_entry rt_req_table[REQ_TABLE_SIZE];
static int rt_req_table_size = 0;
static __u16 ack_id;
static __u16 rtreq_id;
static struct sk_buff_head rxmt_q;
static struct sk_buff_head send_q;
static struct sk_buff_head jitter_q;

/* statistic variables */
unsigned int stat_send_q_timeout = 0;
unsigned int stat_send_q_drop = 0;
unsigned int stat_ack_q_timeout = 0;
unsigned int stat_ack_q_drop = 0;
unsigned int stat_ack_q_resend = 0;
unsigned int stat_send_rt_err = 0;
unsigned int stat_send_rt_req = 0;
unsigned int stat_send_rt_reply = 0;
unsigned int stat_forward_pkts = 0;
unsigned int stat_add_f_rt = 0;
unsigned int stat_add_r_rt = 0;
unsigned int stat_remove_rt = 0;
```

```
unsigned int stat_dsr_send = 0;
unsigned int stat_dsr_frag = 0;
unsigned int stat_output_err = 0;

// struct needed to hook at pre route
static struct nf_hook_ops pre_route = {
  {NULL, NULL},
  pre_route_handler,
  PF_INET,
  NF_IP_PRE_ROUTING,
  NF_IP_PRI_FIRST,
};
// struct needed to hook at local out
static struct nf_hook_ops local_out = {
  {NULL, NULL},
  local_out_handler,
  PF_INET,
  NF_IP_LOCAL_OUT,
  NF_IP_PRI_MANGLE,
};
// struct needed to hook at post route for gateway
static struct nf_hook_ops gw_post_route = {
  {NULL, NULL},
  gw_post_route_handler,
  PF_INET,
  NF_IP_POST_ROUTING,
  NF_IP_PRI_NAT_SRC + 1,
};
// struct needed to hook at post route for gateway
static struct nf_hook_ops gw_post_route_out = {
  {NULL, NULL},
  gw_post_route_out_handler,
  PF_INET,
  NF_IP_POST_ROUTING,
  NF_IP_PRI_FIRST,
};
#ifdef DSR_DEBUG
// struct needed to hook at post route
static struct nf_hook_ops post_route = {
  {NULL, NULL},
  post_route_handler,
  PF_INET,
  NF_IP_POST_ROUTING,
  NF_IP_PRI_FIRST,
};
// struct needed to hook at local in
static struct nf_hook_ops local_in = {
  {NULL, NULL},
  local_in_handler,
  PF_INET,
  NF_IP_LOCAL_IN,
  NF_IP_PRI_FIRST,
};
#endif
```

```
/* initialise the module */
int init_module()
{
  int i;
  struct proc_dir_entry * proc_file;
  struct net_device * netdev = dev_get_by_name(ifname);
  struct in_device * indev = in_dev_get(netdev);

  /* short circuit if statement */
  if (netdev != NULL && indev != NULL && indev->ifa_list != NULL) {
#ifdef DSR_DEBUG
    printk("local:%u.%u.%u.%u\n", NIPQUAD(indev->ifa_list->ifa_local));
    printk("dst:%u.%u.%u.%u\n", NIPQUAD(indev->ifa_list->ifa_address));
    printk("mask:%u.%u.%u.%u\n", NIPQUAD(indev->ifa_list->ifa_mask));
    printk("bcast:%u.%u.%u.%u\n", NIPQUAD(indev->ifa_list->ifa_broadcast));
#endif
    IPADDR = indev->ifa_list->ifa_local;
    NETMASK = indev->ifa_list->ifa_mask;
    BCAST = indev->ifa_list->ifa_broadcast;
  } else {
    return -1;
  }

  hhl = netdev->hard_header_len;
  in_dev_put(indev);
  dev_put(netdev);

  if ((sizeof(struct rxmt_info) >= 48) ||
      (sizeof(struct jitter_send) >= 48) ||
      (sizeof(struct send_info) >= 48)) {
    printk("struct size error %i %i %i\n",
           sizeof(struct rxmt_info),
           sizeof(struct jitter_send),
           sizeof(struct send_info));
    return -1;
  }

  /* calculate network address */
  NETWORK = IPADDR & NETMASK;

  /* init our route cache */
  for (i = 0;i < ROUTE_CACHE_SIZE;i++)
    rt_cache[i].dst = 0;

  /* init our rt req id table */
  for (i = 0;i < REQ_TABLE_SIZE;i++)
    rt_req_ids[i].saddr = 0;

  /* init our rt req table */
  for (i = 0;i < REQ_TABLE_SIZE;i++)
    rt_req_table[i].taddr = 0;

  /* nf_register_hook returns int but seems to be always 0 */
```

```
  nf_register_hook(&pre_route);
  nf_register_hook(&local_out);
  if (gw == 1) {
    nf_register_hook(&gw_post_route_out);
    nf_register_hook(&gw_post_route);
  }

#ifdef DSR_DEBUG
  nf_register_hook(&post_route);
  nf_register_hook(&local_in);
#endif

  //printk("creating proc read entry\n");
  proc_file = create_proc_read_entry("dsr", 0/*mode_t*/, &proc_root,
                                     dsr_read_proc, NULL/*data*/);

  ack_id = net_random();
  rtreq_id = net_random();
  skb_queue_head_init(&rxmt_q);
  skb_queue_head_init(&send_q);
  skb_queue_head_init(&jitter_q);

  //printk("success\n");
  printk(KERN_INFO "dsr:Dynamic Source Routing activated on %s.\n", ifname);
  if (gw == 1)
    printk(KERN_INFO "dsr:Running as a gateway.\n");
  else
    printk(KERN_INFO "dsr:Running as a node\n");
  printk(KERN_INFO "dsr:(c) Alex Song <s369677@student.uq.edu.au>\n");
  return 0; // 0 means module succesfully loaded
}

/* cleanup module unregister everything */
void cleanup_module() {
  int i;
  struct sk_buff * skbptr;

  /* nf_unregister_hook returns void */
  nf_unregister_hook(&pre_route);
  nf_unregister_hook(&local_out);
  if (gw == 1) {
    nf_unregister_hook(&gw_post_route_out);
    nf_unregister_hook(&gw_post_route);
  }

#ifdef DSR_DEBUG
  nf_unregister_hook(&post_route);
  nf_unregister_hook(&local_in);
#endif

  /* clean retransmit queue */
  skbptr = skb_peek(&rxmt_q);
  for (i = 0; i < skb_queue_len(&rxmt_q);i++) {
    struct rxmt_info * rxmtinfo;
```

```
    DSR_ASSERT(skbptr != NULL);
    rxmtinfo = (struct rxmt_info *) skbptr->cb;
    del_timer(&(rxmtinfo->timer));
    skbptr = skbptr->next;
  }
  skb_queue_purge(&rxmt_q);

  /* clean route request timers */
  for (i = 0;i < REQ_TABLE_SIZE;i++) {
    if (rt_req_table[i].taddr != 0) {
      del_timer(&(rt_req_table[i].timer));
    }
  }

  /* clean send queue */
  skbptr = skb_peek(&send_q);
  for (i = 0; i < skb_queue_len(&send_q);i++) {
    struct send_info * sendinfo;
    DSR_ASSERT(skbptr != NULL);
    sendinfo = (struct send_info *) skbptr->cb;
    del_timer(&(sendinfo->timer));
    skbptr = skbptr->next;
  }
  skb_queue_purge(&send_q);

  /* clean jitter send queue */
  skbptr = skb_peek(&jitter_q);
  for (i = 0; i < skb_queue_len(&jitter_q);i++) {
    struct jitter_send * jittersend;
    DSR_ASSERT(skbptr != NULL);
    jittersend = (struct jitter_send *) skbptr->cb;
    del_timer(&(jittersend->timer));
    skbptr = skbptr->next;
  }
  skb_queue_purge(&jitter_q);

  //printk("remove proc entry\n");
  remove_proc_entry("dsr", &proc_root);
  printk(KERN_INFO "dsr:Dynamic Source Routing deactivated on %s.\n", ifname);
}

#ifdef DSR_DEBUG
unsigned int local_in_handler(unsigned int hooknum,
                              struct sk_buff **skb,
                              const struct net_device *in,
                              const struct net_device *out,
                              int (*okfn)(struct sk_buff *)) {

  struct iphdr *iph = (*skb)->nh.iph;

  if (DSR_SUBNET(iph->saddr)
      /*&& (iph->protocol == DSR_PROTOCOL || iph->protocol == 1)*/) {
    printk("\nlocal in enter\n");
```

```
    dump_skb_ptr(*skb);
    dump_skb(PF_INET, *skb);

    printk("local in leave\n");
  }
  return NF_ACCEPT;
}

unsigned int post_route_handler(unsigned int hooknum,
                                struct sk_buff **skb,
                                const struct net_device *in,
                                const struct net_device *out,
                                int (*okfn)(struct sk_buff *)) {
  struct iphdr *iph = (*skb)->nh.iph;

  if (DSR_SUBNET(iph->saddr)
      /*&& (iph->protocol == DSR_PROTOCOL || iph->protocol == 1)*/) {
    printk("\npost route enter\n");

    dump_skb_ptr(*skb);
    dump_skb(PF_INET, *skb);

    printk("post route leave\n");
  }
  return NF_ACCEPT;
}
#endif

unsigned int local_out_handler(unsigned int hooknum,
                               struct sk_buff **skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn)(struct sk_buff *)) {

  struct iphdr *iph = (*skb)->nh.iph;

  if (iph->daddr == IPADDR || NOT_DSR_SUBNET(iph->saddr)
      || BADCLASS(iph->daddr) || ZERONET(iph->daddr)
      || LOOPBACK(iph->daddr) || MULTICAST(iph->daddr) ) {

    return NF_ACCEPT;

  } else {
    /* if we are here, src must be on dsr network */
    struct rtable * rt = (struct rtable *) (*skb)->dst;
    struct rt_entry * rt_ptr;
    __u32 fwdaddr;

#ifdef DSR_DEBUG
    printk("\nlocal out enter\n");
    dump_skb_ptr(*skb);
    dump_skb(PF_INET, *skb);
#endif
```

```
    (*skb)->nfcache |= NFC_UNKNOWN;

