Development, Implementation and Quantification of an Ad-hoc Routing Protocol for Mobile Handheld Terminals

Nicholas Joseph Dearham

BScEng (Electronic)

Submitted in fulfillment of the academic requirements for the Degree of Master of Science in Electronic Engineering in the School of Electrical, Electronic and Computer Engineering at the University of Natal, Durban, South Africa

September 25, 2003

Abstract

An ad-hoc network is a collection of mobile nodes (wireless communication devices) that transmit data over systems that do not require any centralized control, such as that found in cellular networks. This makes ad-hoc networks suitable for military type applications, since there is no need for an established backbone infrastructure and hence no single-point-of-failure. However, other uses of ad-hoc systems include search and rescue missions, law enforcement operations, commercial and educational communication of laptop (and other handheld device) data, as well as in the transmission of environmental sensor information.

The mobile ad-hoc concept brings many design challenges. The dynamic freedom of movement from mobile nodes causes random, sometimes rapidly time changing topologies, which are inappropriate for use through traditional wired protocols. In addition, wireless networks generally contain greater bandwidth, processing and power constraints than their wired counterparts, since they are implemented on embedded mobile, handheld devices. Thus, a different approach is needed in the wireless network domain. This has resulted in wireless routing protocols employing adaptive, multi-hop, distributed methodologies in which each node additionally acts as a router for each of its neighbouring nodes, in order to achieve a large degree of network connectivity.

However, due to the broadcast nature of wireless transmissions, ad-hoc systems contain a point-to-multipoint communication architecture, making it well suited to multi-path traffic. One such application is in multicasting, which sends data from one source to two (or more) destinations. But, due to the shared characteristics of the communication channel, such traffic may cause multiple contentions and collisions to occur, which will degrade the efficiency and performance of a protocol.

This dissertation examines these different design tradeoffs through the use of a freely available simulation package, known as NS-2 (Network Simulator – version 2). In addition, a novel routing protocol, known as LAMP (Location Aided Multicasting Protocol), is developed to handle time-bounded audio information, which is employed in a network that consists of sixteen commercial handheld devices.

LAMP utilizes a destination-sequenced, next-hop routing table to forward multicast data. Since mobility causes neighbouring nodes to continually change, next-hop links need to be periodically updated. But, between each update period, a next-hop link may become broken. Thus, if a packet is required to be routed, for which its' next-hop link is unknown, LAMP reverts to a localized location aided flood to find a path to that destination. However, since flooding causes network congestion, it is only employed when its' table forwarding scheme fails.

Results have shown that LAMP improves packet delivery ratios by up to 5% over exisiting flood-limiting schemes. Furthermore, LAMP has been shown to be comparable to leading schemes, even when employed to route data to a single source-destination pair.

Preface

The research work presented in this dissertation was performed by Mr Nicholas Joseph Dearham, under the supervision of Mr. Stephen A. McDonald, at the University of Natal's school of Electrical, Electronic and Computer Engineering. This work has been conducted over the period February 2002 to September 2003, through the generous sponsorship of Armscor.

I thereby declare that the entire dissertation, unless otherwise indicated, is the author's work and has not been submitted in part, or in whole, to any other University for degree purposes.

Signed:

Name:

Date:

As the candidate's supervisor I have approved this thesis for submission.

Signed:

Name:

Date:

Acknowledgements

Firstly, I'd like to thank my supervisor, Stephen McDonald, for all the patience, guidance, honesty, support, and trust you have shown towards me, over the past twenty months. Your interest and involvement in the project has been a great motivational drive in success that was accomplished, especially when stumbling blocks arose. I, therefore, thank you for the giving of your time and for always making your office available for discussion.

Secondly, I'd like to thank Armscor for their financial generosity, since, without this backing, a project of this nature could simply not exist. In particular, I'd like to thank all their personnel for their criticism and personal interest, as this allowed me to see things from both the academic and corporate perspectives, which aided in the construction of simple (but complete) problem scenarios.

Next, I'd like to thank my family and friends for their patience and understanding. To my Dad, thanks for proof reading each page (even though you probably had no idea of what you read). To my Mom, thanks for all the times I took over your dinning room table, and to my brother, for all your advice on C++ debugging techniques. Then to Ed, Tahmid, Yunis, Yusuf, Dirk, Niven and Adrian, thanks for all the laughs and smiles, which made each long day at varsity worthwhile. And to Brett, Al, Andy, Ryan and Noel, thanks for your long-term friendship and may this continue on through the remainder of our lives.

Then to my Bible-study group, Gill, Adrian, Dirk, Trev and Claire, thanks for all your prayers, debates and challenging insights. These have grown me and open my eyes to truth that can be only found through diligent study of the most awesome book in the world, the Bible. May our group continue to make an impact in the lives of all whom we meet.

Lastly, thanks to my Lord and Savior, Jesus Christ. Without you, there would be no reason to live. You bring victory over death and are the only means to eternal salvation. I cannot begin to thank you for all you have done. From the time I took my first breath to today, you have always been the one and only person who has carried me and comforted me, during every second on my being, preventing me from ever being able to count your abundant blessing. I, hereby, dedicate this dissertation to you.

Publications

This work has been published and presented at the following conferences:

- N. J. Dearham, T. Quazi, and S. A. McDonald, "A Comparative Assessment of Ad-Hoc Routing Protocols," in *proc. South African Telecommunication, Networks and Applications Conference (SATNAC)*, 2002.
- N. J. Dearham and S. A. McDonald, "A Location Aided Multicasting Protocol (LAMP) for Sparse Ad-hoc Networks," in *proc. South African Telecommunication, Networks and Applications Conference (SATNAC)*, 2003.
- N. J. Dearham and S. A. McDonald, "A Handheld Implementation of a Location Aided Multicasting Protocol (LAMP)," in *proc. Military Information and Communications Symposium of South Africa (MICSSA)*, 2003. <Accepted for Publication>.

Table of Contents

Abbreviations and Acronyms

ABAM	Associativity-Based Ad hoc Multicast
ABR	Associativity-Based Routing
ACK	Acknowledgment
AHBP	Ad Hoc Broadcast Protocol
AMRIS	Ad hoc Multicast Routing protocol utilizing Increasing id-numberS
AMRoute	Ad hoc Multicast Routing
AODV	Ad hoc On-demand Distance Vector
API	Application Programming Interface
ARP	Address Resolution Protocol
ARPA	Advanced Research Project Agency
ARPANET	Advanced Research Projects Agency Network
ASCII	American Standard Code for Information Interchange
BRG	Broadcast Relay Gateway
BSD	Berkeley System Distribution
CAMP	Core Assisted Mesh Protocol
CBR	Constant Bit-Rate
CBT	Core Based Tree
CCA	Clear Channel Assessment
CGSR	Clusterhead-Gateway Switch Routing
CMOS	Complementary Metal Oxide Semiconductor
CMU	Carnegie Mellon University
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detect
CTS	Clear To Send
CW	Contention Window
DAG	Directed Acyclic Graph
DARPA	The department of Defences' Advanced Research Project Agency
DCF	Distributed Coordination Function
DNAT	Destination Network Address Translation
DoD	Department of Defense
DREAM	Distance Routing Effect Algorithm for Mobility
DRP	Dynamic Routing Protocol
DSDV	Destination-Sequenced Distance Vector

DSR	Dynamic State Routing
DSSS	Direct Sequence Spread Spectrum
DVMRP	Distance Vector Multicast Routing Protocol
EAC	Expected Additional Coverage
FGMP	Forwarding Group Multicast Protocol
FHSS	Frequency Hop Spread Spectrum
FIB	Forward Information Base
FIFO	First In First Out
FSP	Flooding with Self-Pruning
FSR	Fisheye State Routing
FTP	File Transfer Protocol
GDI	Graphics Device Interface
GE	General Electric
GG	Gabriel Graph
GPS	Global Positional System
GPSR	Greedy Perimeter Stateless Routing
GSR	Global State Routing
GUI	Graphical User Interface
HP	Hewlett-Packard
HSR	Hierarchal State Routing
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronic Engineers
IEFT	Internet Engineering Task Force
IFQ	Interface Queue
IGMP	Internet Group Management Protocol
IMP	Interface Message Processor
INET	Internet
IP	Internet Protocol
ISI	Information Science Institute
ISM	Industrial, Scientific and Medical
ISO	International Organization for Standardization
ISP	Internet Service Provider
JVM	Java Virtual Machine
LAMP	Location Aided Multicast Protocol
LAN	Local Area Network
LAR	Location Aided Routing

LBL	Ernest Orlando Lawrence Berkeley National Laboratory
LBM	Location-Based Multicast
LCD	Liquid Crystal Display
LMR	Lightweight Mobile Routing
LOTAR	Location Trace Aided Routing
MAC	Medium Access Controller
MANET	Mobile Ad-hoc Network
MAODV	Multicast Ad hoc On-demand Distance Vector
MFR	Most Forward within transmission Radius
MIT	Massachusetts Institute for Technology
MOSPF	Multicast Open Shortest Path First
MPR	Multi-Point Relays
MRL	Message Retransmission List
MRU	Multicast Routing Update
MDSDV	Multicast Destination-Sequenced Distance Vector
MULTICS	Multiplexed Operating and Computing System
NAM	Network Animator
NAT	Network Address Translation
NAV	Network Allocation Vector
NEST	Network Simulation Test-bed
NIC	Network Interface Card
NMEA	National Marine Electronics Association
NRC	National Research Council
NS	Network Simulator
ODMRP	On-Demand Multicast Routing Protocol
ODN	Open Data Network
OS	Operating System
OSI	Open System Interconnect
OSPF	Open Shortest Path First
OTcl	Object-orientated Terminal Command Language
PARC	Palo Alto Research Center
PC	Personal Computer
PCMCIA	Personal Computer Memory Card International Association
PCS	Positional Communication System
PDA	Personal Digital Assistants
PHY IF	Physical Interface
PIM-DM	Protocol Independent Multicast – Dense Mode

PIM-SM	Protocol Independent Multicast – Sparse Mode
QoS	Quality of Service
REAL	Realistic And Large
RFC	Request For Comment
RIP	Routing Information Protocol
RISC	Reduced Instruction Set Computer
RNG	Relative Neighborhood Graph
ROM	Read Only Memory
RP	Rendezvous Point
RPF	Reverse Path Forwarding
RSA	Route Selection Algorithm
RTOS	Real-Time Operating System
RTS	Request To Send
SBA	Scalable Broadcast Algorithm
SDRAM	Synchronous Dynamic Random Access Memory
SMP	Symmetric Multi-Processors
SNAT	Source Network Address Translation
SPA	Self-Positioning Algorithm
SPT	Shortest-Path Tree
SRI	Stanford Research Institute
SRP	Static Routing Protocol
SSA	Signal Stability-based Adaptive routing
TCP	Transmission Control Protocol
TFT	Thin Film Transistor
TORA	Temporary Ordered Routing Algorithm
TSA	Tree Selection Algorithm
TTL	Time-To-Live
UCB	University of California at Berkeley
UCLA	University of California Los Angeles
UDP	User Datagram Protocol
UNICS	Uniplexed Information and Computing System
USB	Universal Serial Bus
USC	University of Southern California
VINT	Virtual InterNetwork Testbed
WAN	Wide Area Network
WRP	Wireless Routing Protocol
ZRP	Zone Routing Protocol

List of Tables

List of Figures

Conventions Used

The following conventions were used within this dissertation.

Italics	Iitalics indicates a specific definition or connotation. For instance, a *group* is defined to be a multicast group, in which a single IP address is used to represent a particular set of recipients.
CAPITAL	Capital letters indicate an abbreviation or a name taken from a particular reference.
`FUNCTION`	A `Courier New` font represents a function call within the Linux kernel.
<field>	Pointed braces signify a particular field name, from a given table.
( )	Round braces signify additional text or information.
[ ]	Square braces indicate a specific reference.

Introduction

1.1 The Characteristics of an Ad-hoc Network

An ad-hoc network is a set of mobile devices[1] that are able to communicate without the need for base-stations (or any other pre-deployed infrastructure). As each node is fitted with a packet radio, ad-hoc systems are able to form active wireless peer-to-peer connections (Figure 1) to any host that falls within their transmission range and hence do not suffer from a single point of failure.

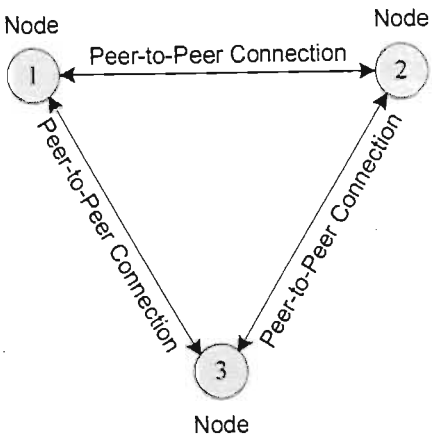


Figure 1: Showing an ad-hoc network consisting of three neighbouring nodes.

But, since packet radios have a limited transmission range, nodes are unable to reach any other node that is not an immediate neighbour. Therefore, to communicate to nodes outside this range, a host requires a neighbouring node to forward data on its' behalf, until the intended host is reached (as illustrated in Figure 2). This process is known as multi-hopping and forms the basis of ad-hoc routing.

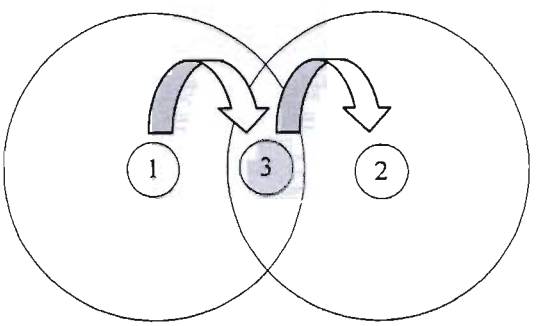


Figure 2: Showing the situation in which nodes 1 & 2 are out of transmission range of each other, but are able to communicate via the common neighbouring node 3.

In addition, mobile users are free to roam dynamically. Since this permits random time changing topologies to occur, each peer-to-peer connection is required to be altered "on the fly". Thus, the multi-hop sequence used by a host needs to be adaptive. However, although neighbouring nodes may adjust their peer-to-peer connections dynamically, this information needs to be distributed to all nodes within the network, or else these nodes will "view" the current network topology differently from others further away. As this will lead to routing errors, protocols for ad-hoc

[1] Node that the words "mobile device", "host" and "node" are used interchangeably to mean a wireless handheld device.

networks need to employ multi-hop schemes that are both adaptive and distributed [1], so that each node may act as a router for each of its neighbours, providing a large degree of network connectivity.

Nevertheless, there are other factors that bear consideration in an ad-hoc network. One of these is the variable capacity of the wireless links. Although links are specified with a fixed upper transmission rate, their actual throughput is affected by the number of nodes that surround it. This is because each neighbouring node is required to share the same channel[2], which allows the possibility that a node will receive a transmission from two different neighbours, simultaneously, causing a packet collision to occur. Thus, to minimize this likelihood, transmitters delay the initial transmission of a packet for a random amount of time, resulting in periods where no data is communicated. This results from what is known as the *multi-access problem,* and causes wireless links to attain capacities that are significantly lower than their wired counterparts.