    /* check if we are sending to dsr network or external */
    if (DSR_SUBNET(iph->daddr)) {
      /* we are sending to the dsr network */
      fwdaddr = iph->daddr;
    } else {
      /* we are sending to external network */
      fwdaddr = rt->rt_gateway;
    }

    rt_ptr = lookup_route(rt_cache, fwdaddr);
    if (rt_ptr == NULL) {
#ifdef DSR_DEBUG
      printk("send route discovery\n");
#endif
      send_rt_req(fwdaddr);
      send_q_add(&send_q, *skb, SEND_BUFFER_SIZE, fwdaddr, okfn);
      //printk("leave\n");
      //kfree_skb(*skb);
    } else {
      dsr_send(skb, rt_ptr, okfn);
    }
#ifdef DSR_DEBUG
    printk("local out leave\n");
#endif
    return NF_STOLEN;
  }
}

unsigned int pre_route_handler(unsigned int hooknum,
                               struct sk_buff **skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn)(struct sk_buff *)) {

  struct iphdr *iph = (*skb)->nh.iph;

  if (iph->protocol == DSR_PROTOCOL) {

    const struct dsr_hdr * dsrhdr = (struct dsr_hdr *) ((*skb)->nh.raw +
                                                        (iph->ihl)*4);
    const unsigned char * optstart = (unsigned char *) (dsrhdr + 1);
    unsigned char * optptr = (unsigned char *) (dsrhdr + 1);
    struct dsr_opt_hdr * dsropt;
    int ackreq = 0;

#ifdef DSR_DEBUG
    printk("\npre route enter\n");
    dump_skb_ptr(*skb);
    dump_skb(PF_INET, *skb);
#endif

    (*skb)->nfcache |= NFC_UNKNOWN;
```

```
(*skb)->nfcache |= NFC_ALTERED;

while (optptr - optstart != ntohs(dsrhdr->length)) {

  dsropt = (struct dsr_opt_hdr *) optptr;
  //printk("i:%u len:%u\n", optptr - optstart, ntohs(dsrhdr->length));
  switch (dsropt->type) {
  case PAD1:
    //printk("got pad\n");
    break;
  case PADN:
    //printk("got padn\n");
    break;
  case ROUTE_REQ:
    //printk("got rt req\n");
    proc_rt_req_opt(*skb, optptr);
    break;
  case ROUTE_REPLY:
    //printk("got rt reply\n");
    proc_rt_reply_opt(*skb, optptr);
    break;
  case ROUTE_ERROR:
    //printk("got rt err\n");
    proc_rt_err_opt(*skb, optptr);
    break;
  case ACK_REQ:
    //printk("got ack req\n");
    proc_ack_req_opt(*skb, optptr/*, in*/);
    ackreq = 1;
    break;
  case ACK:
    //printk("got ack\n");
    proc_ack_reply_opt(*skb, optptr);
    break;
  case SRC_ROUTE:
    //printk("got src rt opt\n");
    //proc_src_rt_opt(*skb, optptr);
    if (proc_src_rt_opt(*skb, optptr) != 0) {
      return NF_DROP;
    }
    break;
  default:
    printk("unknown dsr option %i\n", dsropt->type);
    dump_skb_ptr(*skb);
    dump_skb(PF_INET, *skb);
    return NF_DROP;
    break;
  }

  if (dsropt->type == PAD1)
    optptr++;
  else
    optptr += dsropt->len + 2;
}
```

```
    if (iph->daddr == IPADDR || iph->daddr == BCAST) {
      //printk("for us\n");
      if ((iph->ihl)*4 + ntohs(dsrhdr->length) + 4 >= ntohs(iph->tot_len)) {
        //printk("no payload, drop\n");
        return NF_DROP;
      }
      rmv_dsr_hdr(*skb);
      //okfn(*skb);
      //return NF_STOLEN;
    } else if (gw == 1 && NOT_DSR_SUBNET(iph->daddr)) {
      stat_forward_pkts++;
      rmv_dsr_hdr(*skb);
    } else if (iph->ttl == 1) {
      //printk("ttl expired\n");
      /* if ttl is about to expire and we are supposed to forward */
      rmv_dsr_hdr(*skb);
    } else {
      //printk("forward packet\n");
      if (ackreq == 1)
        ack_q_add(&rxmt_q, *skb, DSR_RXMT_BUFFER_SIZE);
      stat_forward_pkts++;
      //okfn(*skb);
      //return NF_STOLEN;
    }
#ifdef DSR_DEBUG
    printk("pre route leave\n");
#endif
    return NF_ACCEPT;
  }
  return NF_ACCEPT;
}

unsigned int gw_post_route_handler(unsigned int hooknum,
                                   struct sk_buff **skb,
                                   const struct net_device *in,
                                   const struct net_device *out,
                                   int (*okfn)(struct sk_buff *)) {

  struct iphdr *iph = (*skb)->nh.iph;
  DSR_ASSERT(gw == 1);

  if (iph->protocol != DSR_PROTOCOL && NOT_DSR_SUBNET(iph->saddr) &&
      DSR_SUBNET(iph->daddr)) {
    struct rt_entry * rt_ptr;

    //printk("gw post route:%u.%u.%u.%u d:%u.%u.%u.%u\n", NIPQUAD(iph->saddr),
    //       NIPQUAD(iph->daddr));
#ifdef DSR_DEBUG
    dump_skb_ptr(*skb);
    dump_skb(PF_INET, *skb);
#endif

    /* packet from outside dsr net and we are a gateway */
```

```
    rt_ptr = lookup_route(rt_cache, iph->daddr);
    if (rt_ptr == NULL) {
#ifdef DSR_DEBUG
      printk("gw send route discovery\n");
#endif
      send_rt_req(iph->daddr);
      send_q_add(&send_q, *skb, SEND_BUFFER_SIZE, iph->daddr, okfn);
    } else {
      //printk("gw have route\n");
      dsr_send(skb, rt_ptr, okfn);
    }
    return NF_STOLEN;
  }
  return NF_ACCEPT;
}

unsigned int gw_post_route_out_handler(unsigned int hooknum,
                                       struct sk_buff **skb,
                                       const struct net_device *in,
                                       const struct net_device *out,
                                       int (*okfn)(struct sk_buff *)) {

  struct iphdr *iph = (*skb)->nh.iph;

  DSR_ASSERT(gw == 1);
  if (iph->protocol == DSR_PROTOCOL) {
    okfn(*skb);
    return NF_STOLEN;
  }
  return NF_ACCEPT;
}

/* this function is copied from the linux source code fs/proc/proc_misc.c */
int proc_calc_metrics(char *page, char **start, off_t off,
                      int count, int *eof, int len) {
  if (len <= off+count) *eof = 1;
  *start = page + off;
  len -= off;
  if (len>count) len = count;
  if (len<0) len = 0;
  return len;
}

int dsr_read_proc(char *page, char **start, off_t off, int count, int *eof,
                  void *data) {

  int len = 0;
#ifdef DSR_DEBUG
  int i, j;
#endif

  MOD_INC_USE_COUNT;
  /* stats */
  len += sprintf(page + len, "send_q_drop:%u send_q_timeout:%u ack_q_drop:%u "
```

```
                "ack_q_timeout:%u\n", stat_send_q_drop, stat_send_q_timeout,
                stat_ack_q_drop, stat_ack_q_timeout);

  len += sprintf(page + len, "send_rt_err:%u send_rt_reply:%u "
                "add_rt f:%u r:%u send_rt_req:%u\n",
                stat_send_rt_err, stat_send_rt_reply, stat_add_f_rt,
                stat_add_r_rt, stat_send_rt_req);

  len += sprintf(page + len, "forward pkts:%u ack_q_resend:%u remove_rt:%u "
                "output_err:%u\n", stat_forward_pkts, stat_ack_q_resend,
                stat_remove_rt, stat_output_err);

  len += sprintf(page + len, "dsr_send:%u dsr_frag:%u\n",
                stat_dsr_send, stat_dsr_frag);

  len += sprintf(page + len, "send_q len:%i\n", skb_queue_len(&send_q));
  len += sprintf(page + len, "rxmt_q len:%i\n", skb_queue_len(&rxmt_q));
  /* dump rt req ids */
  len += sprintf(page + len, "rt_req_ids len:%i\n", rt_req_ids_size);
#ifdef DSR_DEBUG
  for (i = 0;i < REQ_TABLE_SIZE;i++) {
    if (rt_req_ids[i].saddr != 0) {
      len += sprintf(page + len, "saddr:%u.%u.%u.%u len:%i\n",
                    NIPQUAD(rt_req_ids[i].saddr), rt_req_ids[i].len);
      for (j = 0;j < rt_req_ids[i].len;j++) {
        len += sprintf(page + len, "%i ",
                      rt_req_ids[i].id[(rt_req_ids[i].head + j)
                                      % REQ_TABLE_IDS]);
      }
      len += sprintf(page + len, "\n");
    }
  }
#endif
  /* dump rt req table */
  len += sprintf(page + len, "rt_req_table len:%i\n", rt_req_table_size);
#ifdef DSR_DEBUG
  for (i = 0;i < REQ_TABLE_SIZE;i++) {
    if (rt_req_table[i].taddr != 0) {
      len += sprintf(page + len, "taddr:%u.%u.%u.%u n_req:%i ",
                    NIPQUAD(rt_req_table[i].taddr), rt_req_table[i].n_req);
      len += sprintf(page + len, "timeout:%i",
                    (int)(rt_req_table[i].timeout *1000)/HZ);
      len += sprintf(page + len, "\n");
    }
  }
#endif
  /* dump route cache */
  len += sprintf(page + len, "rt_cache len:%i\n", rt_cache_size);
#ifdef DSR_DEBUG
  for (i = 0;i < ROUTE_CACHE_SIZE;i++) {
    if (rt_cache[i].dst != 0) {

      len += sprintf(page + len, "%i %u.%u.%u.%u age:%u segs:%u\n",
                    i, NIPQUAD(rt_cache[i].dst),
```

```
                    (unsigned int) (CURRENT_TIME - rt_cache[i].time),
                    rt_cache[i].segs_left);
      for (j = 0;j < rt_cache[i].segs_left;j++) {
        len += sprintf(page + len, " %u.%u.%u.%u",
                       NIPQUAD(rt_cache[i].addr[j]));
      }
      len += sprintf(page + len, "\n");
    }
  }
#endif
  MOD_DEC_USE_COUNT;
  DSR_ASSERT(len <= count);
  return proc_calc_metrics(page, start, off, count, eof, len);
}
```

A.7. dsr_debug.h