Other factors include energy or power. While wired systems rely on continuous, non-exhaustive power supplies, wireless systems are generally equipped with batteries for energy consumption. Therefore, protocols are required to be energy efficient in addition to the other characteristics mentioned above. Since this causes a multi-variable problem to occur, routing strategies are non-trivial.

To complicate things further, this dissertation focuses on the problem of sending application data to many destinations simultaneously. While numerous protocols have been successfully developed to send data from one source node to one destination, few have managed to send data to multiple destinations (known as multicasting) efficiently. This arises from the fact the NIC (Network Interface Card) will only process packets that match its unique MAC (Medium Access Controller) address or an address known as the MAC broadcast address. Thus, for a selected set of neighbouring nodes to receive a packet, the underlying MAC protocol would need to interact with the appropriate routing scheme to determine the required MAC addresses for each successive hop. As such an interaction has not been implemented in current commercially available wireless products (at the time of this writing), only two options exist for a routing protocol developer:

1. Data can be addressed to a single neighbouring node (known as unicasting), or
2. Data can be sent to all nodes that fall within a host's transmission range (known as broadcasting).

Since both of these strategies do not require the MAC layer to individually address multiple next-hop destinations, the MAC has no way of determining whether multiple (more than one) hosts have received a particular transmission correctly. Thus, for multicast and broadcast data sequences, acknowledgement procedures are ignored by the MAC layer, rendering it unreliable, unless governed by the underlying routing scheme.

Therefore, the aim of this dissertation is to develop a wireless routing protocol that can send data to multiple destinations efficiently and yet meet all the above mentioned characteristics of an ad-hoc network.

---

[2] A channel is a term used to describe the propagation of radio waves through the air medium, and hence may be subjected to fading, noise and multi-user interference.

1.2 The Positional Communication System (PCS)

Before presenting details of existing ad-hoc protocols, mention should be made of its' application. The idea behind this project is to use mobile devices for ground cover personnel, so that each is equipped with the latest audio and visual information of members within their cell. To this end, sixteen wireless nodes were coupled together over an ad-hoc network, of which one node was fitted with a tactical radio (such as illustrated in Figure 3).

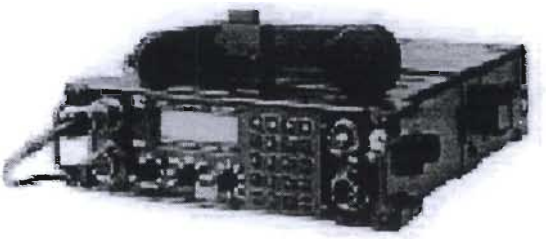


Figure 3: Showing a typical tactical radio.

The tactical radio contains a long-range (low frequency) transmitter that allows data to be received through the military's existing radio network. This permits multiple ad-hoc cells to be connected together, allowing infantry to be closely monitored and coordinated from distant command stations, in order to achieve maximum effectiveness. Note that this dissertation is concerned with the communication aspects of the ad-hoc network only (i.e. the mobile devices contained within the dashed ring of Figure 4).

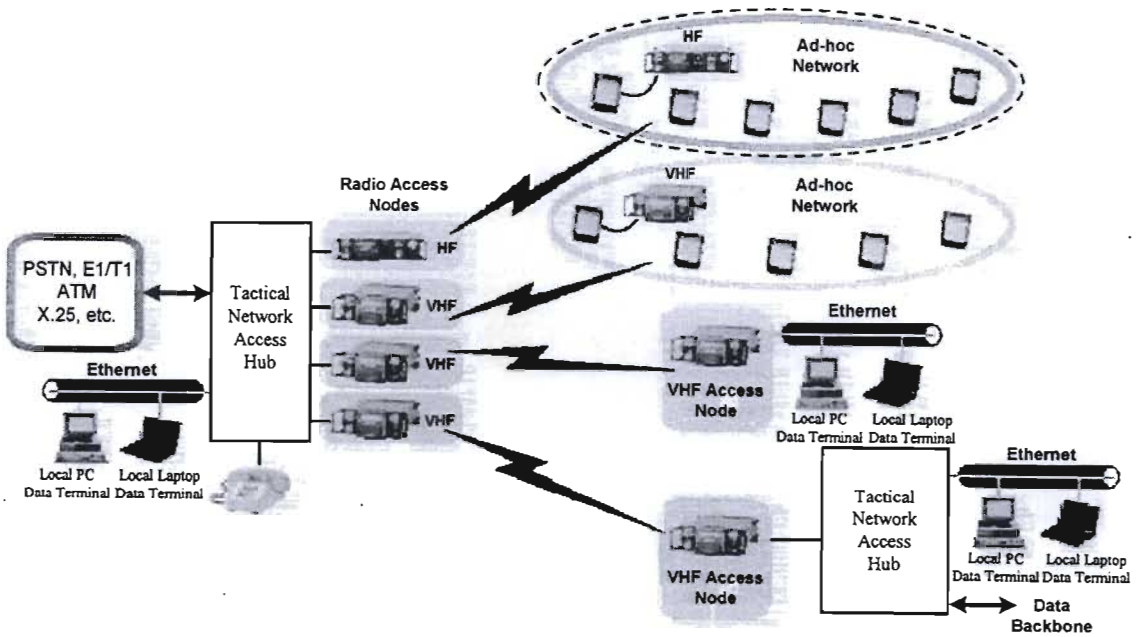


Figure 4: Showing how an ad-hoc network can form part of the military's existing tactical network.

This system became known as the Positional Communication System (PCS), since each node in the ad-hoc network was able to view location information of all users within the network. This was accomplished through the use of GPS (Global Positional System) units, which were serially connected to each mobile device via the NMEA (National Marine Electronics Association) standard. Once the location of a user is known, this information is passed to both neighbouring nodes and a GUI (Graphical User Interface). This way, a host is able to view their location relative to others, allowing a "radar" like vision of all surrounding nodes (as depicted in Figure

5, below). In addition, the GUI was fitted with waypoint placement facilities, which permitted missions to be easily coordinated, updated and illustrated, as required.

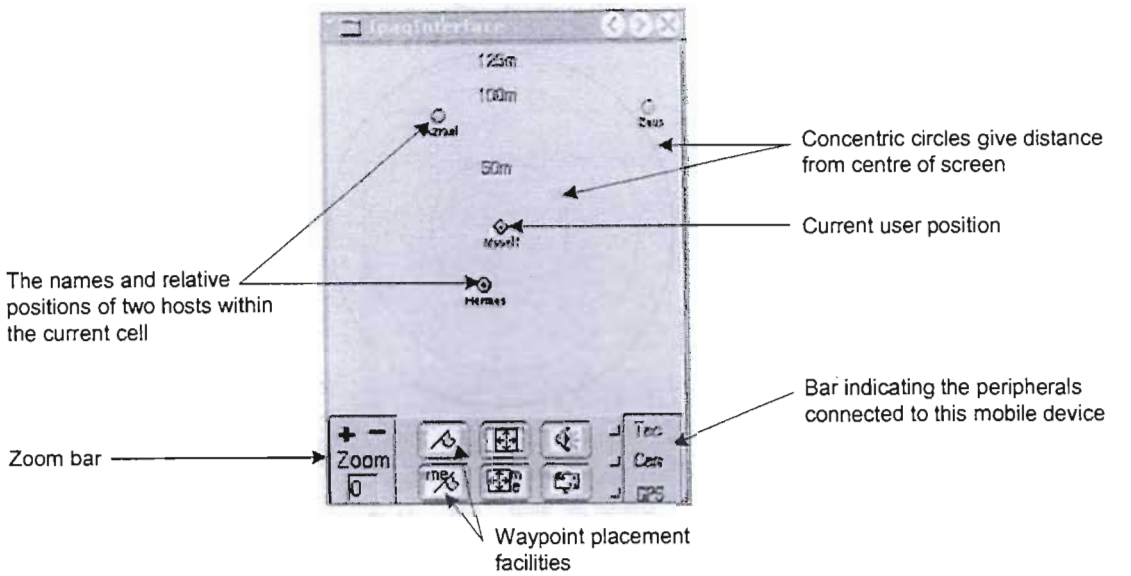


Figure 5: Showing the GUI features available to a typical mobile host.

1.3 Other Applications for Ad-hoc Networks

In addition to military uses, ad-hoc networks may be used for the following:

- Search and rescue missions – where existing wireless infrastructures are disabled, destroyed or have not been deployed yet. An example of this is in cellular systems. Because cell phones rely on base-stations, they become disconnected (and hence inoperable) when these structures fail due to power blackouts or physical damage (or the remote nature of the location). Hence, in such situations where time is critical, ad-hoc networks offer a suitable communication alternative, which can be established with a minimal amount of supporting infrastructure.
- Sensor networks – which scatter a large number of nodes throughout a particular environment for the purpose of gathering information such a pressure, temperature, wind bearing, etc. Since ad-hoc networks allow such systems to be deployed quickly and easily, a great deal of research activity is being conducted to make wireless communication protocols both scalable and energy efficient.
- Rooftop networks – where small radios are being coupled to residential houses, for the purposes of providing Internet services to household PC's. However, for more information in this regard, see [2].
- Vehicular networks – which attach ad-hoc systems to motor vehicles in order to produce applications such as traffic congestion monitoring, inter-vehicle chat and police radar detection [3].
- Private networks – where people may temporarily gather in order to share data via laptop or other wearable devices. Examples of this include seminars, conferences, lectures, tutorials or any other situation in which information is required to be temporarily distributed to a group of people.

1.4 Contributions

In this dissertation, a novel wireless routing protocol is developed that is able to send data to multiple ad-hoc hosts simultaneously. Previous schemes make use of multicast *groups* (defined in Chapter 2), which prevent a source from sending data to a particular set of nodes directly. Although this is an advantage in wired architectures (such as the Internet), it is undesirable in sparse wireless networks where the cost of maintaining *spanning trees* becomes intolerable. This is especially true for handheld devices that contain limited memory, battery and processing power. Thus, instead of maintaining *spanning trees* explicitly, a solution was proposed that made use of an underlying unicast protocol to identify all next-hop hosts for a particular packet. This way, the cost of maintaining the *spanning tree* was lowered to that of the unicast protocol, allowing both (unicast and multicast) schemes to be based on a common algorithm. Since embedded operating systems are becoming more attractive alternatives to dedicated hardware, code commonality is desirable for efficiency.

However, due to the presence of mobility, links through multiple next-hop routers may become temporarily broken (until the unicast protocol is able to refresh each established link). During this period, a localized flood is employed to forward data to all surrounding hosts, in order to find an alternate route around the broken destination path. Results have shown that routing data in this manner causes a greater percentage of packets to be delivered to their intended destination than compared to existing *spanning tree* and *blind-flooding* methodologies.

In addition, this dissertation gives a detailed explanation of the calls that are involved to send (and receive) packets using the Linux 2.4.18 kernel. Previously, documentation only dealt with the 2.0.34 operating system [4]. While there are many websites that provide information with regard to the changes since version 2.0.34, these are geared mainly to one aspect of the kernel and thus were very limited in their descriptions (often leading to many gaps, which could only be resolved through extensive kernel code deciphering). Since such an overview proved valuable to implementation of the proposed algorithm, it is expected that this information will be placed in the public domain, so other research groups may benefit from this endeavor. However, giving core details of kernel code is not the function of this dissertation and therefore it has been placed in Appendix A, for reference purposes only.

1.5 Dissertation Outline

Chapter 2 begins this dissertation by providing a survey of the literature that is available for wireless ad-hoc routing. Here currently published protocols are briefly discussed, with the aim of giving the reader a "feel" for the type of solutions that have been developed prior to this writing. To this end, both unicast and multicast protocols are explored separately, with each offering tradeoffs to the multi-variable ad-hoc problem. Thus, in essence, this chapter provides all the foundational work, from which the newly proposed algorithm (detailed in Chapter 5) was built.

Chapter 3 details the physical devices and its' operating system, since this knowledge is crucial to the implementation of the protocol developed in Chapter 5. Chapter 3 starts by defining the layers that reside both above and below that of the routing code, followed by an explanation of how these layers are used to send (and receive) packets over the operating systems' infrastructure. Once this has been established, this chapter ends by showing how packets may be manipulated, so that the newly proposed protocol can be incorporated into the existing OS framework.

Chapter 4 describes the artificial simulation environment. Although it is the physical environment that matters, simulation provides a platform on which ideas may be constructed, evaluated and compared. Thus, it is here where the academic processes to formulate new routing schemes occur. Hence, this chapter aims to give an overview of one such environment, known as *NS-2* (Network Simulator – version 2), which was employed throughout Chapter 5. Here, *NS-2*'s models, languages and limitations are discussed, giving the reader a full understanding of the assumptions and conditions that was adopted to compare various routing schemes together.

Chapter 5 depicts the details of the newly proposed protocol, known as LAMP (Location Aided Multicast Protocol). This chapter forms the core of this dissertation and thus is used to motivate, evaluate and illustrate all aspects of LAMP's inception. In this regard, four case studies are undertaken to compare LAMP's multicast and unicast capabilities against other leading schemes. Thus, from these results one is able to observe both the advantages and disadvantages of LAMP, and hence evaluate its suitability to the PCS.

Chapter 6 provides an implementation status of LAMP. In this regard, the interactions of the developed modules are described, followed by details of each validation test performed.

Finally, Chapter 7 concludes by highlighting the main aspects of each proceeding chapter, after which directions are given towards future work.

A Survey of Routing Protocols for Ad-hoc Networks

2.1 Introduction

Routing is the process of determining a communication path for a data packet from a source node to a destination node. This may be done either explicitly or implicitly, depending on the scheme employed. In addition, data may not always be intended for a single destination, but rather multiple destinations. Hence, the field of routing has been subdivided into two main protocol methodologies:

1. Unicast – where data is forwarded to single source-destination pairs, and
2. Multicast – where data is delivered to multiple destinations.

This chapter explores both, but primarily focuses on those schemes that have been specifically developed to deal with the problems that occur in wireless ad-hoc communication environments. Note that the protocols presented herein were selected from those methodologies that were commonly found amongst the literature surveyed and hence represent the basis of all known schemes. Thus, it was from this foundation that the construction of the novel protocol of Chapter 5 could take place.

2.2 Unicast Protocols

Unicast routing provides a means for data to be transmitted from a source node to a single destination. On traditional wired interfaces, this has been achieved through the use of distance vector and link-state algorithms, such as Bellman-Ford [5, 6] and Dijkstra [6]. However, these protocols were developed in an environment that is conceptually different to that of ad-hoc networks, where topology changes occur often and resources are limited. Studies [5, 7] have shown that traditional wired protocols produce a considerable amount of overhead during protocol update messaging and are unable to converge to a route rapidly during link changes, motivating the need for more efficient routing approaches.

Literature[5, 8-10] on ad-hoc routing schemes have separated these approaches into two distinct categories, namely those that are location based and those are topology based. Section 2.2.1 details *location based* protocols, while section 2.2.2 looks at *topology based* schemes.

2.2.1 Location Based Protocols

Location based protocols make use of positional information to aid routing decisions. This information may be obtained from GPS [11], SPA (Self-Positioning Algorithm) [12], triangulation or proximity methods [13]. However, in addition to acquiring ones' position, a technique is needed to periodically distribute this information to neighbouring nodes. This process is known as a location *service* and may vary, depending on the amount of nodes selected to store this positional information and the amount of nodes to which they each store [10]. For instance:

- Some nodes may be used to maintain positional information of only a few other nodes [14],
- Some nodes may be used to maintain positional information of all other nodes (not employed),
- All nodes may be used to maintain positional information of only a few other nodes [15],
- All nodes may be used to maintain positional information of all other nodes [16].

The choice of *service* depends on the amount of overhead that can be tolerated and the speed (and accuracy) at which this information can be obtained. For instance, the first and third categories save storage overhead, since the position of only a few nodes are maintained.

However, when a packet is intended for a node that does not form part of this maintained group, an additional scheme is required to retrieve location information for this unknown host, often resulting in larger transmission overheads and even packet delays. On the other hand, the last category maintains the position of all nodes, resulting in a larger initial overhead. But, since the position of all nodes is known, no additional scheme is required, causing a saving in transmission overhead and delay.

Nevertheless, once a location *service* has been established, routing is performed purely from the position of the required destination and location of forwarding neighbours. Since this negates the need for link management procedures (as found in *topology based* schemes), routing tables are not required, making these protocol semi-stateless. Furthermore, *location based* schemes have the natural ability to forward packets all nodes located within a certain geographical area, known as *geocasting* (see section 2.3.1.2).

In this dissertation, two main types of *location based* protocols are discussed. These schemes differ in the way in which packets are forwarded and are classified as either *greedy forwarding* or *direction limited flooding* methodologies, as indicated in Figure 6.

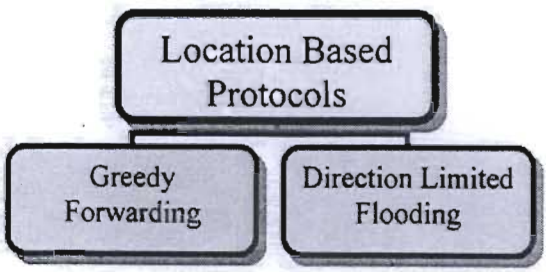


Figure 6: Showing the classification of *location based* schemes.

2.2.1.1 Greedy Forwarding Protocols

Greedy forwarding protocols send data to the neighbouring node that is the closest to the destination, with respect its location. Thus, greedy protocols conceptually forward data over the shortest hop path, without the need for routing tables. However, as will be shown, greedy forwarding techniques do not always ensure a path will be found to the destination node, even when such a path exists. A comparative study of these schemes may be found in [10].

The *greedy forwarding* protocols discussed in this section are depicted in Figure 7.

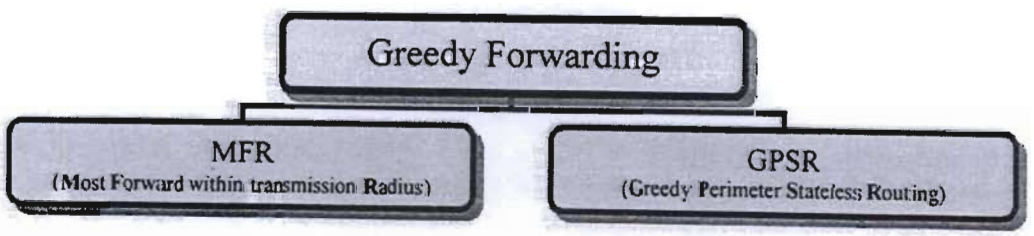


Figure 7: Showing the *greedy-forwarding* protocols described within this sub-section.

2.2.1.1.1 MFR (Most Forward within transmission Radius)

MFR, described in [17], is a near stateless protocol, since data is routed on a per-packet basis, through the use of next-hop locations only.

The MFR protocol assumes that each node is aware of the locations of all nodes that are within its' transmission range and the location of the intended destination, which will be gathered by some location *service*. Routing is then preformed in a greedy fashion, in which a packet is forwarded to the closest neighbouring node (to the destination), after each successive hop.

However, this strategy fails when a *local maximum* is experienced, in which all neighbouring nodes are deemed further (to the destination) than the transmitting node. An example of this is depicted in Figure 8, in which the bottom two nodes are deemed further from the destination (with respect to the *local maximum* node); causing the packet to be dropped when in fact a solution to the destination node should have been found.

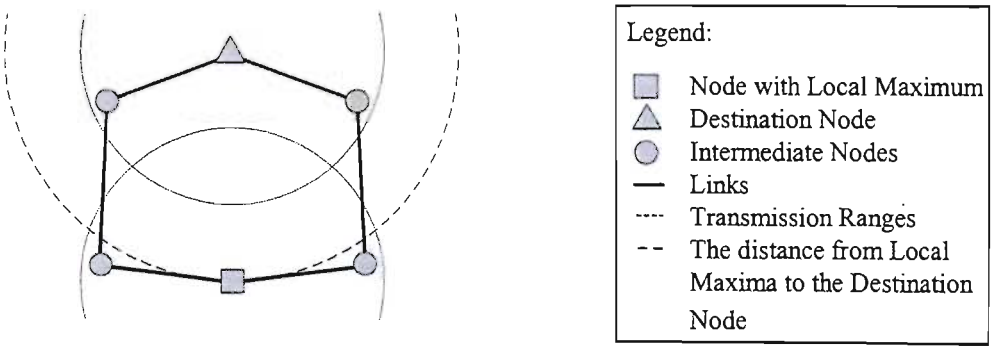


Figure 8: Showing a situation in which the *greedy forwarding* methodology fails.

To limit this problem, MFR modifies its algorithm to allow the least backward node to be selected, if no forward nodes exist, causing the next closest node to the destination (other than itself) to route the packet. However, this may result in routing loops being formed [10], in addition to *local maxima*. Nevertheless, compared to the previous strategy, the modification does allow further solutions to be found, one of which being Figure 8.

Other similar forwarding schemes have been proposed [18-20], but since all these schemes transmit a packet in the forward direction, with respect to the destination's position, they all suffer from the same *local maximum* dilemma.

2.2.1.1.2 GPSR (Greedy Perimeter Stateless Routing)

Like MFR, GPSR [21] is also a stateless routing protocol that forwards packets in a greedy manner. However, unlike MFR, GPSR reverts to *perimeter forwarding* when packets experience *local maxima*, returning to the greedy forwarding strategy once the packet is received by a node which is deemed closer to the destination, with respect to the node that previously contained the *local maximum*.

The *perimeter forwarding* strategy makes use of the right-hand rule to route packets around *local maxima*, and works as follows: A packet is sent along the next link that is found to be counter-clockwise about the transmitting node, with respect to the link pierced by the line formed between the *local maximum* node and the destination. An illustration of this is given in Figure 9. Here the packet is sent along link *e*, since it is the first counter-clockwise link (from the pierced link *c*), when viewed from node 3. Similarly, the packet is forwarded along links *i*

and *j*, when viewed from nodes 4 and 6, respectively. However, before the right-hand rule can be applied, the packet is required to be sent along the first counter-clockwise link (link *b*), with respect to the direction formed by the *local maximum* node and the destination.

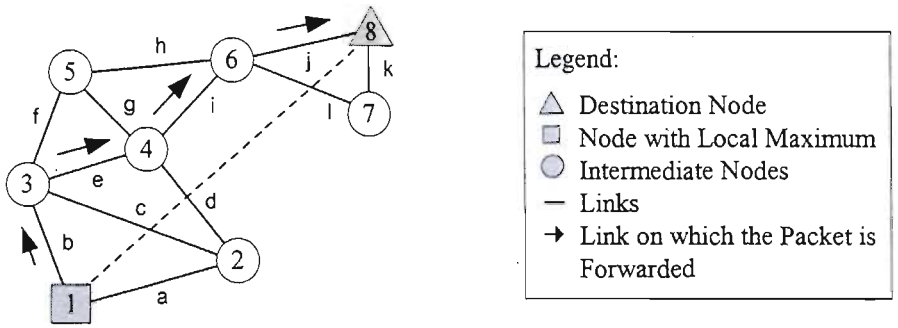


Figure 9: Showing the traversal of a packet using the perimeter forwarding methodology.

Nevertheless, for the right-hand rule to function correctly, no links are allowed to cross each other within the region under consideration [21]. But, since ad-hoc network configurations generally do contain intersecting links, heuristics are required to overcome these obstructions. Two such heuristics are the RNG (Relative Neighborhood Graph) [22] and GG (Gabriel Graph) [23] algorithms. The GPSR protocol may make use of either algorithm, although the RNG is preferred, since less intersecting links are produced. However, both algorithms require additional criteria to ensure that the resultant graph is not partitioned in any way.

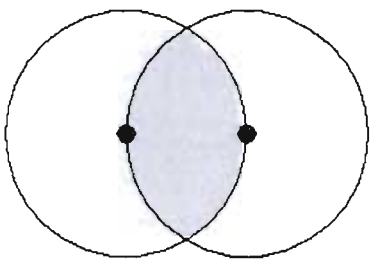


Figure 10: Showing the RNG criteria.

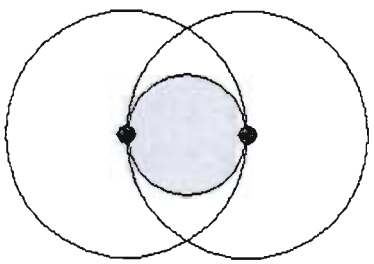


Figure 11: Showing the GG criteria.

The RNG formula works as follows: A link only exists between two nodes A and B, if no other nodes are found within the region formed by the intersection of two circles with radii equal to the distance between A and B (the gray region of Figure 10).

It has been shown from literature [22] that the RNG algorithm is a sub-set of the GG algorithm and thus is less strict in its approach. It works as follows: A link only exists between to nodes A and B, if no other nodes are found within the region formed by a circle with a diameter equal to the distance between A and B (the gray region of Figure 11).

In static topologies, GPSR has been shown [21] to achieve 100% packet delivery success, when a path to the destination exists. However, since an up-to-date location *service* cannot be accomplished when random node movement is present, uncertainties in the position of neighbouring nodes cause GPSR to achieve a packet delivery success rate of over 97%.

2.2.1.2 Direction Limited Flooding Protocols

As its name implies, *direction limited flooding* protocols flood data to destination nodes. However, these schemes try to reduce the amount of overhead produced by restricting the number of forwarding nodes, where the restriction criterion is based on the direction of neighbours. A comparative study of these schemes may be found in [24].

The *direction limited flooding* protocols discussed in this section are indicated in Figure 12.

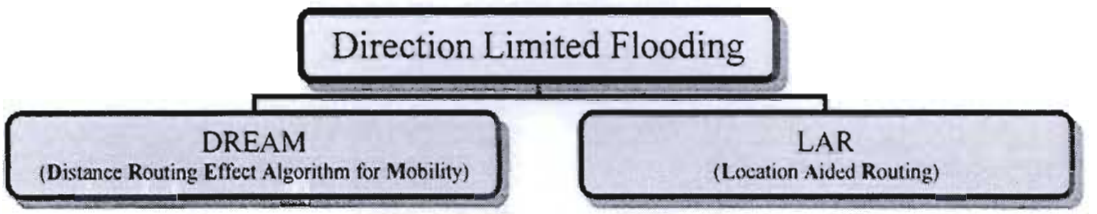


Figure 12: Showing the *direction limited flooding* protocols described within this sub-section.

2.2.1.2.1 DREAM (Distance Routing Effect Algorithm for Mobility)

DREAM, given in [16], portrays a protocol that can be used when the mobility rate of nodes is high, rendering *topology-based* schemes inappropriate.

The DREAM protocol explicitly defines the way its' location *service* should be implemented. It uses a novel technique, which regularly floods location updates according to both the mobility rate of a node and what is termed the *distance effect*.

The *distance effect* is described in [16] as follows: "The greater the distance separating two nodes, the slower they appear to be moving with respect to each other". Thus nodes further away, require location updates less frequently than closer nodes. This is known as the location *service*'s spatial resolution. The same effect was noted in FSR (described later in section 2.2.2.1.5).

However, the temporal resolution of the location *service* can also be adjusted. This becomes apparent when one considers the mobility rate of nodes. The faster a node moves, the more frequent the node is required to transmit its position. Since this frequency is coupled to the nodes' periodic update interval, the faster a node moves, the shorter it sets its' location update period. Thus, positional information is only conveyed when needed, resulting in better use of the location *service*.

DREAM's packet forwarding strategy works as follows: When a node (A) wishes to send data (D) to a particular destination node (B), it will acquire the position of B (through the location *service* described previously) and forward the packet to all neighbouring nodes that are deemed to be in the direction of B. Since the position of nodes is not accurate, this direction is determined by drawing two tangential lines from node A to the circle formed by what is termed the *expected zone* (described below). An illustration of this is given in Figure 13. If no nodes are found within the required direction, a recovery procedure has to be initiated. However, the details of this mechanism are not described within the DREAM protocol specification. Once neighbouring nodes receive D, they re-transmit it to other neighbouring nodes (deemed to be within the direction of B), until D is received by B.

The *expected zone* is the region within which the destination node is expected to be contained, given certain additional information, such as its' average speed, its' bearing and the time since the location of the destination node was last known. Thus, generally, this zone is a circle

centered at the destination (see Figure 13), with a radius equal to the time elapsed since the last update multiplied by the destinations' average speed.

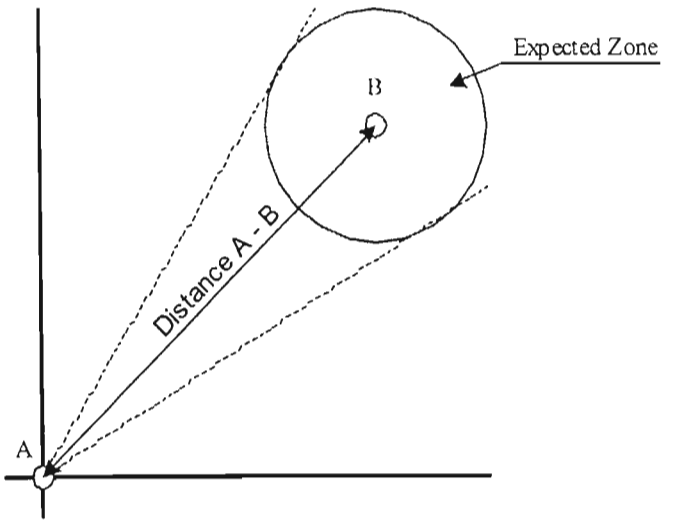


Figure 13: Showing how "direction" is determined in DREAM.