```
#ifndef DSR_DEBUG_H
#define DSR_DEBUG_H

#include <linux/kernel.h>
#include <linux/module.h>

#ifdef __KERNEL__

/* dump_skb includes */
#include <net/ip.h>
#include <net/route.h>
#include <net/tcp.h>
#include <linux/netfilter_ipv4.h>

#define DSR_ASSERT(x)             \
do {                    \
  if (!(x)) {             \
    printk("DSR_ASSERT: %s:%s:%u\n",      \
           __FUNCTION__, __FILE__, __LINE__);  \
        BUG();}                                  \
} while(0)


//#define DSR_DEBUG

void dump_skb(int pf, struct sk_buff *skb);
void dump_skb_ptr(struct sk_buff *skb);

#endif /* _KERNEL_ */
#endif /* DSR_DEBUG_H */
```

A.8. dsr_debug.c

```
#include "dsr_debug.h"

void dump_skb(int pf, struct sk_buff *skb)
{

  const struct iphdr *ip = skb->nh.iph;
  __u32 *opt = (__u32 *) (ip + 1);
  int opti;
  __u16 src_port = 0, dst_port = 0;
  int icmptype = -1;

  if (ip->protocol == IPPROTO_ICMP)
    icmptype = skb->nh.raw[20];

  printk("dsrskb: %s len=%u shared=%u cloned=%u PROTO=%d:%i\n",
   skb->sk ? "(owned)" : "(unowned)",
   skb->len, skb_shared(skb), skb_cloned(skb), ip->protocol, icmptype);

  if (ip->protocol == IPPROTO_TCP
      || ip->protocol == IPPROTO_UDP) {
    struct tcphdr *tcp=(struct tcphdr *)((__u32 *)ip+ip->ihl);
    src_port = ntohs(tcp->source);
    dst_port = ntohs(tcp->dest);
  }

  printk("s:%u.%u.%u.%u:%hu d:%u.%u.%u.%u:%hu"
   " L=%hu T=%hu S=0x%2.2hX I=%hu F=0x%4.4hX\n",
   NIPQUAD(ip->saddr),
   src_port, NIPQUAD(ip->daddr),
   dst_port,
   ntohs(ip->tot_len), ip->ttl, ip->tos, ntohs(ip->id),
   ntohs(ip->frag_off));

  /* dump some stuff after the ip header */
  for (opti = 0; opti < 6; opti++)
    printk("O=%8.8X ", *opt++);
  printk("\n");
}

void dump_skb_ptr(struct sk_buff *skb) {

  const struct rtable * rt = (struct rtable *) skb->dst;

  if (rt != NULL) {
    printk("route dump: rt_src %u.%u.%u.%u rt_dst %u.%u.%u.%u"
      " rt_gw %u.%u.%u.%u\n",
     NIPQUAD(rt->rt_src), NIPQUAD(rt->rt_dst), NIPQUAD(rt->rt_gateway) );
  } else {
    printk("route dump: no route info\n");
  }
}
```

A.9. dsr_input.h

```
#ifndef DSR_INPUT_H
#define DSR_INPUT_H

#include <linux/skbuff.h>

void rmv_dsr_hdr(struct sk_buff * skb);
int proc_src_rt_opt(struct sk_buff *skb, unsigned char * optptr);
void proc_rt_req_opt(struct sk_buff *skb, unsigned char * optptr);
void proc_rt_reply_opt(struct sk_buff *skb, unsigned char * optptr);
void proc_rt_err_opt(struct sk_buff *skb, unsigned char * optptr);
void proc_ack_req_opt(struct sk_buff *skb, unsigned char * optptr);
void proc_ack_reply_opt(struct sk_buff *skb, unsigned char * optptr);

#endif /* DSR_INPUT_H */
```

A.10. dsr_input.c

```
#include "dsr_input.h"
#include "dsr_header.h"
#include "dsr_debug.h"
#include "dsr_route.h"
#include "dsr_output.h"

extern __u32 IPADDR;
extern __u32 NETMASK;
extern __u32 NETWORK;
extern int   gw;
extern __u16 ack_id;
extern struct sk_buff_head rxmt_q;
extern struct rt_entry rt_cache[];
extern struct id_fifo rt_req_ids[];
extern int rt_req_ids_size;

int in_rt_req(unsigned char * optptr) {

  struct dsr_rt_req_opt * rtreq = (struct dsr_rt_req_opt *) optptr;
  int i;

  for (i = 0;i < ((rtreq->len - 6) / 4);i++)
    if (rtreq->addr[i] == IPADDR) {
      //printk("in rt req\n");
      return 1;
    }
  //printk("not in rt req\n");
  return 0;
}

int lookup_rt_req_id(__u32 saddr, __u16 id) {
```

```
  int i, j, count;

  //printk("lookup id %i size %i\n", id, rt_req_ids_size);

  for (i = 0, count = 0; count < rt_req_ids_size;i++) {
    if (rt_req_ids[i].saddr != 0) {
      count++;
      if (rt_req_ids[i].saddr == saddr) {
  for (j = 0;j < rt_req_ids[i].len;j++) {
    if (rt_req_ids[i].id[(rt_req_ids[i].head + j) % REQ_TABLE_IDS] == id)
      return 1;
  }
  //printk("not found\n");
  return 0;
      }
    }
  }
  //printk("not found\n");
  return 0;
}

void add_rt_req_id(__u32 saddr, __u16 id) {

  int i, count;

  for (i = 0, count = 0;count < rt_req_ids_size;i++) {
    if (rt_req_ids[i].saddr != 0) {
      count++;
      if (rt_req_ids[i].saddr == saddr) {
  //printk("add rt req id found saddr\n");
  rt_req_ids[i].id[rt_req_ids[i].tail] = id;
  rt_req_ids[i].tail = (rt_req_ids[i].tail + 1) % REQ_TABLE_IDS;
  if (rt_req_ids[i].len < REQ_TABLE_IDS) {
    rt_req_ids[i].len++;
  } else {
    rt_req_ids[i].head = (rt_req_ids[i].head + 1) % REQ_TABLE_IDS;
  }
  return;
      }
    }
  }

  /* have to crete a new entry or replace an entry */
  if (rt_req_ids_size < REQ_TABLE_SIZE) {
    for (i = 0;i < REQ_TABLE_SIZE;i++) {
      if (rt_req_ids[i].saddr == 0) {
  rt_req_ids[i].saddr = saddr;
  rt_req_ids[i].time = CURRENT_TIME;
  rt_req_ids[i].id[0] = id;
  rt_req_ids[i].head = 0;
  rt_req_ids[i].tail = 1;
  rt_req_ids[i].len = 1;
  rt_req_ids_size++;
  //printk("add rt req id found empty entry\n");
```

```
  return;
      }
    }
    DSR_ASSERT(0);
  } else {
    int oldest = 0;

    for (i = 1;i < REQ_TABLE_SIZE;i++) {
      if (rt_req_ids[i].time < rt_req_ids[oldest].time) {
  oldest = i;
      }
    }
    rt_req_ids[oldest].saddr = saddr;
    rt_req_ids[oldest].time = CURRENT_TIME;
    rt_req_ids[oldest].id[0] = id;
    rt_req_ids[oldest].head = 0;
    rt_req_ids[oldest].tail = 1;
    rt_req_ids[oldest].len = 1;
    //printk("add rt req id found oldest entry\n");
    return;
  }
}

void rmv_dsr_hdr(struct sk_buff * skb) {

  struct iphdr * iph = skb->nh.iph;
  struct dsr_hdr * dsrhdr = (struct dsr_hdr *) (skb->nh.raw + (iph->ihl)*4);
  int size = ntohs(dsrhdr->length) + 4;

  iph->protocol = dsrhdr->nexthdr;

  skb->nh.raw = skb_pull(skb, size);
  /* iph will point to the "old" header at this moment */
  memmove(skb->nh.raw, skb->nh.raw - size, (iph->ihl)*4);

  /* set iph to the "new" header */
  iph = skb->nh.iph;
  iph->tot_len = htons(skb->len);

  /* recalculate check sum */
  ip_send_check(iph);
}

void proc_rt_req_opt(struct sk_buff * skb, unsigned char * optptr) {

  struct iphdr * iph = skb->nh.iph;
  struct dsr_rt_req_opt * rtreq = (struct dsr_rt_req_opt *) optptr;

  if (iph->saddr == IPADDR)
    return;

  add_reverse_route(iph->saddr, rtreq->addr, (rtreq->len - 6) / 4);

  if (rtreq->taddr == IPADDR) {
```

```
#ifdef DSR_DEBUG
    printk("send rt reply\n");
#endif
    send_rt_reply(skb, rtreq);
  } else {
    if (in_rt_req(optptr) == 0 &&
  lookup_rt_req_id(iph->saddr, rtreq->ident) == 0) {

#ifdef DSR_DEBUG
      printk("proc rt req rebroadcasting\n");
#endif
      add_rt_req_id(iph->saddr, rtreq->ident);
      rebcast_rt_req(skb, optptr);
    }
  }
}

void proc_rt_reply_opt(struct sk_buff * skb, unsigned char * optptr) {

  const struct iphdr * iph = skb->nh.iph;
  struct dsr_rt_reply_opt * rtreply = (struct dsr_rt_reply_opt *) optptr;

  if (iph->daddr == IPADDR) {
    add_forward_route(iph->saddr, rtreply->addr, (rtreply->len - 3) / 4);
  }
}

void proc_rt_err_opt(struct sk_buff *skb, unsigned char * optptr) {
  struct dsr_rt_err_opt * rterr = (struct dsr_rt_err_opt *) optptr;
  remove_route(rt_cache, rterr->errsaddr, rterr->typeinfo);
}

void proc_ack_req_opt(struct sk_buff * skb, unsigned char * optptr/*,
          const struct net_device * dev*/) {

  struct iphdr * iph;
  struct dsr_ack_req_opt * ackreq = (struct dsr_ack_req_opt *) optptr;

  send_ack_reply(skb, optptr);

  /*
   * if we are going to forward this packet then update the ack request
   */
  iph = skb->nh.iph;
  if (iph->daddr != IPADDR && (gw == 0 || DSR_SUBNET(iph->daddr))) {
    ack_id++;
    ackreq->ident = ack_id;
    ackreq->saddr = IPADDR;
  }
}

void proc_ack_reply_opt(struct sk_buff *skb, unsigned char * optptr) {

  const struct dsr_ack_req_opt * ackopt = (struct dsr_ack_req_opt *) optptr;
```

```
  int q_len = skb_queue_len(&rxmt_q);
  struct sk_buff * skbptr; // = skb_dequeue(&rxmt_q);
  struct rxmt_info * rxmtinfo; // = (struct rxmt_info *) skbptr->cb;
  int finished = 0;
  int i;