DREAM results in a packet delivery ratio of 80% and above, depending on the amount of congestion present [16]. Since DREAM is also classified as a flooding technique, it offers 25%-250% less routing latencies compared to reactive protocols, such as AODV and DSR (detailed in section 2.2.2.2).

2.2.1.2.2 LAR (Location Aided Routing)

LAR, explained in [25], provides a way of limiting the search area when seeking a route from source to destination. Instead of flooding the whole network for a possible route, the source sends a route request to its neighbours, which is only forwarded if its' neighbour is positioned in what is termed a *request zone*.

Request zones are zones that include areas over-and-above the destinations' *expected zone*. One would think that the *expected zone* and the *requested zone* would be the same, but due to uncertainties and the likelihood that there may be no nodes contained within the *expected zone* (including the source node), the *request zone* is usually made larger than the *expected zone*. This is shown in Figure 14.

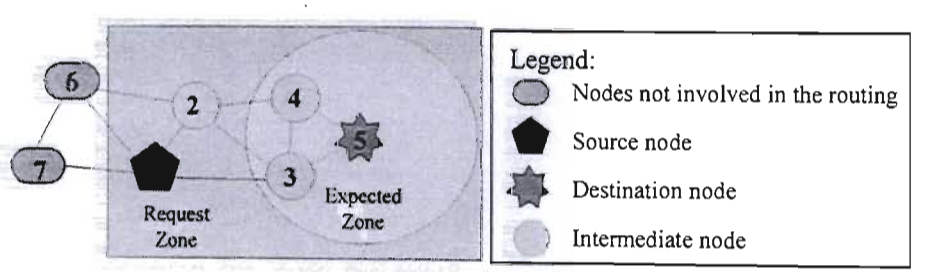


Figure 14: Showing the *expected* and *request zones*, using the first algorithm of LAR.

LAR proposes two different algorithms to determine the *request zone*. The first of these is the smallest rectangle that encloses both the source node and the *expected zone* (as depicted in Figure 14), while the second does not rely on geometry, but uses the following scheme: A node

receiving data from its neighbour only forwards the packet if its' location is deemed closer (or equal) to the destination than the previous node, or else the packet is dropped.

However, both these algorithms contain undesirable properties. The first algorithm suffers from the fact that there is no guarantee that a path will be found, which consists solely of the nodes that are contained within the defined rectangle [25]. This means that if a path is not found within a specified time interval, the source node will be required to issue a new route request that contains an expanded *request zone*, until a path is found (assuming that the destination node is not partitioned from the network in some way). This of course leads to larger routing latencies and message overhead.

The same problem exists for the second algorithm, since the nodes closer to the destination may not always be the nodes that contain a route to that destination (see Figure 8). However, this can be resolved if one keeps track of both time and distance metrics, so that a node can also forward data if it has a better global time to the destination, since this is similar to saying that a node must route data along the shortest hop path [26].

2.2.2 Topology based Protocols

Topology based protocols make use of link information to transfer data. This category can be divided further into *table-driven* (proactive), *source-initiated* (reactive) and *hybrid* schemes. A diagram showing this division is given in Figure 15, below.

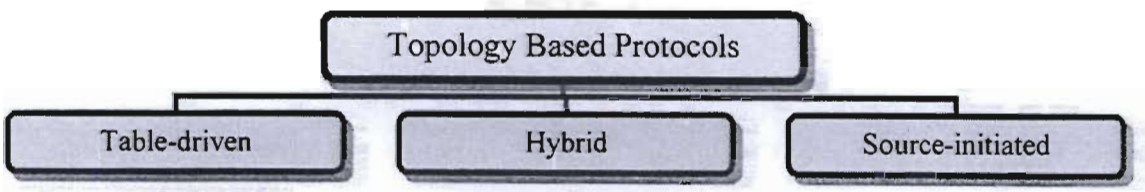
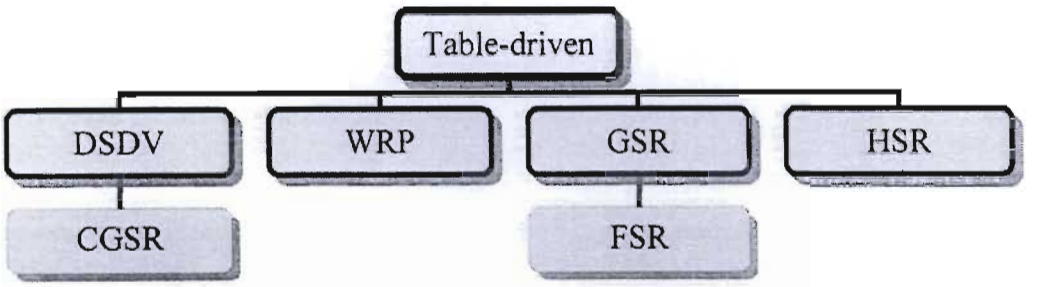


Figure 15: Showing the classification of *topology-based* routing schemes.

2.2.2.1 Table-Driven (Proactive) Routing Protocols

Table-driven protocols are protocols that require a node to maintain up-to-date routing knowledge (tables), concerning the link status of other nodes within the network. This causes *table-driven* protocols to contain very little delay with regards to route searches, since next-hop information is readily available from within the nodes' locally established routing table.

However, in order to establish up-to-date tables, periodic update messaging is required. Since the speed of the node usually corresponds to the rate that this information is required to be maintained, the period update interval may need to be altered inversely proportional to speed (such as done in DREAM) or as soon as a link failure is detected (or both). Hence, proactive schemes suffer from large overheads, which result from continual protocol updates, causing scarce resources, such as channel bandwidth and battery power to consumed. The proactive routing protocols discussed in this section are indicated in Figure 16.



DSDV (Destination-Sequenced Distance-Vector) **GSR** (Global State Routing)
CGSR (Clusterhead-Gateway Switch Routing) **FSR** (Fisheye State Routing)
WRP (Wireless Routing Protocol) **HSR** (Hierarchal State Routing)

Figure 16: Showing the *table-driven* protocols described within this sub-section.

2.2.2.1.1 DSDV (Destination-Sequenced Distance-Vector)

The DSDV protocol described in [27] is a modified version the distributed Bellman-Ford Algorithm [5, 6] that was used successfully in many dynamic packet switched networks [9].

The Bellman-Ford method provided a means of calculating the shortest paths from any source to any destination pair, given that the metrics (distance-vectors) to each link are known. However, it is subject to routing loops [5, 8]. Routing loops are a phenomenon associated with networks whereby packets are forwarded in a circular motion throughout the network, without ever reaching the destination, due to insufficient routing information being provided to nodes when links fail. DSDV overcame this problem by including *destination-sequenced* numbers.

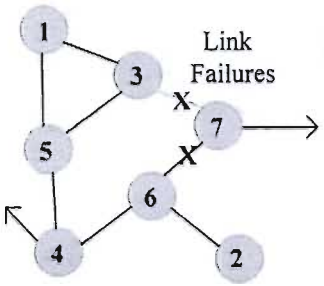
In DSDV, each node is required to transmit a local sequence number, which is periodically increased by two and transmitted along with any other routing update messaging to all neighbouring nodes. On reception of these update messages, the neighbouring nodes use the following algorithm to determine whether to ignore the update or whether to make the necessary changes to its routing table:

If the sequence number of the received update message entry is greater than that currently present within that nodes' locally established routing table,
Then replace that routing table entry with the received update entry,
Else
  If the sequence numbers match, but the update message entry contains a lower (better) metric (hop count) than that found within the routing table,
  Then replace that routing table entry with the received update entry,
  Else ignore this update entry.
Move to the next entry within the update message, until all entries have been examined.

If the algorithm above results in any change to the routing table of a node, then these changes are considered to be update message entries relevant for neighbouring nodes and therefore will be transmitted on the nodes next periodic update interval, resulting in a propagation of the routing information throughout the network.

Routing Table of node 4:

Dest. Node	Next hop	# of hops	Seq. #
1	5	2	12
2	6	2	36
3	5	2	56
4	4	0	28
5	5	1	82
6	6	1	52
7	5	∞	45



Routing Table of node 7:

Dest. Node	Next hop	# of hops	Seq. #
1	3	∞	11
2	3	∞	37
3	3	∞	57
4	3	∞	27
5	3	∞	83
6	3	∞	53
7	7	0	44

Figure 17: Showing how broken links are interpreted in the routing table of the DSDV protocol.

However, if any node detects a link failure while trying to send data to a particular destination, then that node is required to increase the sequence number of that destination by one and immediately transmit this information to neighbouring nodes (known as a triggered update). Since the destination is required to send even numbered sequence numbers (due to increasing its' sequence number by two each time), odd numbered sequence numbers are interpreted as being unattainable destinations. This is clearly shown in the last two columns of node 7's routing table, given in Figure 17, where infinitely symbols are used to indicate broken links.

Since destination nodes are able to increase their sequence number by two, broken links are quickly superseded by the particular destination concerned, resulting in consistent routing tables throughout the network.

2.2.2.1.2 CGSR (Clusterhead-Gateway Switch Routing)

CGSR [28] uses a hierarchical routing scheme [29] that closely models the concepts of the existing Internet infrastructure.

Hierarchical techniques require the use of a cluster head selection algorithm [30-33] to nominate *clusterheads*, through which all data of that cluster is routed. Once the *clusterheads* have been nominated, *gateway* nodes are then selected on the basis that they join two *clusterheads* together, as shown in Figure 18.

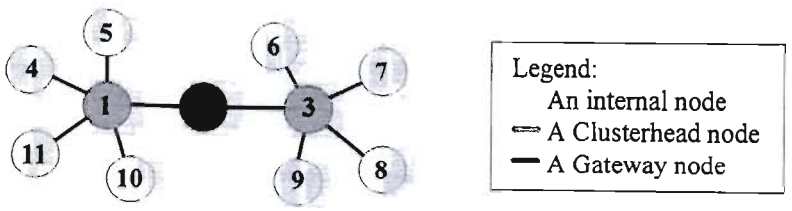


Figure 18: Showing the hierarchical definitions of CGSR.

Since the *clusterhead* selection process results in additional overhead [34], it is only instigated when absolutely necessary. In CGSR, this occurs when a node moves out of range of any of the currently selected *clusterheads* or when two *clusterheads* become neighbours. Once the node hierarchy has been established, packets may be routed according to an underlying unicasting scheme, which is modified in some way to send data via *clusterheads* and *gateways* instead of through other next-hop neighbours. This results in the underlying protocol becoming scalable, since less state information is required to be stored at each node. To this end, DSDV is employed as CGSR's underlying routing scheme.

2.2.2.1.3 WRP (Wireless Routing Protocol)

WRP, developed in [35], is a modified version of the Dijkstra Algorithm described in [6], except that it makes use of four tables to route the data with loop freedom [8], resulting in large memory consumptions.

The first of these is a *link cost* table that keeps track of the metrics (cost) of each established link, where a metric may give an indication of the delay, hop-distance or congestion (or some complex combination) of the link from one node to another. This table is then used to calculate the cheapest cost to each node, forming a *distance* table. The combination of these two tables allows a spanning tree of the network to be produced (shown in Figure 19), from which a *routing* table can be constructed. Thus, WRP uses its *link cost* and *distance* tables to establish a full topological map of the network, before establishing a routing path, allowing multiple paths to a destination to be remembered. This is conceptually different from DSDV, where the routing table is established without knowledge of the topology of the network, allowing single next-hop routes to be known only.

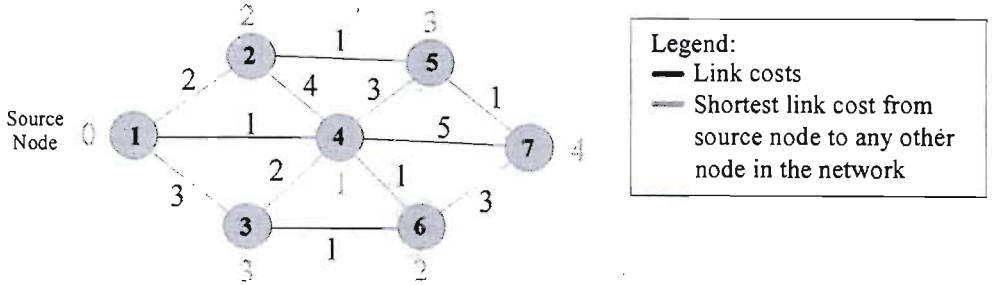


Figure 19: Showing graphically the WRP information that would be stored in the *link cost* and *distance* tables of node 1 (the source node).

In addition to using the three tables mentioned above, WRP makes use of *message retransmission list* (MRL) to determine which update messages should be retransmitted and which neighbouring nodes should acknowledge the retransmission. Depending on the outcome of the MRL, *link costs* can be altered, allowing the *distance* and *routing* tables to be maintained.

In WRP, nodes learn about the existence of neighbouring links by listening in on transmissions. Hence, any node not involved in the transmission of data is expected to send out *hello* packets to ensure connectivity.

2.2.2.1.4 GSR (Global State Routing)

GSR [29, 36] is similar to WRP, in that is maintains a full topological map of the network at each node, using a link-state routing algorithm. The main difference between the traditional link-state routing algorithms and GSR is that instead of flooding the whole network with topology information when the topology changes, GSR makes use of periodic sequence-numbered updates (similar to DSDV) to keep its link-state tables up-to-date. Like WRP, once the link-state tables are updated, shortest paths can be calculated to all destinations and routing tables constructed.

The main disadvantages of GSR are that the periodic updates cause large latencies in establishing correct routing tables, due to the propagation period of the update messages throughout the network, and that these periodic updates result in large bandwidth consumption [29].

2.2.2.1.5 FSR (Fisheye State Routing)

FSR, described in [29], originates from a graphical technique known as *Fisheye* [37], whereby high pixel detail levels are captured about certain focal points, but which decrease as the distance from the focal point increases. In much the same manner, FSR performs routing, except that instead of pixel detail levels decreasing with increased distance, accurate maintenance of routing information is used instead, with central nodes replacing focal points. Thus, packets that are being transmitted to a particular destination receive more accurate information about where the destination is, as the packets progresses closer and closer to the final destination.

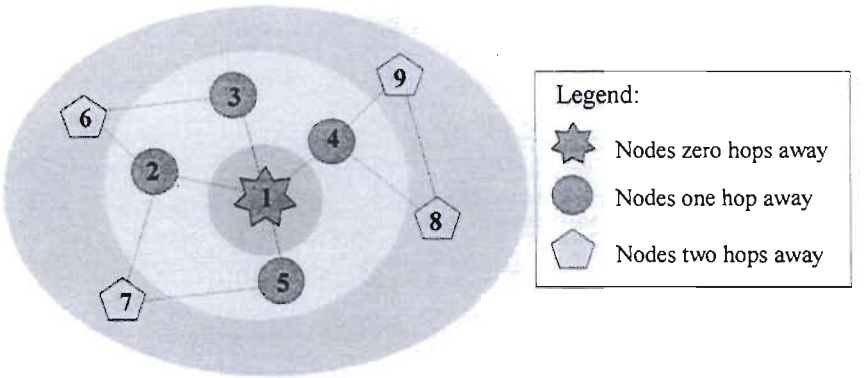


Figure 20: Showing the scope of FSR, with node 1 as the source node

This technique, shown in Figure 20 above, is achieved by varying the update messaging periods of a particular node according to the number of hops the other nodes of the network are away from it. For example, neighbouring nodes are informed of changes more often than nodes further away. Hence, more bandwidth can be utilized for the routing of actual useful data instead of being consumed by routing updates, since FSR uses GSR as it underlying routing protocol.

2.2.2.1.6 HSR (Hierarchal State Routing)

HSR, also described in [29], is based largely on the concepts of IP [38] and was developed to overcome some of the drawbacks of CGSR, since CGSR uses *physical partitioning* (partitioning based on location or position) only, while HSR uses both physical and *logical partitioning* (partitioning based on certain properties, for example members of the same company, family or cell).

HSR uses hierarchical link states, as opposed to distance vectors, to route data and hence address a node from the upper most level to the bottom most level, where each level is determined through an iterative use of one of the cluster head selection algorithms of CGSR. This process is shown in Figure 21, over the page. Note that selected cluster heads of the previous iteration are used for the next level of abstraction, until only one cluster head (the upper most level) remains.

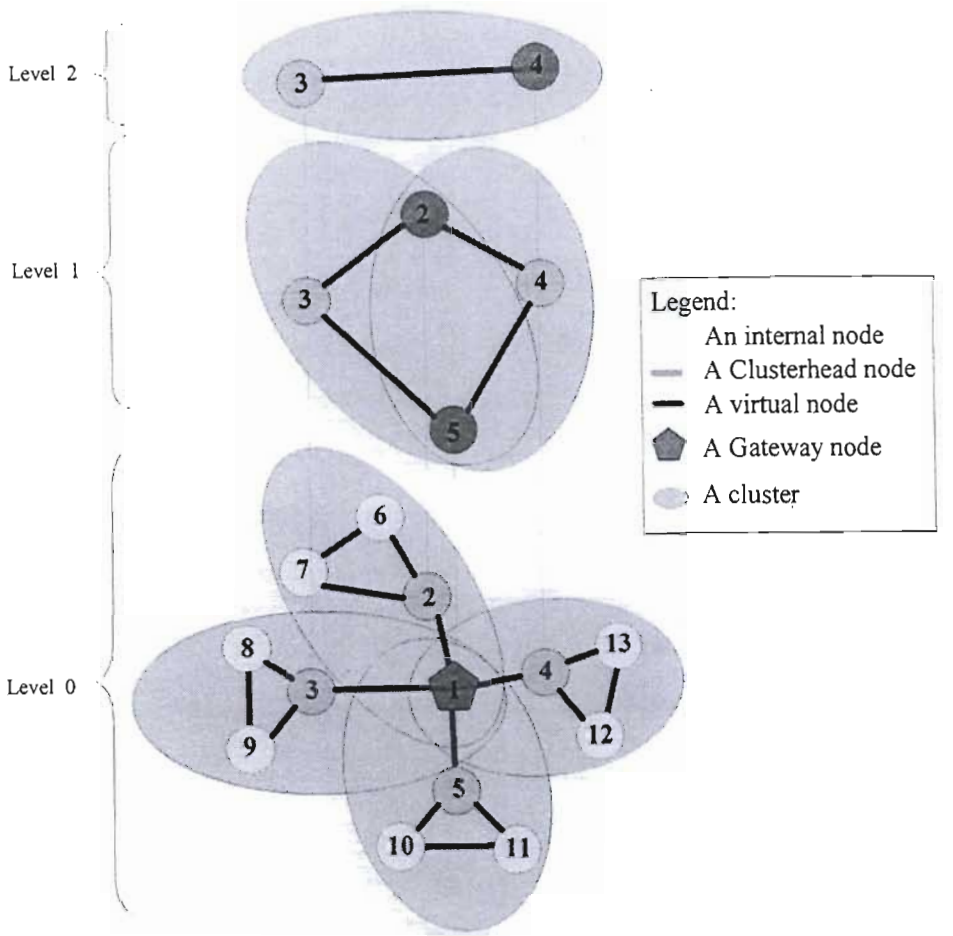


Figure 21: Showing how HSR uses the Clusterhead selection algorithm to form a hierarchical (physical) partitioning scheme.

The beauty of HSR is, however, not in its hierarchical (physical) partitioning scheme, but rather in its adoption of a *logical partitioning* scheme. This is achieved by placing nodes into logical groups, within which a leader is selected, known as the home agent. The duty of home agents is to maintain their group and to report its group (subnet) address to the upper most level. Nodes not chosen as home agents are then required to periodically give the home agent their hierarchical address, which is determined through the *physical partitioning* scheme described previously.

Hence, a node wishing to communicate with another node, need only know that nodes logical address. Once it has obtained this address, it simply applies the groups' subnet (obtained from the upper most level) and forwards the packet to the obtained home agent. The home agent then re-routes this packet to the desired node, through its hierarchical link-state table. Once the source and destination nodes have learnt the hierarchical addresses of one another, they are able to send the required packets directly, by-passing the home agent.

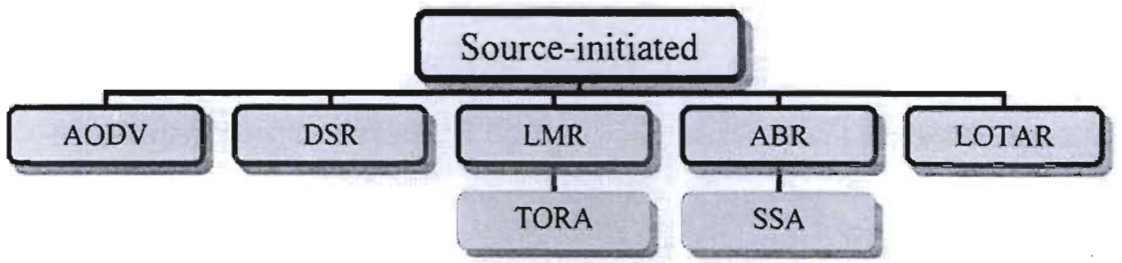
2.2.2.2 Source-Initiated (Reactive) Routing Protocols

Source-initiated protocols were devised to limit the amount of neighbouring information stored at each node. This was achieved through the use of routes, as apposed to next-hop tables. A route (or path) to a destination requires significantly less data storage than up-to-date next-hop tables, since knowledge of next-hop neighbours are only needed when they supply a route. Thus, once a route is no longer required, knowledge of this neighbour may be erased, unless the

neighbouring node is currently supplying another route. Hence, an optimal use of memory is employed, which allows reactive schemes to be used in large networks.

However, in order to establish a route not currently in progress, a *route discovery* mechanism is required. This requires the source node to flood route requests to all network nodes, until the destination replies with a routing path. During this process, routers monitor the request/reply exchange, allowing them to learn about a route for the required transmission. Since the exchange mechanism requires finite time, *source-initiated* algorithms generally suffer from larger network delays compared to proactive schemes. However, since there is no maintenance of nodes not involved in routing, reactive schemes allow nodes to *sleep*, reducing both the overhead and the amount of power consumed.

The reactive routing protocols discussed in this section are indicated in Figure 22.



AODV (**A**d-hoc **O**n-demand **D**istance-**V**ector) **ABR** (**A**ssociativity-**B**ased **R**outing)
DSR (**D**ynamic **S**ource **R**outing) **SSA** (**S**ignal **S**tability-based **A**daptive routing)
LMR (**L**ightweight **M**obile **R**outing) **LOTAR** (**Lo**cation **T**race **A**ided **R**outing)
TORA (**T**emporary **O**rdered **R**outing **A**lgorithm)

Figure 22: Showing the source-initiated protocols described within this sub-section.

2.2.2.2.1 AODV (Ad-hoc On-demand Distance-Vector)

AODV, explained in [39], builds on the DSDV algorithm [8] to make it connection (circuit) orientated and hence requires the establishment of a path from source to destination, before any routing can take place.

AODV establishes a path using a two-stage process. The first stage involves the establishment of a reverse path, whereby the source node, wishing to send data to a particular destination, broadcasts a *route request* to each of its immediate neighbours. These nodes receive the request and make a note of the predecessor node, before re-broadcasting the request on to their neighbours. This process continues, until the destination node receives the request, initiating the second stage of the path discovery. Since each node along the path from source to destination has recorded which node sent it the request, a path from the destination node to the source can be established. The task of the destination node is to simply reply along this path (usually from the first request it receives), so that each node can reverse the direction of the established path, causing a forward path to be constructed from source to destination. Any node that does not receive the reply after a certain amount of time may assume it is not involved in the transfer and thus may discard the path. Since no reply was received from the destination, nodes 2, 3 & 5 discard the path in Figure 23.

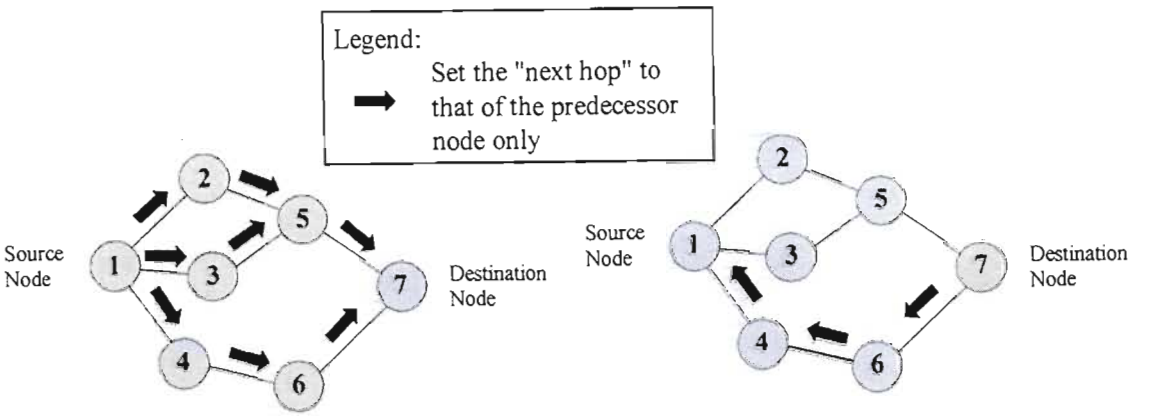


Figure 23: Showing the two stage route discovery mechanism of AODV

When a path becomes invalid, due to movement of nodes, the node that detected the broken link is required to inform the source, which simply erases the old path and searches for a new one. This process is known as *route maintenance*.

2.2.2.2.2 DSR (Dynamic Source Routing)

DSR, given in [40], is similar to AODV, except that it makes use of *route caches* that hold knowledge of all predecessor nodes, as opposed to AODV, which only keeps track of the previous hop. This is shown in Figure 24, below.

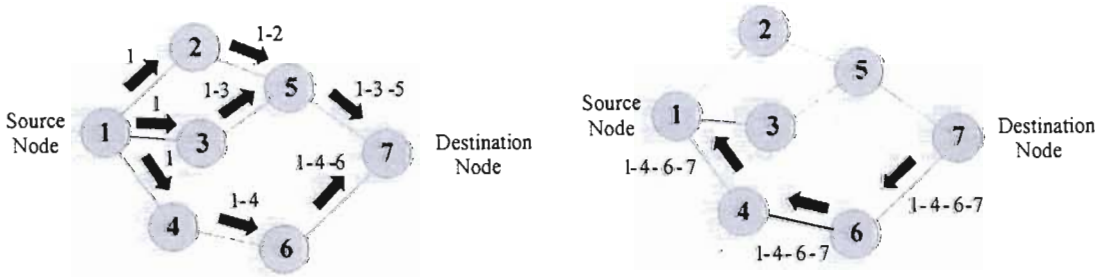


Figure 24: Showing the route discovery mechanism of DSR

Before a node initiates a *route discovery* request, it consults its' cache to determine if an un-expired route exists. If the node finds the route in question, it simply uses this route; else the node reverts to a *route discovery* mechanism, which requires each node to append its address to the request before forwarding it (as shown in Figure 24). Prior to doing so, however, DSR additionally requires each node to consult its cache to determine whether the destination node can be obtained through an existing path. If not, then the intermediate node will simply forward the request on towards its' neighbours; else DSR allows this node to reply back to the source in one of the following manners:

- If the intermediate node contains a route in its cache that leads back to the source, it must use this reply path instead,
- Else, the intermediate node must reply back to the source along the partially established path, constructed during the source request,
- Else, the intermediate node must perform its own *route discovery*, with the destination of the request being that of the source.

If the above process still causes the request to reach the destination node, the first request received is used as the reply path. Note that the core difference between DSR and AODV is that DSR does not make use of a routing table to determine a next-hop neighbour. Instead, DSR

embeds the whole forwarding path in each packet, at the source. This way, routers need only lookup the next-hop from inside the packet. However, if this strategy fails due to a broken link, then DSR consults its cache memory to resolve the forwarding host. A study was conducted by Das et al [41] to compare AODV against DSR. The results of this study showed DSR performing above AODV (in terms of load, packet delivery and route choice) when few nodes were used and the congestion was kept low. However, when these conditions were increased, AODV was shown to perform better than DSR, since DSR was unable to remove redundant entries from its cache memory, causing it to select obsolete routes.

2.2.2.2.3 LMR (Lightweight Mobile Routing)

LMR, described in [42], differs from other *source-initiated* routing protocols in that it assigns values to the links of its neighbours, through a neighbour link-state table. These values can be one of five different possibilities – namely *unassigned*, *upstream*, *downstream*, *downstream-blocked*, *unassigned-waiting* and *awaiting-broadcast*.

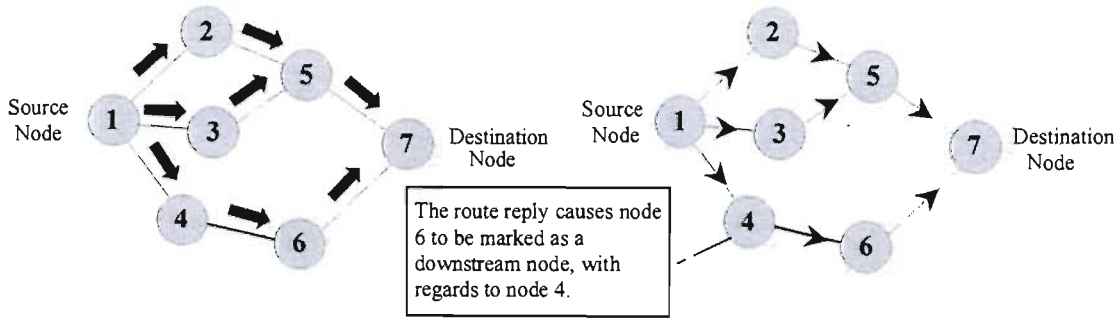


Figure 25: Showing how LMR modifies AODV to bring directivity into its links.