  /* optimisation: add neighbour node to route */
  /* forward and backward should be the same since route length is zero */
  add_forward_route(skb->nh.iph->saddr, NULL, 0);

  if (q_len == 0) {
    return;
  }

  /* find and remove the packet from the ack queue */
  for (i = 0;i < q_len && finished == 0;i++) {
    skbptr = skb_dequeue(&rxmt_q);;
    rxmtinfo = (struct rxmt_info *) skbptr->cb;
    DSR_ASSERT(skbptr != NULL);
    if (ackopt->ident == rxmtinfo->ident) {
      finished = 1;
      del_timer(&(rxmtinfo->timer));
      skb_unlink(skbptr);
      kfree_skb(skbptr);
    } else {
      skb_queue_tail(&rxmt_q, skbptr);
    }
  }
}

int proc_src_rt_opt(struct sk_buff * skb, unsigned char * optptr) {

  const struct iphdr * iph = skb->nh.iph;
  struct dsr_src_rt_opt * srcopt = (struct dsr_src_rt_opt *) optptr;
  struct rxmt_info * rxmtinfo = (struct rxmt_info * ) skb->cb;
  struct rtable * rth;
  __u32 fwdaddr;
  int err;

  int n_addr;  /* number of addresses on source route */
  int i;       /* index to the next address to be visited */

  n_addr = (srcopt->len - 2) / 4;

  /* don't process src rt if it is to us or we are the gateway */
  if (iph->daddr == IPADDR || (gw == 1 && NOT_DSR_SUBNET(iph->daddr))) {
    add_reverse_route(iph->saddr, srcopt->addr, n_addr);
    return 0;
  }

  /* we are not gateway and it is headed for an external node */
  if (NOT_DSR_SUBNET(iph->daddr))
    return -1;
```

```
  (srcopt->segs_left)--;
  i = n_addr - srcopt->segs_left;
  DSR_ASSERT(i > 0);

  add_forward_route(iph->daddr, srcopt->addr + i, srcopt->segs_left);
  add_reverse_route(iph->saddr, srcopt->addr, i - 1);

  /* if ttl is about to expire and we are supposed to forward */
  if (iph->ttl == 1) {
    return 0;
  }

  rxmtinfo->srcrt = srcopt;
  //printk("rxmtinfo:%u\n", rxmtinfo);
  //printk("proc src rt rxmtinfo->srcrt:%u\n", rxmtinfo->srcrt);
  if (srcopt->segs_left == 0) {
    //printk("route to %u.%u.%u.%u\n", NIPQUAD(iph->daddr));
    /* this will also forward the last hop to the gateway correctly as well */
    fwdaddr = iph->daddr;
  } else {
    //printk("route to %u.%u.%u.%u\n", NIPQUAD(srcopt->addr[i]));
    fwdaddr = srcopt->addr[i];
  }

  /*
   * we know the route right here so we do the routing first using dsr info
   * and the os won't route it again since it already routed by us. the os
   * would not route correctly since it will assume the destination is
   * reachable since we are on the same subnet.
   */
  if ((err = ip_route_input(skb, fwdaddr, iph->saddr, iph->tos, skb->dev))) {
    printk("ip_route_input failed dropping packet err:%i\n", err);
    dump_skb_ptr(skb);
    dump_skb(PF_INET, skb);
    return -1;
  }

  /*
   * we are forwarding packets within the same subnet and ip_route_input
   * will send icmp redirects at a backed off rate. we don't really want
   * this. the next two lines is to stop icmp redirects at ip_forward()
   */
  rth = (struct rtable *) skb->dst;
  DSR_ASSERT(rth != NULL);
  rth->rt_flags &= (~(RTCF_DOREDIRECT));
  return 0;
}
```

A.11. dsr_output.h

```
#ifndef DSR_OUTPUT_H
```

```
#define DSR_OUTPUT_H

#include <linux/skbuff.h>
#include "dsr_header.h"

void dsr_send(struct sk_buff **skb, struct rt_entry * dsr_rt,
        int (*output)(struct sk_buff*));
void send_rt_reply(const struct sk_buff * skb, struct dsr_rt_req_opt * rtreq);
void send_rt_err(struct sk_buff * skb);
void send_rt_req(__u32 taddr);
void rebcast_rt_req (const struct sk_buff * skb, unsigned char * optptr);
void send_ack_reply(const struct sk_buff * skb, unsigned char * optptr);
int dsr_output(struct sk_buff *skb);

#endif /* DSR_OUTPUT_H */
```

A.12. dsr_output.c

```
#include "dsr_output.h"
#include "dsr_debug.h"
#include "dsr_route.h"
#include "dsr_queue.h"

extern __u32 IPADDR;
extern __u32 NETMASK;
extern __u32 BCAST;
extern __u32 NETWORK;
extern int  gw;
extern int  hhl;
extern __u16 ack_id;
extern __u16 rtreq_id;
extern struct sk_buff_head rxmt_q;
extern struct rt_entry rt_cache[];
extern struct rt_req_entry rt_req_table[];
extern int rt_req_table_size;
extern struct sk_buff_head jitter_q;
/* stats */
extern unsigned int stat_send_rt_err;
extern unsigned int stat_send_rt_req;
extern unsigned int stat_send_rt_reply;
extern unsigned int stat_dsr_send;
extern unsigned int stat_dsr_frag;
extern unsigned int stat_output_err;

/*
 * this function allows us to send packets without traversing
 * the rest of netfilter. used for dsr specific packets.
 */
int dsr_output(struct sk_buff *skb) {

  struct net_device *dev = skb->dst->dev;
```

```
  struct dst_entry *dst = skb->dst;
  struct hh_cache *hh = dst->hh;

  skb->dev = dev;
  skb->protocol = __constant_htons(ETH_P_IP);

  if (hh) {
    read_lock_bh(&hh->hh_lock);
    memcpy(skb->data - 16, hh->hh_data, 16);
    read_unlock_bh(&hh->hh_lock);
    skb_push(skb, hh->hh_len);
    return hh->hh_output(skb);
  } else if (dst->neighbour)
    return dst->neighbour->output(skb);

  printk(KERN_DEBUG "dsr_output\n");
  kfree_skb(skb);
  return -EINVAL;
}

struct sk_buff * create_dsr_packet(int length, __u32 daddr) {

  struct sk_buff * newskb;
  struct iphdr * iph;

  /* 20 for the ip header */
  newskb = alloc_skb(20+length+hhl+15,GFP_ATOMIC);

  if (newskb == NULL) {
    return NULL;
  } else {
    skb_reserve(newskb, (hhl+15)&~15);
    skb_put(newskb, 20+length); /* iph + dsr stuff */
    newskb->nh.raw = newskb->data;

    iph = newskb->nh.iph;
    iph->version  = 4;
    iph->ihl      = 5;
    iph->tos      = 0;
    iph->frag_off = 0;
    iph->frag_off |= htons(IP_DF);
    iph->ttl      = MAX_ROUTE_LEN + 1;
    iph->daddr    = daddr;
    iph->saddr    = IPADDR;
    iph->protocol = DSR_PROTOCOL;
    iph->tot_len  = htons(newskb->len);
    iph->id       = 0;

    ip_send_check(iph);
    return newskb;
  }
}

void jitter_send_timeout(unsigned long data) {
```

```
  struct sk_buff * skb = (struct sk_buff *) data;
  skb_unlink(skb);
  dsr_output(skb);
}

void jitter_send(struct sk_buff * skb) {

  struct jitter_send * jitter = (struct jitter_send *) skb->cb;

  skb_queue_tail(&jitter_q, skb);
  init_timer(&(jitter->timer));
  jitter->timer.function = jitter_send_timeout;
  jitter->timer.data = (unsigned long) skb;
  jitter->timer.expires = jiffies +
    ((net_random() % BCAST_JITTER) * HZ) / 1000;
  add_timer(&(jitter->timer));
}

void add_ack_req_opt(unsigned char * buff) {

  struct dsr_ack_req_opt * ackreq = (struct dsr_ack_req_opt *)(buff);

  ackreq->type = ACK_REQ;
  ackreq->len = 6;
  ack_id++;
  ackreq->ident = ack_id;
  ackreq->saddr = IPADDR;
}

int add_src_rt_opt(struct sk_buff * skb, unsigned char * buff,
                   const struct rt_entry * rt_ptr) {

  const struct iphdr * iph = skb->nh.iph;
  struct dsr_src_rt_opt * srcopt = (struct dsr_src_rt_opt *) (buff);
  struct rxmt_info * rxmtinfo = (struct rxmt_info * ) skb->cb;
  int i;
  __u32 fwdaddr;

  DSR_ASSERT(rt_ptr != NULL);

  srcopt->type = SRC_ROUTE;
  srcopt->segs_left = rt_ptr->segs_left;
  srcopt->len = (rt_ptr->segs_left * 4) + 2;
  // not implemented yet
  // srcopt->salvage = 0;
  // srcopt->reserved = 0;
  // srcopt->lasthopx = 0;
  // srcopt->firsthopx = 0;

  for (i = 0;i < rt_ptr->segs_left;i++)
    srcopt->addr[i] = rt_ptr->addr[i];

  if (rt_ptr->segs_left == 0)
    fwdaddr = iph->daddr;
```

```
  else
    fwdaddr = srcopt->addr[0];

  rxmtinfo->srcrt = srcopt;
  //printk("rxmtinfo:%u\n", rxmtinfo);
  //printk("add src rt rxmtinfo->srcrt:%u\n", rxmtinfo->srcrt);

  if (gw == 1 && DSR_SUBNET(fwdaddr) && NOT_DSR_SUBNET(iph->saddr)) {
    /* release old route */
    dst_release(skb->dst);
    if (reroute_output(skb, IPADDR, fwdaddr) != 0) {
      printk("add_src_rt: route failed\n");
      return -1;
    }
  } else {
    if (reroute_output(skb, iph->saddr, fwdaddr) != 0)
      return -1;
  }
  return 0;
}

void dsr_options_fragment(struct sk_buff * skb)
{
  unsigned char * optptr = skb->nh.raw;
  struct ip_options * opt = &(IPCB(skb)->opt);
  int  l = opt->optlen;
  int  optlen;

  while (l > 0) {
    switch (*optptr) {
    case IPOPT_END:
      return;
    case IPOPT_NOOP:
      l--;
      optptr++;
      continue;
    }
    optlen = optptr[1];
    if (optlen<2 || optlen>l)
      return;
    if (!IPOPT_COPIED(*optptr))
      memset(optptr, IPOPT_NOOP, optlen);
    l -= optlen;
    optptr += optlen;
  }
  opt->ts = 0;
  opt->rr = 0;
  opt->rr_needaddr = 0;
  opt->ts_needaddr = 0;
  opt->ts_needtime = 0;
  return;
}

void dsr_fragment_send(struct sk_buff *skb, int dsrlen, int ackreqoff,
```

```
                       int srcrtoff, int (*output)(struct sk_buff*))
{
  struct iphdr *iph;
  unsigned char *raw;
  unsigned char *ptr;
  struct net_device *dev;
  struct sk_buff *skb2;
  unsigned int mtu, hlen, left, len;
  int offset;
  int not_last_frag;
  struct rtable *rt = (struct rtable*)skb->dst;
  int err = 0;

  /* alex: */
  struct rxmt_info * rxmtinfo;

  dev = rt->u.dst.dev;

  /* Point into the IP datagram header */
  raw = skb->nh.raw;
  iph = (struct iphdr*)raw;

  /* Setup starting values. */
  /* alex: +dsrlen */
  hlen = iph->ihl * 4 + dsrlen;
  left = ntohs(iph->tot_len) - hlen;  /* Space per frame */
  mtu = rt->u.dst.pmtu - hlen;        /* Size of data space */
  ptr = raw + hlen;                   /* Where to start from */

  /* Fragment the datagram. */
  offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
  not_last_frag = iph->frag_off & htons(IP_MF);

  /* Keep copying data until we run out. */
  while(left > 0) {
    len = left;
    /* IF: it doesn't fit, use 'mtu' - the data space left */
    if (len > mtu)
      len = mtu;
    /* IF: we are not sending upto and including the packet end
       then align the next start on an eight byte boundary */
    if (len < left) {
      len &= ~7;
    }
    /* Allocate buffer. */
    if ((skb2 = alloc_skb(len+hlen+dev->hard_header_len+15,
        GFP_ATOMIC)) == NULL) {
      NETDEBUG(printk(KERN_INFO "dsr: frag: no memory for new fragment!\n"));
      err = -ENOMEM;
      goto fail;
    }

    /* Set up data on packet */
    skb2->pkt_type = skb->pkt_type;
```

```
    skb2->priority = skb->priority;
    skb_reserve(skb2, (dev->hard_header_len+15)&~15);
    skb_put(skb2, len + hlen);
    skb2->nh.raw = skb2->data;
    skb2->h.raw = skb2->data + hlen;

    /* Charge the memory for the fragment to any owner it might possess */
    if (skb->sk)
      skb_set_owner_w(skb2, skb->sk);
    skb2->dst = dst_clone(skb->dst);
    skb2->dev = skb->dev;

    /* Copy the packet header into the new buffer. */
    memcpy(skb2->nh.raw, raw, hlen);

    /* Copy a block of the IP datagram. */
    memcpy(skb2->h.raw, ptr, len);
    left -= len;

    /* Fill in the new header fields. */
    iph = skb2->nh.iph;
    iph->frag_off = htons((offset >> 3));

    /* ANK: dirty, but effective trick. Upgrade options only if
     * the segment to be fragmented was THE FIRST (otherwise,
     * options are already fixed) and make it ONCE
     * on the initial skb, so that all the following fragments
     * will inherit fixed options.
     */
    if (offset == 0)
      dsr_options_fragment(skb);

    /*
     * Added AC : If we are fragmenting a fragment that's not the
     * last fragment then keep MF on each bit
     */
    if (left > 0 || not_last_frag)
      iph->frag_off |= htons(IP_MF);
    ptr += len;
    offset += len;

#ifdef CONFIG_NETFILTER
    /* Connection association is same as pre-frag packet */
    skb2->nfct = skb->nfct;
    nf_conntrack_get(skb2->nfct);
#ifdef CONFIG_NETFILTER_DEBUG
    skb2->nf_debug = skb->nf_debug;
#endif
#endif

    /* Put this fragment into the sending queue. */
    IP_INC_STATS(IpFragCreates);

    iph->tot_len = htons(len + hlen);
```

```
    ip_send_check(iph);

    /* alex: */
    rxmtinfo = (struct rxmt_info *) skb2->cb;
    rxmtinfo->srcrt = (struct dsr_src_rt_opt *)
      (skb2->nh.raw + srcrtoff);
    add_ack_req_opt(skb2->nh.raw + ackreqoff);
    /* alex: */
    ack_q_add(&rxmt_q, skb2, DSR_RXMT_BUFFER_SIZE);

    err = output(skb2);
    if (err)
      goto fail;
  }
  kfree_skb(skb);
  //IP_INC_STATS(IpFragOKs);
  //return err;
  return;
 fail:
  /* alex: */
  stat_output_err++;
  kfree_skb(skb);
#ifdef DSR_DEBUG
  printk("dsr_fragment_send failed, err:%i\n", err);
#endif
  return;
  //IP_INC_STATS(IpFragFails);
  //return err;
}

void dsr_send(struct sk_buff **skb, struct rt_entry * dsr_rt,
              int (*output)(struct sk_buff*)) {

  struct iphdr * iph = (*skb)->nh.iph;
  struct sk_buff * newskb;
  struct dsr_hdr * dsrhdr;
  unsigned char * optptr;
  /* size is the total amout of stuff added to the packet */
  int size = (dsr_rt->segs_left*4)+4 + 4 + 8; /* src rt + dsrhdr + ackreq */

  /*
   * nasty nasty nasty !!! need to keep original headroom free
   * otherwise the kernel will panic and crash
   * more nasty stuff, must do (size+15)&~15
   */
  newskb = skb_copy_expand(*skb, ((size+15)&~15) + skb_headroom(*skb),
                           skb_tailroom(*skb), GFP_ATOMIC);

  /* can't allocate extra buffer space */
  if (newskb == NULL) {
    kfree_skb(*skb);
    return;
  }
```

```
  /* set original owner */
  if ((*skb)->sk != NULL)
    skb_set_owner_w(newskb, (*skb)->sk);

  /* free old skb */
  kfree_skb(*skb);
  *skb = newskb;

  (*skb)->nh.raw = skb_push(*skb, size);

  /* copy ip header, iph points to the old header */
  memmove( (*skb)->nh.raw, (*skb)->nh.raw + size, (iph->ihl)*4);

  /* iph points to the new header now */
  iph = (*skb)->nh.iph;
  dsrhdr = (struct dsr_hdr *) ( (*skb)->nh.raw + (iph->ihl)*4);
  dsrhdr->nexthdr = iph->protocol;
  dsrhdr->reserved = 0;
  dsrhdr->length = htons(size - 4);
  iph->protocol = DSR_PROTOCOL;

  /* set the don't fragment flag */
  iph->frag_off |= htons(IP_DF);

  /* change length and recalculate checksum */
  iph->tot_len = htons( (*skb)->len );
  ip_send_check(iph);

  /* ip and dsr header is done, now have to add dsr options */
  optptr = (unsigned char *) (dsrhdr + 1);
  add_src_rt_opt(*skb, optptr, dsr_rt);
  optptr += dsr_rt->segs_left * 4 + 4;

  /* optptr now points to the empty ack req option */
  if ( (*skb)->len > (*skb)->dst->pmtu) {
    int ackreqoff = optptr - (*skb)->nh.raw;
    int srcrtoff = ackreqoff - (dsr_rt->segs_left * 4 + 4);
#ifdef DSR_DEBUG
    printk("dsr fragment send\n");
#endif
    /* select new id if we need to */
    if ( (*skb)->nh.iph->id == 0)
      ip_select_ident( (*skb)->nh.iph, (*skb)->dst);

    stat_dsr_frag++;
    dsr_fragment_send(*skb, size, ackreqoff, srcrtoff, output);
  } else {
#ifdef DSR_DEBUG
    printk("dsr normal send\n");
#endif
    add_ack_req_opt(optptr);
    ack_q_add(&rxmt_q, *skb, DSR_RXMT_BUFFER_SIZE);
    stat_dsr_send++;
```

```
    if(output(*skb)) {
      stat_output_err++;
#ifdef DSR_DEBUG
      printk("dsr send failed\n");
#endif
    }
  }
  return;
}

void finish_send_rt_req(__u32 taddr) {

  struct iphdr * iph;
  struct sk_buff * newskb;
  struct dsr_hdr * dsrhdr;
  struct dsr_rt_req_opt * reqopt;

  /* create rt req packet */
  newskb = create_dsr_packet(4+8, BCAST);

  if (newskb != NULL) {
    iph = newskb->nh.iph;
    dsrhdr = (struct dsr_hdr *) (newskb->nh.raw + (iph->ihl)*4);
    reqopt = (struct dsr_rt_req_opt *) (dsrhdr + 1);

    dsrhdr->nexthdr = NO_NEXT_HEADER;
    dsrhdr->length = htons(8);

    reqopt->type = ROUTE_REQ;
    reqopt->len = 6; /* 2 + 4 */
    reqopt->ident = rtreq_id;
    rtreq_id++;
    reqopt->taddr = taddr;

    if (reroute_output(newskb, iph->saddr, iph->daddr) == 0) {
      stat_send_rt_req++;
      jitter_send(newskb);
    } else {
      kfree_skb(newskb);
    }
  }
}

void rt_req_timeout(unsigned long data) {

  //printk("rt req timeout at %u\n", CURRENT_TIME);
  DSR_ASSERT(data < REQ_TABLE_SIZE);
  if (rt_req_table[data].n_req < MAX_REQ_RXMT) {
    /* send another rt req */
    finish_send_rt_req(rt_req_table[data].taddr);
    rt_req_table[data].n_req++;

    if (rt_req_table[data].timeout * 2 < (MAX_REQ_PERIOD * HZ))
      rt_req_table[data].timeout *= 2;
```