The *route discovery* request of LMR is similar to AODV, except that when the reply is returned from the destination node, all the intermediate nodes are marked as *downstream* links, as shown in Figure 25. This process forms what is known as a *Directed Acyclic Graph* (DAG), causing multiple routes to be established to a destination. This increases reliability, since alternate paths exists when the currently chosen path becomes invalid. The choice of which path to use can be determined through the use of a path cost metric or simply using the first (and thus possibly the shortest) path received by the source.

When a broken link is detected in LMR, the node that detected the broken link is required to do one of two things depending on whether it contains any *upstream* nodes. If it does, it is required to send out a link failure broadcast, which allows neighbouring nodes to mark the link to this node as undirected (*unassigned*) and allow them to look for an alternate route; else the node is required to issue its' own *route request*, in the hope of finding a new path.

To prevent routing loops, link-state values of *downstream-blocked*, *unassigned-waiting* and *awaiting-broadcast* are used in specific situations, which are well documented in [42]. Also, any disconnected node that requires a route to a particular destination is required to periodically broadcast route requests, as in the *route discovery* request phase, until network reconnection occurs.

2.2.2.2.4 TORA (Temporary Ordered Routing Algorithm)

TORA [43] is used in a highly adaptive, dynamic environment and is based on the concepts of link reversal [44] and LMR [42], where nodes keep track of the link states of their neighbours.

During the *route discovery* and *maintenance* phases, TORA establishes a sequence of directed links from the source to the destination node, as given previously in Figure 25. From these link directions, each node is then able to be assigned a logical *height* metric, based on the hop-count from the destination node (the base node). Hence, when data is transferred it flows from the node with the greatest *height* metric (the source) to the node with the lowest (the destination), as shown in Figure 26.

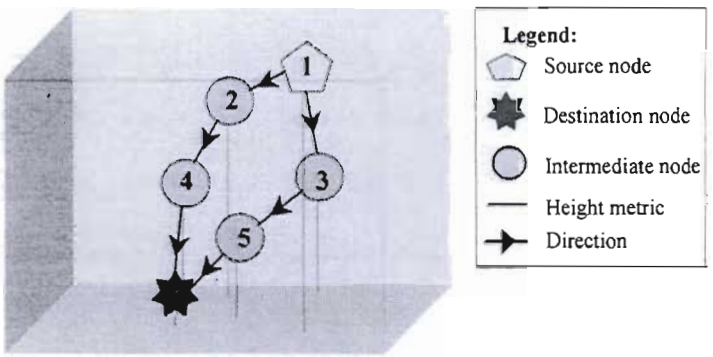


Figure 26: Showing the DAG established in TORA

When a link fails, all the link directions from the broken link to the source are reversed, until that path is completely erased. Since TORA makes use of link-states, multiple paths are established to the destination and so, hopefully, when a particular path is erased, data may still be routed through one of the remaining paths. Given that the *height* metric of a node has a temporal dependency (due to the mobility of nodes), TORA assumes that all nodes have synchronized clocks, through the use of some external source, such as the satellite time from a GPS.

A study was performed by [45] in which DSDV, AODV, DSR and TORA were compared against one another. There, TORA was shown to induce the most amount of routing overhead, due to the establishment of its' DAG. In addition, TORA delivered fewer packets to the intended destination node, since temporary routing loops occurred when the direction (and thus the *height*) of each link was dynamically altered. But, since TORA was able to remember multiple routes, it was able to route data with less routing latency than the other schemes compared, especially during high node mobility conditions.

2.2.2.2.5 ABR (Associativity-Based Routing)

ABR, given in [46], is unique with regards to the metric it employs. Instead of using hop-count, ABR uses *degree of association stability*.

In ABR, each node periodically generates a *beacon* to signify its existence, which, when received by neighbouring nodes, causes an increase in the association entry, with respect to itself and the *beacon* node. Thus, *association stability* is defined to be the connection stability of one node with respect to another, in-terms-of time and space [8]. The association tables reset (with respect to the two nodes in question) when one node moves out of proximity of its

neighbour. Thus, the objective of ABR is to discover stable neighbouring nodes, which will act as routers for the mobile network, ensuring that control messaging can be kept to a minimum.

Route discovery broadcasts in ABR are similar to AODV, except that each intermediate node additionally appends its association table (with respect to its neighbours) to the request. The destination node, on reception of the request, selects the route with the most stability, based on the associations of all the links, and replies back to the source along this chosen path.

Route maintenance involves partial *route discovery*, invalid *route erasure*, valid *route updates* and new *route discovery*, depending on which node moves. If the source node moves, new *route discovery* takes place. If the destination moves, the immediate upstream node erases its route and sends a partial request for the destination. If the destination receives the partial request, it replies with the best partial route, or else the immediate upstream node times-out and the process backtracks to the next upstream node. If the backtracking occurs for more than halve the hop-count of the previous valid route, then the *route discovery* process begins all over again.

Route deletion (or erasure) is used when the route is no longer needed. It involves broadcasting a route delete message to all nodes, since the source may be unaware of any changes that were made to the current valid route, during backtracking.

2.2.2.2.6 SSA (Signal Stability-Based Adaptive Routing)

SSA, explained in [47], selects routes based on signal strength and node stability. SSA is basically the combination of two cooperating protocols – namely DRP (Dynamic Routing Protocol) and SRP (Static Routing Protocol). All transmissions are passed first to DRP, which then passes control to SRP.

DRP is responsible for maintaining the signal strength table and for building the routing table. The signal strength table is built by marking neighbouring links as either *weak* or *strong*, which is obtained periodically from the signal strength of neighbouring *beacons*.

SRP looks at the routing table to determine where to send the received packet next. If the destination is not in the routing table, a *route-search* process is initiated to find the route.

Route discovery requests of SSA, are similar to ABR, except that the request is only propagated through the network if the received link is marked as being *strong*. In SSA, however, the destination node chooses the first *route-search* packet received as the route for the data transfer, since this route is likely to contain the least hop-count and be the less congested. If the reply does not reach the source after a specified amount of time, the source is able change a field (the *PREF* field) in the *route discovery* request, to indicate that any link will do, thus allowing *weak* links to be considered within the *route-search* process. Hence, SSA tries to route the majority of its packets over *strong* links first, before reverting to weaker (less stable) links.

When a link fails in SSA, nodes are required to send an error message back to the source, which clears all links previously associated with the route and initiates a new *route-search*.

2.2.2.2.7 LOTAR (Location Trace Aided Routing)

LOTAR, developed in [26], is based on LAR, except that it provides an efficient way to get and maintain the location information of each node. LOTAR assumes that each node knows its own speed, location and global time, such that these parameters can be periodically transmitted as a location *service* to neighbouring nodes, allowing all nodes within the network to locate each

other. In addition to maintaining location tables, nodes are required to hold next-hop routing tables, but only if they are actively involved in routing. Hence, LOTAR is a reactive protocol that makes uses of positional information to find routes and thus forms a hybrid between *topology* and *location based* schemes.

Route query messaging of LOTAR is similar to the first algorithm proposed in LAR, except that *route requests* are only forwarded if the receiving node falls within the *request zone* or it contains a newer global time to the destination node. In addition, nodes update the *request zone* and the global time properties of the query, before it is forwarded to neighbouring nodes, thus predicting the *expected region* of the destination node more accurately, after each successive hop.

When routes become invalid, instead of initiating another *route discovery* query from source to destination, LOTAR elects a *sponsor* node to find an alternate path. The *sponsor* node is the node that contains the latest global time to the destination node, with respect to the path from source node to the node where the link breakage occurred, and thus should be the node that is closest to the destination along this partially connected route. Hence, the *sponsor* node is the node that is more likely to find the destination (through *route discovery*) with minimal amount of modification to the original route.

Besides *route discovery* and *maintenance*, LOTAR also provides prediction of the likelihood of the route becoming broken, through the use of a *pessimistic lifetime equation* (Equation 1), which is only triggered at that node when it is involved in routing. If this equation produces a value that is smaller than some minimum threshold value, then a *handoff flow* mechanism is activated. The *handoff flow* mechanism tries to find at least one node that is located somewhere between the two nodes that may become broken, i.e. a node with a transmission range big enough to reach both of these nodes (whose link is about to be disconnected). If such a node exists, the lifetime equation is calculated for each side of this newly selected node and the minimum value selected. If this minimum happens to be greater than the threshold, then the handoff through this node occurs. If the handoff is not established within a certain amount of time, the source is required to re-discover a new route from itself to the destination node.

$$Lifetime = \frac{R - L}{v_i + v_j} \qquad (1)$$

Where:

R is the transmission range
L is the distance between nodes i and j
v_x is the velocity of node x.

However, the incorporation of an extra node (the handoff node) into the routing path may lead to un-optimized routing, when the original handoff path established is no longer needed. Thus, LOTAR uses an additional mechanism to undo the original handoff, when the *pessimistic lifetime equation* produces a value that is sufficiently large.

2.2.2.3 Hybrid Topology Based Protocols

Hybrid protocols collectively combine the advantages of both proactive and reactive schemes, in order to achieve greater efficiencies. This is achieved by using proactive algorithms for neighbouring (local) transmissions and reactive algorithms for global transmissions. Hence up-to-date information is kept for closer nodes only, while current routes are kept for destinations further away. Thus, these schemes trade a bit of overhead for reasonable delays. However, like all topology based algorithms, these protocols require the number of topological changes to be smaller than a certain rate, since links need a finite time to be maintained, either periodically or through some *route maintenance* scheme.

The only *hybrid* protocol commonly found in literature is ZRP, as indicated in Figure 27.

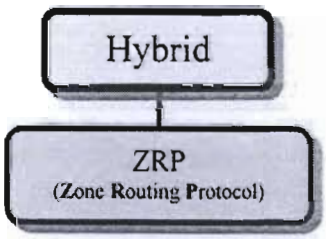


Figure 27: Showing the *hybrid* topology based protocol described within this sub-section.

2.2.2.3.1 ZRP (Zone Routing Protocol)

ZRP, given in [48], incorporates *intraZone* routing for nodes that fall within a certain zone and *interZone* routing for the nodes that fall outside this zone. A *zone* is defined to be the nodes that fall within a certain distance (in number of hops) from a particular node. Thus, a *zone radius* of two means that all nodes that are two (or less) hops away, form part of that particular nodes *zone*. *Peripheral nodes* are then defined to be nodes that are exactly the *zone radius* away. These definitions can be found in Figure 28.

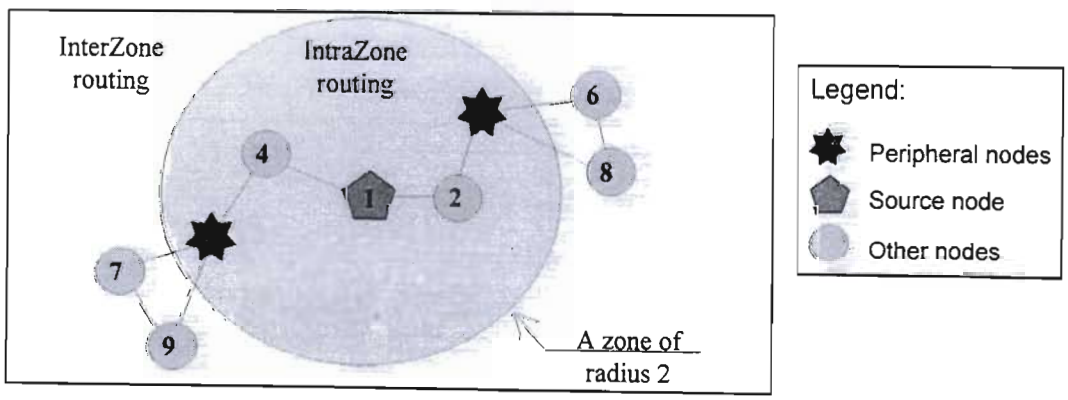


Figure 28: Showing the definitions of intraZones and interZones in ZRP.

IntraZone routing is not defined by ZRP and can be incorporated by many different routing protocols, such as DSDV, AODV, OSPF [49], RIP [50], etc., in-fact different zones may even be allocated with different routing protocols, as only the ID and the distance (in hops) of each node within the *zone* is required to be known.

InterZone routing, on the other hand, is specified by ZRP. When a source wishes to send data to a destination node, it first determines whether that node falls within its *zone*. If it does, then the routing path is known (through the *intraZone* routing protocol) and thus no further overhead is

required. If, however, it does not, then ZRP broadcasts a *route request* to all its *peripheral nodes*, adding its node to the request. This same algorithm is then employed to the *peripheral nodes*, which contain their own *zone radius* and hence other *peripheral nodes*, until the destination node is found. Care must be taken to ensure that *route requests* do not loop back into *zones* already queried, thereby causing a greater overhead than flooding [48]. Once the destination is located, a reply is sent back through all the intermediate *peripheral nodes*, thus establishing a path back to the source.

When route failures occur, the node that detected the broken link is required to send a route failure notification back to the source. The source can then decide to either search for a new route or attempt to partially fix the broken route, by issuing a request to find the missing node.

2.3 Multicast Protocols

Multicasting is a technique that is used to forward data to multiple destination hosts, simultaneously. Since this is the focus of this dissertation, this section describes those schemes that were applicable to the ad-hoc domain. However, before such an endeavor is undertaken, mention needs to be made of why traditional wired protocols were deemed inappropriate.

Traditional wired multicast protocols are based on concept of *groups*, where each *group* contains a collection of receivers. Since multiple receivers may join or leave a particular *group* dynamically through the use of IGMP (Internet Group Management Protocol) [51], a single IP multicast address is used to identify the *group*, allowing multicast sources to be unaware of the individual IP addresses of the receivers. IP multicast addresses are allocated by the IANA (Internet Assigned Numbers Authority) and fall in the range 224.0.0.0 to 239.255.255.255, some of which are never used outside the *autonomous system* (AS) (see [52]). However, a mechanism is still required by routers to identify the path to individual receivers from the allocated IP multicast address. This is achieved through use of *spanning tree* algorithms. Two main types of wired *spanning trees* exist – *source trees* and *shared trees*.

Examples of *source-based trees* include DVMRP (Distance Vector Multicast Routing Protocol) [53] and PIM-DM (Protocol Independent Multicast – Dense Mode) [54]. Both these schemes use *broadcast-and-prune* [55] and *reverse path forwarding* (RPF) [56] techniques to establish *shortest-path trees* (SPT's), routed from every source node to every multicast *group* member. Unfortunately, these techniques require the first multicast packet to be broadcasted throughout the whole network, causing a waste of bandwidth and the potential for *broadcast storms* (see section 2.3.5.1). Hence, MOSPF (Multicast Open Shortest Path First) [57] was developed to extend these protocols with link-state and *group* membership information, eliminating the need for flooding. However, even with these improvements, *source trees* produce a lot of control overhead and are unable to adapt rapidly to topological changes, which is a criteria that is crucial to the context of ad-hoc networks [58].