```
    else
      rt_req_table[data].timeout = (MAX_REQ_PERIOD * HZ);

    rt_req_table[data].timer.expires = jiffies + rt_req_table[data].timeout;
    add_timer(&(rt_req_table[data].timer));
  } else {
    /* remove rt req entry */
    rt_req_table[data].taddr = 0;
    rt_req_table_size--;
  }
}

void send_rt_req(__u32 taddr) {

  int i;
  int oldest = 0;
  int empty = 0;
  int replace;

  for (i = 0;i < REQ_TABLE_SIZE;i++) {
    if (rt_req_table[i].taddr != 0) {
      if (rt_req_table[i].taddr == taddr) {
        //printk("send rt req found taddr\n");
        return;
      } else {
        if (rt_req_table[i].time < rt_req_table[oldest].time)
          oldest = i;
      }
    } else {
      empty = i;
    }
  }

  if (rt_req_table_size < REQ_TABLE_SIZE) {
    /* create new entry */
    replace = empty;
    rt_req_table_size++;
  } else {
    /* replace old entry */
    replace = oldest;
    del_timer(&(rt_req_table[oldest].timer));
  }

  rt_req_table[replace].taddr = taddr;
  rt_req_table[replace].time = jiffies;
  rt_req_table[replace].timeout = (REQ_PERIOD * HZ) / 1000;
  rt_req_table[replace].n_req = 1;

  init_timer(&(rt_req_table[replace].timer));
  rt_req_table[replace].timer.function = rt_req_timeout;
  rt_req_table[replace].timer.data = replace;
  rt_req_table[replace].timer.expires = jiffies +
    rt_req_table[replace].timeout;
  add_timer(&(rt_req_table[replace].timer));
```

```
  //printk("finish send\n");
  finish_send_rt_req(taddr);
}

void send_rt_reply(const struct sk_buff * skb, struct dsr_rt_req_opt * rtreq) {

  /* skb is a rt req packet */
  struct iphdr * iph = skb->nh.iph;
  struct dsr_src_rt_opt * srcrt;
  struct rxmt_info * rxmtinfo;
  struct dsr_rt_reply_opt * rtreply;
  struct dsr_padn_opt * padn;
  struct dsr_hdr * dsrhdr;
  unsigned char * optptr;
  struct sk_buff * newskb;
  int n_addr = (rtreq->len - 6) / 4;
  int src_rt_size = 4 + (n_addr * 4);
  int rt_reply_size = 5 + (n_addr * 4);
  int j;
  __u32 fwdaddr;

  //printk("send rt err size %i\n", src_rt_size);
  /* dsr hdr + rt reply + ack req + src rt + pad3*/
  newskb = create_dsr_packet(4+rt_reply_size+8+src_rt_size+3, iph->saddr);

  if (newskb != NULL) {
    iph = newskb->nh.iph;
    dsrhdr = (struct dsr_hdr *) (newskb->nh.raw + (iph->ihl)*4);
    //ackopt = (struct dsr_ack_opt *) (dsrhdr + 1);

    dsrhdr->nexthdr = NO_NEXT_HEADER;
    dsrhdr->length = htons(rt_reply_size+8+src_rt_size+3);

    /* pad3 */
    optptr = (unsigned char *) (dsrhdr + 1);
    padn = (struct dsr_padn_opt *) optptr;
    padn->type = PADN;
    padn->len = 3 - 2;

    optptr += 3;
    rtreply = (struct dsr_rt_reply_opt *) optptr;
    rtreply->type = ROUTE_REPLY;
    rtreply->len = rt_reply_size - 2;
    rtreply->ident = rtreq->ident;
    optptr += rt_reply_size;

    //printk("send err next hop: %u.%u.%u.%u\n", NIPQUAD(rterr->typeinfo));

    /* add an ack req to route error message */
    add_ack_req_opt(optptr);
    optptr += 8; /* ack req */

    srcrt = (struct dsr_src_rt_opt *) optptr;
    srcrt->type = SRC_ROUTE;
```

```
    srcrt->len = src_rt_size - 2;
    srcrt->segs_left = n_addr;
    for (j = 0;j < n_addr;j++) {
      rtreply->addr[j] = rtreq->addr[j];
      srcrt->addr[j] = rtreq->addr[n_addr - j - 1];
    }

    if (srcrt->segs_left == 0)
      fwdaddr = iph->daddr;
    else
      fwdaddr = srcrt->addr[0];

    rxmtinfo = (struct rxmt_info *) newskb->cb;
    rxmtinfo->srcrt = srcrt;
    //printk("rxmtinfo:%u\n", rxmtinfo);
    //printk("send rt reply rxmtinfo->srcrt:%u\n", rxmtinfo->srcrt);

    if (reroute_output(newskb, iph->saddr, fwdaddr) == 0) {
      stat_send_rt_reply++;
      ack_q_add(&rxmt_q, newskb, DSR_RXMT_BUFFER_SIZE);
      dsr_output(newskb);
    } else {
      kfree_skb(newskb);
    }
  }
}

void send_rt_err(struct sk_buff * skb) {

  struct iphdr * iph = skb->nh.iph;
  struct rxmt_info * rxmtinfo = (struct rxmt_info *) skb->cb;
  struct dsr_rt_err_opt * rterr;
  struct dsr_src_rt_opt * srcrt;
  struct dsr_hdr * dsrhdr;
  unsigned char * optptr;
  struct sk_buff * newskb;
  const struct rtable * rt = (struct rtable *) skb->dst;
  int n_addr;       /* number of addresses on source route */
  int index;        /* index to the next address to be visited */
  int src_rt_size, j;
  __u32 fwdaddr;

  DSR_ASSERT(rt != NULL);
  DSR_ASSERT(gw == 0 || DSR_SUBNET(skb->nh.iph->daddr));
#ifdef DSR_DEBUG
  printk("send rt err\n");
#endif
  //printk("rxmtinfo:%u\n", rxmtinfo);
  //printk("rxmtinfo->srcrt:%u\n", rxmtinfo->srcrt);
  n_addr = (rxmtinfo->srcrt->len - 2) / 4;
  index = n_addr - rxmtinfo->srcrt->segs_left;

  if (rxmtinfo->srcrt->segs_left == 0) {
    if (NOT_DSR_SUBNET(skb->nh.iph->daddr)) {
```

```
      remove_route(rt_cache, IPADDR, rt->rt_gateway);
    } else {
      remove_route(rt_cache, IPADDR, skb->nh.iph->daddr);
    }
  } else {
    remove_route(rt_cache, IPADDR, rxmtinfo->srcrt->addr[index]);
  }

  if (index == 0)
    return;
  else
    src_rt_size = 4 + (index - 1) * 4;
  //printk("send rt err size %i\n", src_rt_size);
  /* dsr hdr + rt err + ack req + src rt */
  newskb = create_dsr_packet(4+16+8+src_rt_size, iph->saddr);

  if (newskb != NULL) {
    iph = newskb->nh.iph;
    dsrhdr = (struct dsr_hdr *) (newskb->nh.raw + (iph->ihl)*4);
    rterr = (struct dsr_rt_err_opt *) (dsrhdr + 1);
    //ackopt = (struct dsr_ack_opt *) (dsrhdr + 1);

    dsrhdr->nexthdr = NO_NEXT_HEADER;
    dsrhdr->length = htons(16+8+src_rt_size);

    rterr->type = ROUTE_ERROR;
    rterr->len = 16 - 2;
    rterr->errtype = 1;    /* node unreachable */
    rterr->errsaddr = IPADDR;
    rterr->errdaddr = iph->saddr;

    if (rxmtinfo->srcrt->segs_left == 0) {
      if (NOT_DSR_SUBNET(skb->nh.iph->daddr)) {
        rterr->typeinfo = rt->rt_gateway;
      } else {
        rterr->typeinfo = skb->nh.iph->daddr;
      }
    } else {
      rterr->typeinfo = rxmtinfo->srcrt->addr[index];
    }
    optptr = (unsigned char *) (rterr + 1);

    //printk("send err next hop: %u.%u.%u.%u\n", NIPQUAD(rterr->typeinfo));

    /* add an ack req to route error message */
    add_ack_req_opt(optptr);
    optptr += 8;  /* ack req */

    /* need to reverse half of the orig src rt and use that as the src rt */
    srcrt = (struct dsr_src_rt_opt *) optptr;
    srcrt->type = SRC_ROUTE;
    srcrt->len = src_rt_size - 2;
    srcrt->segs_left = index - 1;
    for (j = 0;j < srcrt->segs_left;j++)
```

```
      srcrt->addr[j] = rxmtinfo->srcrt->addr[index-j-2];

    if (srcrt->segs_left == 0)
      fwdaddr = iph->daddr;
    else
      fwdaddr = srcrt->addr[0];

    rxmtinfo = (struct rxmt_info *) newskb->cb;
    rxmtinfo->srcrt = srcrt;
    //printk("rxmtinfo:%u\n", rxmtinfo);
    //printk("send rt err rxmtinfo->srcrt:%u\n", rxmtinfo->srcrt);

    if (reroute_output(newskb, iph->saddr, fwdaddr) == 0) {
      stat_send_rt_err++;
      ack_q_add(&rxmt_q, newskb, DSR_RXMT_BUFFER_SIZE);
      dsr_output(newskb);
    } else {
      kfree_skb(newskb);
    }
  }
}

void rebcast_rt_req(const struct sk_buff * skb, unsigned char * optptr) {

  struct iphdr * iph = skb->nh.iph;
  struct dsr_rt_req_opt * rtreq; // = (struct dsr_rt_req_opt *) optptr;
  const int optoff = optptr - skb->nh.raw;
  struct sk_buff * newskb;
  struct dsr_hdr * dsrhdr = (struct dsr_hdr *) (skb->nh.raw + (iph->ihl)*4);

  newskb = skb_copy_expand(skb, skb_headroom(skb) + 4,
                           skb_tailroom(skb), GFP_ATOMIC);