Shared trees, on the other hand, use a single designated node to construct the *spanning tree*. This node is called the *core node* or *rendezvous point* (RP), through which all multicast traffic is forwarded. In *shared trees*, receivers are required to send *join requests* to the core, allowing non-core routers to connect to the first router found to be part of the delivery tree. However, since multicast sources are required to send unicast packets to the core, they need not be part of the tree. Examples of *shared trees* are CBT (Core Based Tree) [59] and PIM-SM (Protocol Independent Multicast – Sparse Mode) [60]. *Shared trees* contain less overhead compared to

source trees, but use paths that may not be optimal, resulting in additional delay. Furthermore, trees of this nature require both the core node and the *spanning tree* to adapt to topology changes rapidly, or else a reduction in throughput is experienced [58].

Thus, the traditional wired multicast algorithms discussed above, cannot be directly applied to the ad-hoc context, since it demands protocols that are adaptive, yet incur little overhead, to due its' limited resources. Hence, new schemes were required. Literature [58, 61] classifies these schemes into five categories – namely *group-based*, *source-based*, *core-based*, *mesh-based* or *flooding-based* – depending on the delivery structure employed. Figure 29 shows an illustration of this classification.

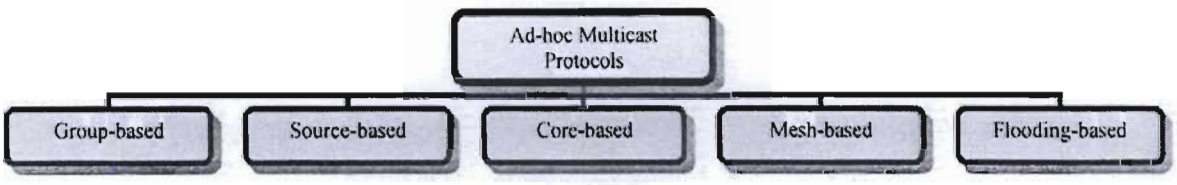


Figure 29: Showing the classification of ad-hoc multicast protocols.

2.3.1 Group-based Multicasts

Group-based protocols rely on a group of nodes, rather than a tree, to deliver multicast data. This group, known as a *forwarding group*, allows fewer states to be maintained at each intermediate node, since there is no need for maintenance of a tree, only procedures to ensure that the *forwarding group* is able to reach each multicast member. Hence, overhead is kept at a minimum. In addition, *group-based* protocols allow multicast data to be sent over forwarding nodes only, which may make use of multiple redundant paths. Examples of these schemes include the ODMRP (**O**n-**D**emand **M**ulticast **R**outing **P**rotocol) and LBM (**L**ocation-**B**ased **M**ulticast).

2.3.1.1 ODMRP (On Demand Multicast Routing Protocol)

ODMRP, defined in [62], is a flooding-based multicast protocol. However, instead of flooding data to all nodes within the mobile network, ODMRP only floods multicast packets within the *forwarding group*, which is maintained through the use of periodic control messages. The concept of a *forwarding group* was first introduced in FGMP (Forwarding Group Multicast Protocol) [63] and consists of a subset of nodes which are specially designated to forward multicast data to all *group* members. Hence, procedures are required to ensure that at least one path exists from the multicast source to every multicast destination. These procedures involve the advertisement of multicast sources and the periodic broadcasting of a *join-table* by multicast receivers.

Advertisement of multicast sources begins when a node contains data intended for a particular *group*. Sources advertise themselves by flooding a *join-request* to all nodes within the network. On the reception of the *join-request*, nodes record the address of the previous-hop node, allowing a path to be remembered back to each advertised source. If nodes wish to receive data from the advertised source, it is additionally required to broadcast a *join-table*, which consists of a list of known sources and their respective previous-hop addresses. When *join-table* messages are received by neighbouring nodes, they look for their address within the previous-hop list and, if found, broadcast a new *join-table* message (rooted at each neighbour) to their neighbours, until the *join-table* is received by each multicast source. Hence, a *forwarding group* is

established and consists of all intermediate nodes that re-broadcasted the *join-table* message. Since multicast receivers are required to periodically transmit their *join-table*, the *forwarding group* is continually refreshed. Thus, nodes may remove themselves from the *forwarding group*, if a *join-table* message is not heard within a particular time interval. For a more detailed description of the *join-table* process, see [62].

Since *join-request* messages are flooded throughout the network, multiple paths may be established to each multicast receiver. Hence, when links are broken due to mobility, alternate paths become available, ensuring data is still forwarded to receivers between refresh intervals. In addition, ODMRP can also operate as a very efficient unicast protocol, allowing implementations to use ODMRP for both unicasts and multicasts purposes.

2.3.1.2 LBM (Location-Based Multicast)

LBM [64] differs from the other multicast protocols in that it delivers multicast data to all nodes that fall within a specified region, known as the *multicast region*. Hence, LBM is a *geo-casting* protocol which makes use of location information to determine the *multicast region* and, hence, the *forwarding group*.

Like LAR, LBM makes use of a *request zone* to determine its *forwarding group*. However, in LBM, the *request zone* does not encompass the *expected zone*, but rather the intended *multicast region*. The *multicast region* is a region specified by each multicast source and is identified through the use of a rectangular box, which is recognized by four positional coordinates found within each data packet. Hence, forwarding nodes are determined in one of two ways. The first of these is through the use of a second rectangular box, which encompasses both the source node and the *multicast region* (Figure 30), while the second method allows all nodes closer to the *multicast region* (with respect to the source node) to forward multicast data (Figure 31).

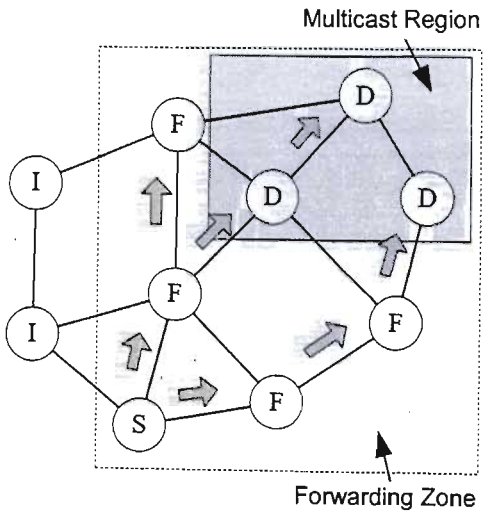


Figure 30: Showing the first scheme of LBM, in which a *Forwarding Zone* is explicitly defined.

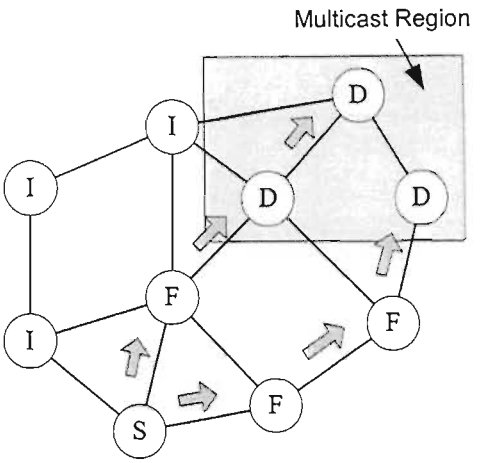


Figure 31: Showing the second scheme of LBM, in which a *Forwarding Zone* is implicitly defined.

However, since LBM makes use of the same two forwarding algorithms found in LAR, it inherits the same problems described in section 2.2.1.2.2.

2.3.2 Source-based Multicasts

Source-based multicasts require trees to be constructed and maintained for each source node within each *group*. Thus, protocols of this type forward data along the most efficient path from the source node to each multicast receiver, where efficiency is based on some metric, such as delay, load, battery life or shortest number of hops. It has been shown [65] that finding the total minimum cost for a *spanning tree* is NP-complete (i.e. order[n]^{p}), thus SPT's are generally accepted as a reasonable alternative. However, since these protocols require considerable overhead to maintain several efficient multicast trees, they are generally not employed in ad-hoc networks where link breakages occur regularly. Nevertheless, examples of *source-based* ad-hoc multicast protocols include DVMRP (**D**istance **V**ector **M**ulticast **R**outing **P**rotocol) and ABAM (**A**ssociativity-**B**ased **A**d hoc **M**ulticast).

2.3.2.1 DVMRP (Distance Vector Multicast Routing Protocol)

[66] presented three extensions to DVMRP, to limit the overhead produced by its wired counterpart. These extensions enabled DVMRP to be employed into an ad-hoc environment and included:

- Wireless leaf-node detection,
- Dynamic grafting and pruning,
- And duplication avoidance.

However, before these extensions can be discussed, an understanding of the original DVMRP is required.

In DVMRP, SPT construction requires multicast traffic to be initially flooded throughout the network domain, through the use of the RPF algorithm. When the flood reaches a *leaf node* (a node consisting of no downstream neighbours), the *leaf node* is required to check the IP multicast address of the flooded data to determine whether it is a non-member of the established *group*. If this is found to be the case, the *leaf node* may stop further transmissions for this *group*, by sending a *prune* message to its upstream neighbour. When a *prune* message is received by an

intermediate non-member node, it is required to wait to see if all its' downstream links get pruned off, and, if so, may also send a *prune* message to its upstream neighbour. Once the *prune* process has completed, the SPT is established. An illustration of this process is given in Figure 32.

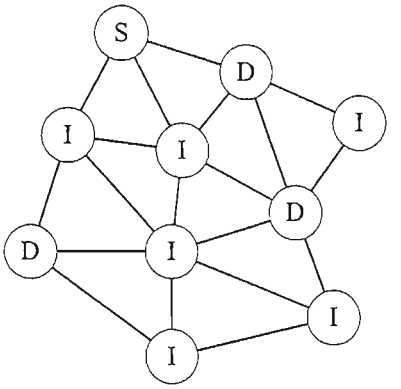


Figure 32(a): Showing the topology of the ad-hoc network used to illustrate the broadcast-and-prune method employed in DVMRP.

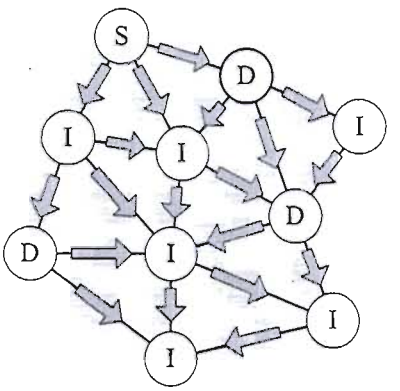


Figure 32(b): Showing the initial flood of the multicast traffic, known as the broadcast stage.

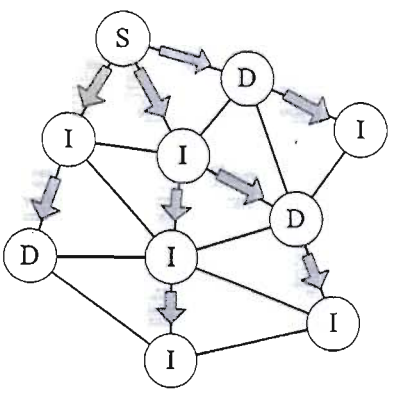


Figure 32(c): Showing how the flooded traffic resulted in the formation of a tree rooted at the source node. Note that the tree extends to all leave nodes, irrespective of their group status.

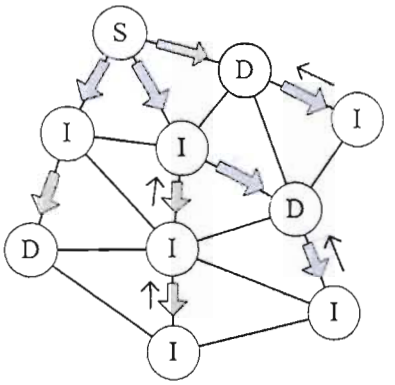


Figure 32(d): Showing how non-member leaf nodes inform upstream members to stop forwarding multicast traffic, known as the prune stage.

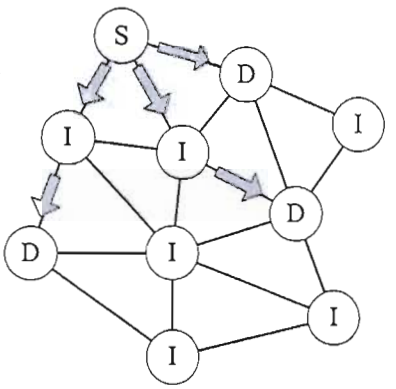


Figure 32(e): Showing the resultant SPT.

The above process works well when receivers do not dynamically join or leave a *group*. To account for this, DVMRP employs periodic timers on its member routers, which flood multicast traffic to all neighbours once the timer expires. Thus, a latency occurs when a receiver wishes to join a new *group*, since it will have to wait until the next flooding period.

Additional problems occur in mobile wireless networks. The first of these arises from the fact that *leaf node* detection is difficult to achieve when neighbouring nodes dynamically change in the presence of mobility. Two schemes to resolve this issue were proposed by [66]. One is to exchange routing tables among neighbouring nodes, while the other requires the use of acknowledgements to determine the presence of neighbours.

Another issue arises from the fact that in a highly mobile environment node movement may occur quicker than what the RPF algorithm can maintain. Hence, instead of using RPF, [66] proposed to use packet duplication as a means of controlling the flooding process. Thus, packets are only forwarded to neighbouring nodes if they are found to be unique.

Lastly, [66] extended DVMRP with the use of dynamic *pruning* and *grafting*. *Grafting* provides a means for a new member to join a *group* without waiting for the flooding period. [66] used this concept to maintain the SPT during node movement, by keeping track of upstream nodes. As soon as a node detects a new upstream neighbour with a shorter hop path to the source, it is required to send a *prune* message to the current upstream node and *graft* message to the new neighbour, causing DVMRP to adapt faster to topology changes.

However, even with these enhancements, many people question DVMRP's application to ad-hoc networks [58], due the amount of overhead it produces.

2.3.2.2 ABAM (Associativity-Based Ad hoc Multicast)

ABAM, explained in [67], is a *source-based* protocol that considers the stability of routes within the tree, since stable routes (routes containing few link breakages) require little maintenance and thus may offer reduced overhead. Hence, ABAM builds on the concepts of ABR for use within the context of multicasting.

In ABAM, trees are constructed through the use of a three-phase process. The first phase is initiated when source nodes wish to send multicast data to a particular *group*. It requires the source node to flood *MBQ* (Multicast Broadcast Query) packets to all nodes within the network. When a node wishes to receive data from the multicast source, it executes a *route selection algorithm* (RSA) to find the best route back to the source. This criterion is determined by looking at the *quality-of-service* (QoS) and stability metrics contained within the *MBQ* packet. Once a route is found, the receiver is required to reply with a *MBQ-REPLY*, triggering the second phase. When the source node receives all the *MBQ-REPLY* messages, it applies a *tree selection algorithm* (TSA) to find routes common to each receiver. Since each route received was selected to be the best, the task of the TSA is to find the best combined *spanning-tree*. Once the multicast tree is established, the final phase is instigated through the transmission of a *MC-SETUP* packet. The *MC-SETUP* packet contains the full delivery path of the tree, which is sent to all intended recipients, allowing all participating nodes to update their multicast routing tables.