  /* can't allocate extra buffer space */
  if (newskb == NULL)
    return;

  newskb->nh.raw = skb_push(newskb, 4);

  memmove(newskb->nh.raw, newskb->nh.raw + 4,
          (iph->ihl)*4 + 4 + ntohs(dsrhdr->length));

  iph = newskb->nh.iph;
  dsrhdr = (struct dsr_hdr *) (newskb->nh.raw + (iph->ihl)*4);
  rtreq = (struct dsr_rt_req_opt *) (newskb->nh.raw + optoff);

  rtreq->addr[(rtreq->len - 6)/4] = IPADDR;
  rtreq->len += 4;

  dsrhdr->length = htons(ntohs(dsrhdr->length) + 4);

  iph->tot_len = htons(newskb->len);
  ip_send_check(iph);
```

```
  if (reroute_output(newskb, IPADDR, BCAST) == 0) {
    jitter_send(newskb);
  } else {
    kfree_skb(newskb);
  }
}

void send_ack_reply(const struct sk_buff * skb, unsigned char * optptr) {

  struct iphdr * iph;
  struct dsr_ack_req_opt * ackreq = (struct dsr_ack_req_opt *) optptr;
  struct dsr_hdr * dsrhdr;
  struct dsr_ack_opt * ackopt;
  struct sk_buff * newskb;

  /* create ack reply */
  DSR_ASSERT(skb->dev != NULL);
  newskb = create_dsr_packet(4+12, ackreq->saddr);

  if (newskb != NULL) {
    iph = newskb->nh.iph;
    dsrhdr = (struct dsr_hdr *) (newskb->nh.raw + (iph->ihl)*4);
    ackopt = (struct dsr_ack_opt *) (dsrhdr + 1);

    dsrhdr->nexthdr = NO_NEXT_HEADER;
    dsrhdr->length = htons(12);

    ackopt->type = ACK;
    ackopt->len = 10;
    ackopt->ident = ackreq->ident;
    ackopt->saddr = iph->saddr;
    ackopt->daddr = iph->daddr;

    if (reroute_output(newskb, iph->saddr, iph->daddr) == 0) {
      dsr_output(newskb);
    } else {
      kfree_skb(newskb);
    }
  }
}
```

A.13. dsr_queue.h

```
#ifndef DSR_QUEUE_H
#define DSR_QUEUE_H

#include "dsr_header.h"
#include "dsr_debug.h"
#include "dsr_output.h"

void ack_q_add(struct sk_buff_head * list, struct sk_buff * skb, int q_len);
```

```
void send_q_add(struct sk_buff_head * list, struct sk_buff * skb,
    int q_len, __u32 fwdaddr, int (*output)(struct sk_buff *));

#endif /* DSR_QUEUE_H */
```

A.14. dsr_queue.c

```
#include "dsr_queue.h"

extern __u16 ack_id;
/* stats */
extern unsigned int stat_send_q_timeout;
extern unsigned int stat_send_q_drop;
extern unsigned int stat_ack_q_drop;
extern unsigned int stat_ack_q_timeout;
extern unsigned int stat_ack_q_resend;

void send_timeout(unsigned long data) {

  struct sk_buff * skb = (struct sk_buff *) data;
  DSR_ASSERT(skb != NULL);

  //printk("send timed out !!!\n");
  stat_send_q_timeout++;
  skb_unlink(skb);
  kfree_skb(skb);
}

void send_q_add(struct sk_buff_head * list, struct sk_buff * skb,
    int q_len, __u32 fwdaddr, int (*output)(struct sk_buff *)) {

  struct send_info * sendinfo;

  DSR_ASSERT(skb_queue_len(list) <= q_len);
  if (skb_queue_len(list) == q_len) {
    struct sk_buff * headskb = skb_dequeue(list);
    sendinfo = (struct send_info *) headskb->cb;
    del_timer(&(sendinfo->timer));
    kfree_skb(headskb);
    stat_send_q_drop++;
#ifdef DSR_DEBUG
    printk("send q too long\n");
#endif
  }

  sendinfo = (struct send_info *) skb->cb;
  init_timer(&(sendinfo->timer));
  sendinfo->fwdaddr = fwdaddr;
  sendinfo->output = output;
  sendinfo->timer.function = send_timeout;
  sendinfo->timer.expires = jiffies + (SEND_BUFFER_TIMEOUT * HZ);
```

```
  sendinfo->timer.data = (unsigned long) skb;
  add_timer(&(sendinfo->timer));
  skb_queue_tail(list, skb);
}

void ack_timeout(unsigned long data) {

  struct sk_buff * skb = (struct sk_buff *) data;
  struct rxmt_info * rxmtinfo = (struct rxmt_info *) skb->cb;
  DSR_ASSERT(skb != NULL);

#ifdef DSR_DEBUG
  printk("ack timed out !!!\n");
  //printk("rxmtinfo:%u\n", rxmtinfo);
  //printk("rxmtinfo->srcrt:%u\n", rxmtinfo->srcrt);
#endif
  if (rxmtinfo->n_rxmt < DSR_MAXRXTSHIFT) {
    struct sk_buff * newskb;
    rxmtinfo->n_rxmt++;
#ifdef DSR_DEBUG
    printk("resend %i\n", rxmtinfo->n_rxmt);
#endif
    rxmtinfo->timer.expires = jiffies + ACK_TIMEOUT_JF;
    add_timer(&(rxmtinfo->timer));

    if ((newskb = skb_clone(skb, GFP_ATOMIC)) == NULL)
      return;

    stat_ack_q_resend++;
    dsr_output(newskb);
  } else {
#ifdef DSR_DEBUG
    printk("send rt err for ack %u\n", rxmtinfo->ident);
#endif
    stat_ack_q_timeout++;
    skb_unlink(skb);
    send_rt_err(skb);
    kfree_skb(skb);
  }
}

void ack_q_add(struct sk_buff_head * list, struct sk_buff * skb,
         int q_len/*, int (*output)(struct sk_buff *)*/) {

  struct sk_buff * newskb;
  struct rxmt_info * rxmtinfo;

  DSR_ASSERT(skb_queue_len(list) <= q_len);
  if (skb_queue_len(list) == q_len) {
    stat_ack_q_drop++;
    return;
  }

  newskb = skb_clone(skb, GFP_ATOMIC);
```

```
  if (newskb == NULL)
    return;

  rxmtinfo = (struct rxmt_info *) newskb->cb;

  rxmtinfo->ident = ack_id;
  rxmtinfo->n_rxmt = 0;

  init_timer(&(rxmtinfo->timer));
  rxmtinfo->timer.function = ack_timeout;
  rxmtinfo->timer.expires = jiffies + ACK_TIMEOUT_JF;
  rxmtinfo->timer.data = (unsigned long) newskb;
  add_timer(&(rxmtinfo->timer));

  skb_queue_tail(list, newskb);
}
```

A.15. dsr_route.h

```
#ifndef DSR_ROUTE_H
#define DSR_ROUTE_H

#include <linux/netfilter_ipv4.h>
#include <linux/netdevice.h>
#include <linux/proc_fs.h>
#include <linux/ip.h>
#include <linux/inet.h>
#include <net/checksum.h>
#include <net/icmp.h>

#include "dsr_header.h"

int reroute_output(struct sk_buff *skb, __u32 src, __u32 dst);
void add_reverse_route(__u32 daddr, __u32 * addr, int n_addr);
void add_forward_route(__u32 daddr, __u32 * addr, int n_addr);

/*
 * look up a route and clean out any expired routes
 * if a route is found it is set as new if no route is found NULL
 * is returned
 */
struct rt_entry * lookup_route(struct rt_entry * rt_cache, __u32 dst);
void remove_route(struct rt_entry * rt_cache, __u32 srchop, __u32 dsthop);

#endif /* DSR_ROUTE_H */
```

A.16. dsr_route.c

```
#include "dsr_route.h"
```

```
#include "dsr_debug.h"
#include "dsr_output.h"

extern __u32 IPADDR;
extern __u32 NETMASK;
extern __u32 NETWORK;
extern struct sk_buff_head send_q;
extern struct rt_entry rt_cache[];
extern struct rt_req_entry rt_req_table[];
extern int rt_cache_size;
extern int rt_req_table_size;
extern unsigned int stat_add_f_rt;
extern unsigned int stat_add_r_rt;
extern unsigned int stat_remove_rt;

/* FIXME: change in oif may mean change in hh_len. Check and realloc —RR */
/* modified from ipv4/netfilter/iptable_mangle.c */
int reroute_output(struct sk_buff *skb, __u32 src, __u32 dst) {
  struct iphdr *iph = skb->nh.iph;
  struct rtable *rt;
  struct rt_key key = { dst:dst,
      src:src,
      oif:skb->sk ? skb->sk->bound_dev_if : 0,
      tos:RT_TOS(iph->tos)|RTO_CONN,
  };

  if (ip_route_output_key(&rt, &key) != 0) {
    printk("dsr reroute_output: No more route.\n");
    return -EINVAL;
  }

  /* Drop old route. */
  dst_release(skb->dst);

  skb->dst = &rt->u.dst;
  return 0;
}

void remove_route(struct rt_entry * rt_cache, __u32 srchop, __u32 dsthop) {

  int i, j;
  stat_remove_rt++;
#ifdef DSR_DEBUG
  printk("rmv rt, srchop:%u.%u.%u.%u dsthop:%u.%u.%u.%u\n", NIPQUAD(srchop),
   NIPQUAD(dsthop));
#endif
  /*
   * brute force search for now, hash table later
   * remove every entry with srchop and dsthop
   */
  for (i = 0;i < ROUTE_CACHE_SIZE;i++) {
    if (rt_cache[i].dst != 0) {
      if (CURRENT_TIME - rt_cache[i].time < ROUTE_CACHE_TIMEOUT) {
  int finished;
```

```
  /* start - end - middle */
  if (rt_cache[i].segs_left == 0) {
    if(IPADDR == srchop && rt_cache[i].dst == dsthop) {
      rt_cache[i].dst = 0;
      rt_cache_size--;
    }
  } else if (IPADDR == srchop && rt_cache[i].addr[0] == dsthop) {
    rt_cache[i].dst = 0;
    rt_cache_size--;
  } else if (rt_cache[i].addr[rt_cache[i].segs_left - 1] == srchop
       && rt_cache[i].dst == dsthop) {
    rt_cache[i].dst = 0;
    rt_cache_size--;
  } else {
    finished = 0;
    for (j = 0;j < rt_cache[i].segs_left && finished == 0;j++) {
      if (rt_cache[i].addr[j] == srchop &&
    rt_cache[i].addr[j+1] == dsthop) {
        rt_cache[i].dst = 0;
        rt_cache_size--;
        finished = 1;
      }
    }
  }
      } else {
  /* timed out entry, clean it */
  rt_cache[i].dst = 0;
  rt_cache_size--;
      }
    }
  }
}

void add_new_forward_route(__u32 daddr, __u32 * addr, int n_addr) {

  struct rt_entry * rt;
  int i;

  if (rt_cache_size < ROUTE_CACHE_SIZE) {
    int finished = 0;
    rt = rt_cache;
    /* create new entry */
    for (i = 0;i < ROUTE_CACHE_SIZE && finished == 0;i++) {
      if (rt_cache[i].dst == 0) {
  finished = 1;
  rt = rt_cache + i;
      }
    }
    DSR_ASSERT(finished == 1);
    rt_cache_size++;
  } else {
    int oldest = 0;
    /* replace oldest entry */
    for (i = 1;i < ROUTE_CACHE_SIZE;i++) {
```

```
      if (rt_cache[i].dst != 0) {
  if (rt_cache[i].time < rt_cache[oldest].time) {
    oldest = i;
  }
      }
    }
    rt = rt_cache + oldest;
  }

  rt->dst = daddr;
  rt->time = CURRENT_TIME;
  rt->segs_left = n_addr;

  for (i = 0;i < n_addr;i++)
    rt->addr[i] = addr[i];
}

void add_new_reverse_route(__u32 daddr, __u32 * addr, int n_addr) {

  struct rt_entry * rt;
  int i;

  if (rt_cache_size < ROUTE_CACHE_SIZE) {
    int finished = 0;
    rt = rt_cache;
    /* create new entry */
    for (i = 0;i < ROUTE_CACHE_SIZE && finished == 0;i++) {
      if (rt_cache[i].dst == 0) {
  finished = 1;
  rt = rt_cache + i;
      }
    }
    DSR_ASSERT(finished == 1);
    rt_cache_size++;
  } else {
    int oldest = 0;
    /* replace oldest entry */
    for (i = 1;i < ROUTE_CACHE_SIZE;i++) {
      if (rt_cache[i].dst != 0) {
  if (rt_cache[i].time < rt_cache[oldest].time) {
    oldest = i;
  }
      }
    }
    rt = rt_cache + oldest;
  }

  rt->dst = daddr;
  rt->time = CURRENT_TIME;
  rt->segs_left = n_addr;

  for (i = 0;i < n_addr;i++)
    rt->addr[i] = addr[n_addr - i - 1];
}
```

```
void remove_rt_req_id(__u32 daddr) {
  int i, count;

  /* clear entry in rt req table */
  for (i = 0, count = 0;count < rt_req_table_size;i++) {
    DSR_ASSERT(i < REQ_TABLE_SIZE);
    if (rt_req_table[i].taddr != 0) {
      count++;
      if (rt_req_table[i].taddr == daddr) {
  del_timer(&(rt_req_table[i].timer));
  rt_req_table[i].taddr = 0;
  rt_req_table_size--;
  return;
      }
    }
  }
  return;
}

void check_send_q() {

  int i;
  struct rt_entry * rt;
  int q_len = skb_queue_len(&send_q);
  struct sk_buff * skbptr;
  struct send_info * sendinfo;

  /* see if we can send anything in the send queue */
  for (i = 0;i < q_len;i++) {
    skbptr = skb_dequeue(&send_q);
    sendinfo = (struct send_info *) skbptr->cb;
    DSR_ASSERT(skbptr != NULL);
    rt = lookup_route(rt_cache, sendinfo->fwdaddr);
    if (rt != NULL) {
      del_timer(&(sendinfo->timer));
      /* dsr_send should free the skb */
      dsr_send(&skbptr, rt, sendinfo->output);
    } else {
      skb_queue_tail(&send_q, skbptr);
    }
  }
}

int have_route(__u32 daddr, __u32 * addr, int n_addr) {

  int i, j;

  /* look through all the routes to see if we already have the SAME route */
  for (i = 0;i < ROUTE_CACHE_SIZE;i++) {
    if (rt_cache[i].dst != 0) {
      if (CURRENT_TIME - rt_cache[i].time < ROUTE_CACHE_TIMEOUT) {
  if (rt_cache[i].dst == daddr && rt_cache[i].segs_left == n_addr) {
    int diff = 0;
```

```
    for (j = 0;j < n_addr && diff == 0;j++) {
      /* route is different */
      if (rt_cache[i].addr[j] != addr[j])
        diff = 1;
    }
    /* if the route is the same then we return */
    if (diff == 0) {
      rt_cache[i].time = CURRENT_TIME;
      return 1;
    }
  }
      } else {
  /* timed out entry, clean it */
  rt_cache[i].dst = 0;
  rt_cache_size--;
      }
    }
  }
  return 0;
}

/* addr can be null */
void add_reverse_route(__u32 daddr, __u32 * addr, int n_addr) {

  int i;
  stat_add_r_rt++;

  for (i = 0;i < n_addr;i++) {
    if (have_route(addr[n_addr - i - 1], addr + n_addr - i, i) == 0) {
      add_new_reverse_route(addr[n_addr - i - 1], addr + n_addr - i, i);
      remove_rt_req_id(addr[n_addr - i - 1]);
    }
  }

  if (DSR_SUBNET(daddr) && have_route(daddr, addr, n_addr) == 0) {
    add_new_reverse_route(daddr, addr, n_addr);
    remove_rt_req_id(daddr);
  }
  check_send_q();
}

/* addr can be null */
void add_forward_route(__u32 daddr, __u32 * addr, int n_addr) {

  int i;
  stat_add_f_rt++;

  for (i = 0;i < n_addr;i++) {
    if (have_route(addr[i], addr + i - 1, i) == 0) {
      add_new_forward_route(addr[i], addr + i - 1, i);
      remove_rt_req_id(addr[i]);
    }
  }
```

```
  if (DSR_SUBNET(daddr) && have_route(daddr, addr, n_addr) == 0) {
    add_new_forward_route(daddr, addr, n_addr);
    remove_rt_req_id(daddr);
  }
  check_send_q();
}

struct rt_entry * lookup_route(struct rt_entry * rt_cache, __u32 dst) {

  int i;
  struct rt_entry * rt_ptr = NULL;

  /*
   * brute force search for now, hash table later
   * the most current entry with the smallest hop count will be used
   */
  for (i = 0;i < ROUTE_CACHE_SIZE;i++) {
    if (rt_cache[i].dst != 0) {
      if (CURRENT_TIME - rt_cache[i].time < ROUTE_CACHE_TIMEOUT) {
  if (rt_cache[i].dst == dst) {
    /* found an entry */
    if (rt_ptr == NULL)
      rt_ptr = rt_cache + i;
    else if (rt_cache[i].segs_left < rt_ptr->segs_left)
      rt_ptr = rt_cache + i;
    else if (rt_cache[i].segs_left == rt_ptr->segs_left)
      if (time_after(rt_cache[i].time, rt_ptr->time))
        rt_ptr = rt_cache + i;
  }
      } else {
  /* timed out entry, clean it */
  rt_cache[i].dst = 0;
  rt_cache_size--;
      }
    }
  }

  /* refresh the route we found */
  if (rt_ptr != NULL)
    rt_ptr->time = CURRENT_TIME;

  return rt_ptr;
}
```

References

[1] A. Siu, "Piconet a wireless ad-hoc network for mobile handheld devices," Master's thesis, University of Queensland, St Lucia, Dept. Computer Science and Electrical Engineering, 2000.

[2] I. Keys, "Piconet a wireless ad-hoc network for mobile handheld devices," Master's thesis, University of Queensland, St Lucia, Dept. Computer Science and Electrical Engineering, 2000.

[3] S. Corson and J. Macker, "Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations." http://www.ietf.org/rfc/rfc2501.txt, January 1999.

[4] A. S. Tanenbaum, *Computer Networks*. Prentice-Hall, 3rd ed., 1996.

[5] IETF, "Mobile ad-hoc networks (manet)." http://www.ietf.org/html.charters/manet-charter.html, April 2001.

[6] P. Misra, "Routing protocols for ad hoc mobile wireless networks." ftp://ftp.netlab.ohio-state.edu/pub/jain/courses/cis788-99/adhoc_routing/index.html, June 2001.

[7] Bluetooth-SIG, "Specification of the bluetooth system, vol. 1, core." http://www.bluetooth.com/developer/specification/Bluetooth_11_Specifications_Book.pdf, Feburary 2001. Version 1.1.

[8] Bluetooth-SIG, "Specification of the bluetooth system, vol. 1, core." http://www.bluetooth.com/developer/specification/Bluetooth_11_Specifications_Book.pdf, Feburary 2001. Version 1.1.

[9] Lucent, "Orinoco pc card data sheet." ftp://ftp.orinocowireless.com/pub/docs/ORINOCO/BROCHURES/O_PC.pdf, March 2001.

[10] The-NetBSD-Foundation, "The netbsd project." http://www.netbsd.org, March 2001.

[11] The-NetBSD-Foundation, "Netbsd licensing and redistribution." http://www.netbsd.org/Goals/redistribution.html, October 2001.

[12] Linux-Online, "Gnu general public license." http://www.linux.org/info/gnu.html, October 2001.

[13] Palm, "Palm os." http://www.palmos.com, March 2001.

[14] C. Comstock, "Palm linux environment." http://palm-linux.sourceforge.net, May 2001.

[15] J. Dionne and M. Durrant, "uclinux." http://www.uclinux.org, March 2001.

[16] handhelds.org, "handhelds.org." http://www.handhelds.org, March 2001.

[17] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing* (T. Imielinski and H. Korth, eds.), ch. 5, pp. 153–181, Kluwer Academic Publishers, 1996.

[18] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," in *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, (Dallas, Texas), ACM, October 1998.

[19] D. B. Johnson, D. A. Maltz, and Y.-C. Hu, "The dynamic source routing protocol for mobile ad hoc networks." http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-05.txt, March 2001. Work in progress.

[20] P. Russell, "Linux netfilter hacking howto." http://netfilter.samba.org/unreliable-guides/netfilter-hacking-HOWTO/index.html, May 2001.

[21] D. A. Rusling, "The linux kernel." http://www.linuxdoc.org/LDP/tlk/tlk.html, June 2001.

[22] D. B. Johnson, D. A. Maltz, and Y.-C. Hu, "Flow state in the dynamic source routing protocol for mobile ad hoc networks." http://www.ietf.org/internet-drafts/draft-ietf-manet-dsrflow-00.txt, February 2001. Work in progress.

[23] Mobile-IP-Working-Group, "Ip routing for wireless/mobile hosts (mobileip)." http://www.ietf.org/html.charters/mobileip-charter.html, March 2001.