The above process works well when nodes do not dynamically join or leave a *group*. ABAM accounts for this by allowing new multicast members to broadcast a *L-JOIN* (Local Join) packet. When *group* members receive the broadcast, they are obligated to return with a *JOIN-REPLY* message. Since the new member may receive multiple *JOIN-REPLY* messages, it is required to select the best route, using the RSA. Once this choice has been made, the new member is expected to send a *L-JOIN-CONF* packet over the selected route, indicating its choice.

Receivers, wishing to leave a multicast *group*, are expected to send a *L-LEAVE* (Local Leave) packet to its upstream node. Each upstream node will then delete the receiving node from its multicast routing table, until the *L-LEAVE* packet reaches either a branching node (a node which contained two or more downstream neighbours, with respect to the *spanning tree*) or another receiver. An illustration of this is given in Figure 33.

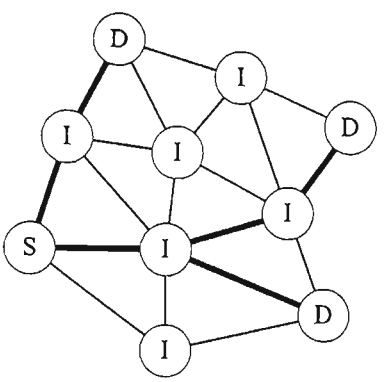


Figure 33(a): Showing the initially established source-tree.

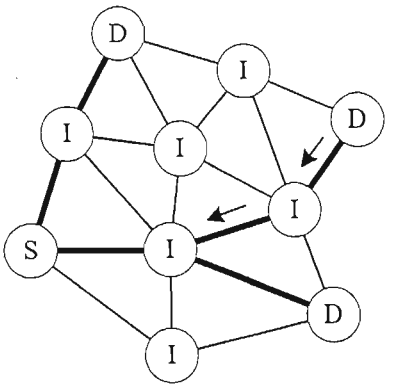


Figure 33(b): Showing how a L-Leave packet is propagated to a branching node, causing a group member to be removed from the tree.

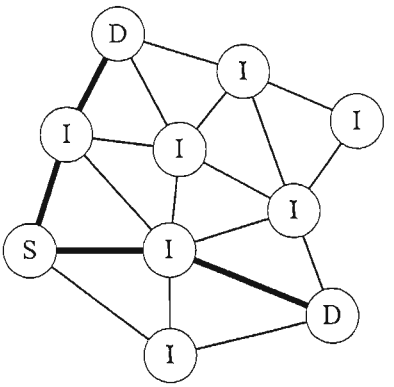


Figure 33(c): Showing the resultant source-tree.

ABAM also accounts for mobility, by initiating different tree *reconstruction* mechanisms, depending on whether the source, receiver or other tree nodes moved and whether these movements occurred concurrently or not. Essentially, ABAM tries to localize reconfigurations, so as to disturb as little of the tree as possible. Nevertheless, since the each *spanning tree* is rooted at each source of each multicast group, there are still situations where a re-establishment of the whole *spanning tree* is required. For a detailed description of each *reconstruction* scheme, see [67].

Toh et al [67] showed ABAM to deliver approximately 15% more packets (to their intended destination) than ODMRP at low mobility rates and with significantly less control overhead. But, as the mobility rate was increased, ODMRP and ABAM have been shown to deliver equally, with ABAM containing less control overhead than ODMRP.

2.3.3 Core-based Multicasts

Core-based trees were developed to limit the amount of overhead produced by *source-based* protocols. They achieved this by reducing the number of trees employed – using a single *shared tree* for each multicast *group*, which is constructed and maintained by a selected RP. RP's are responsible for distributing multicast packets to all *group* members. However, this distribution may be different, depending on whether uni-directional or bi-directional trees are employed. If a uni-directional tree is constructed, all multicast data is forwarded to the RP; else multicast packets may enter at any point within the tree. Thus, bi-directional trees offer greater efficiencies than uni-directional trees, since fewer re-transmissions are required to forward data to all *group* members [58].

However, due to the RP being the most crucial component in the operation the *shared tree, core-based* schemes suffer from a single point of failure. In addition, *core-based* schemes cause traffic congestion at shared links and result in the forwarding of multicast data over suboptimal paths, since data is forced along the *shared tree* [58]. Nevertheless, examples of *core-based* multicasts include: MAODV (**M**ulticast **A**dhoc **O**n-demand **D**istance **V**ector), AMRoute (**A**dhoc **M**ulticast **R**outing) and AMRIS (**A**d hoc **M**ulticast **R**outing protocol utilizing **I**ncreasing id-number**S**).

2.3.3.1 MAODV (Multicast Adhoc On-demand Distance Vector)

MAODV [68] is a multicast version of the unicast AODV protocol (section 2.2.2.2.1) and hence may be used for both unicast and multicast traffic.

In AODV, any destination involved in routing is required to periodically broadcast a sequence number (similar to DSDV) to ensure route *freshness* [58]. Similarly, MAODV broadcasts a multicast *group* sequence number to ensure the *freshness* of the *shared tree*, except this task is done by the chosen RP and not by each multicast destination.

MAODV selects RP's based on the criteria that this node be the first node wishing to transmit multicast data to a particular *group*. Thereafter, new member nodes are required to *graft* to the RP, by joining to the best available member node. MAODV makes this selection by broadcasting a *join* request to the desired multicast *group*. Once the request reaches a member node, a reply is sent back, indicating the *group* sequence number and hop distance. The task of the awaiting member is to then examine these returned fields and send an *activation* packet back along the path which is determined to be both the shortest and *freshest*. Once the *activation* packet reaches the intended member node, the new member node is attached to the *shared tree* and thus will be forwarded with subsequent multicast traffic destined for that *group*. An illustration of this process is given in Figure 34.

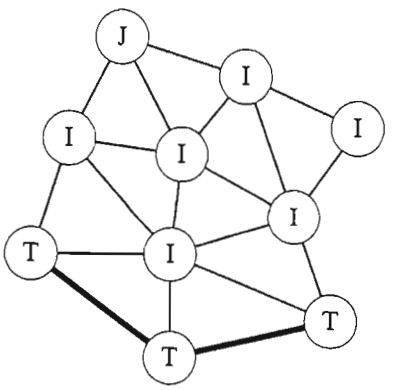


Figure 34(a): Showing the initially established tree.

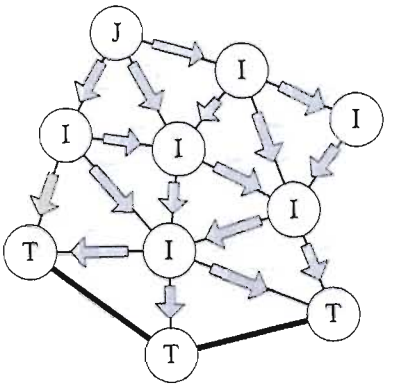


Figure 34(b): Showing how a *join* request is flooded to all tree members.

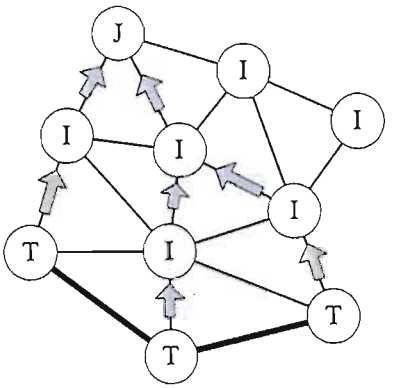


Figure 34(c): Showing how multiple replies may be sent back to the joining node.

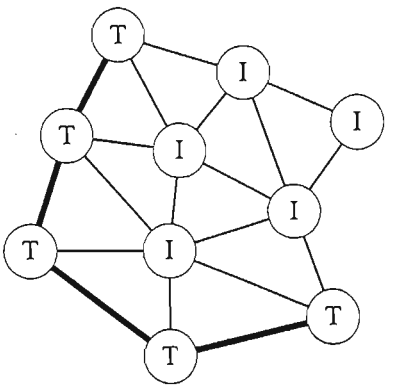


Figure 34(d): Showing the resultant shared tree, once an activation packet has grafted the joining node to an existing group member.

When a leaf member node decides to quit the multicast *group*, it is required to send a *prune* message to its neighbouring upstream *group* member, causing the upstream member to flush its' multicast routing table, with respect to the *leaf node*. However, if, in doing so, the upstream member becomes a *leaf node*, it may apply the same method to its neighbouring upstream member, causing the *shared tree* to be dynamically *pruned*.

In the presence of mobility, the *shared tree* may become segmented. Hence, MAODV requires nodes to initiate a *join* request when they become disconnected. If no reply is received, however, the node may assume that a network partition has occurred and hence elect itself as the RP. Unfortunately, this mechanism may result in multiple RP's being present when the network becomes re-connected. However, since multicast *group* sequence numbers are broadcasted, a RP will realize that another RP exists, and thus will request to *join* with to the RP containing the *freshest* tree.

Nevertheless, comparisons [69, 70] have shown MAODV to contain a poor delivery ratio, offering up to 75% less packets when compared to ODMRP. Mohan et al [70] notes this to be the result of the fragileness in the constructed *shared tree*, whilst in the presence of mobility.

2.3.3.2 AMRoute (Adhoc Multicast Routing)

AMRoute, described in [71], is a bi-directional *shared tree* based scheme that makes use of its' underlying unicasting protocol to *tunnel* multicast traffic.

Tunneling originated from the wired IP multicast implementation of the *wide-area-network* (WAN), known as the MBone [72]. Initially, problems arose in the deployment of multicast traffic for the Internet, since all established routers would need to be upgraded to support multicast capabilities. However, a mechanism, known as *tunneling,* was employed to counter this limitation. *Tunneling* allows a multicast packet to be encapsulated in a unicast datagram, before being transmitted from one multicast network to another, eliminating the need for multicast-enabled routers to be employed along the way. An illustration of this process is shown in Figure 35.

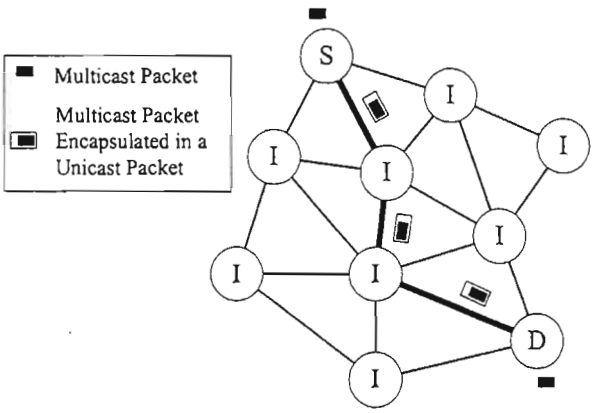


Figure 35: Showing the concept of *tunneling.*

AMRoute makes use of *tunneling* to communicate multicast traffic between *group* members, thus permitting nodes which are not interested in multicast traffic to remove all multicasting procedures. Also, since unicast packets are utilized in the forwarding of both multicast and unicast traffic, AMRoute relies solely on the underlying unicast protocol to ensuring connectivity amongst *group* member nodes. Hence, AMRoute's performance is dependant on the ability of the unicasting protocol employed.

In AMRoute, all *group* members are required to flood *Join-Reqs* to establish the *shared tree.* Once the *Join-Reqs* reaches another member node of the same *group*, the receiving member is required to mark the sender as a *group* neighbour and reply with a *Join-Ack*, allowing the sender to be aware of the receiver. Through this exchange process, all multicast senders and receivers are able to determine each other and the *shared tree* can be constructed. Thus, in AMRoute, the only members who constitute a particular *group* are the multicast senders and receivers themselves.

The tree construct phase is initiated and maintained through the periodic transmission of *Tree-Create* packets to all *group* neighbours. On reception of these packets, *group* members are required to forward them to other *group* neighbours, but only if unique; else a *Tree-Create-Nak* is sent back to the forwarding member. When a *Tree-Create-Nak* is received, the forwarding member node will realize that an alternate path exists and thus will not mark that member as a

downstream *group* neighbour. However, since knowledge of neighbouring *group* members is not erased, this link may be used on the next *Tree-Create* transmission period. Thus, through this mechanism, a single *shared tree* will be periodically established to all *group* members. Note that this method is conceptually different to usual *core-based* trees, since there are no designated RP's. However, since a single tree is used to forward data to all multicast members, AMRoute is still classified as a *core-based* scheme.

When nodes wish to remove themselves from a specific *group*, they are simply required to transmit a *Join-Nak* packet to all neighbouring *group* members and discard any data forwarded to them by the *group*. On reception of the *Join-Nak* packet, neighbouring *group* members will erase all knowledge of that neighbour, but will not attempt to fix the broken tree that may result, since it will be re-established automatically on the next *Tree-Create* transmission period.

A survey conducted by [61], has shown AMRoute to deliver poorly, due to the presence of both temporary routing loops and "critical" uni-directional links, which result from the directional nature of the *shared tree*. In addition, [61] showed AMRoute's performance to be very sensitive to both traffic load and mobility.

2.3.3.3 AMRIS (Ad hoc Multicast Routing protocol utilizing Increasing id-numberS)

AMRIS, explained in [73], is a *shared tree* protocol that makes use of increasing ID numbers to direct the flow of multicast data.

To establish direction into the *shared tree*, AMRIS requires one of the multicast sources (known as the *Sid*) to broadcast a *NEW-SESSION* packet. This packet contains, among other things, the multicast session ID, the nodes membership status (which indicates whether the node is interested in receiving multicast data from the session) and the node's *msm-id* (multicast session membership ID). Upon receiving the *NEW-SESSION* packet, each node is required to record the membership status of their neighbouring node for that particular session in a *Neighbour-Status* table and re-broadcast the *NEW-SESSION* packet with a newly calculated *msm-id*. This calculated *msm-id* is required to be larger (but not consecutive) than that received by the previous-hop neighbour, allowing *msm-id*'s to increase as it radiates outwards from the *Sid*. This reason for doing so will soon become clear, but essentially it allows a logical *height* to be assigned to each node, as done in TORA (section 2.2.2.2.4), except that, in AMRIS, the *height* is rooted at the *Sid* and not at the destination. Once the session has been established, each node is required to broadcast *beacon* messages, which allow neighbouring nodes to periodically update their *Neighbour-Status* table.

When a node wishes to join a multicast session, it is required to look in its' *Neighbour-Status* table for a possible neighbouring node that contains a smaller *msm-id*, since this node will be a possible parent to the *Sid*. Once such a node is found, a *Join-Req* packet is unicasted to it. If the node receiving the *Join-Req* packet is already a member of the multicast session, it is required update its' *Neighbour-Status* table and reply with a *Join-Ack* packet; otherwise, it is required to forward the *Join-Req* packet to its' potential parent node, until a multicasting session member is found. If, however, the original node does not receive a reply within a certain time interval, it is required to issue an *expanding ring search* (a broadcast which is only flooded to n next-hop neighbours, where n is increased each time) to find a session member. Once the broadcast packet is received by a session member, it is required to reply with a *Join-Ack* packet, which is

transmitted along the reverse path back to the joining node. However, since many replies may have been returned, no multicasting packets will be forwarded to the joining node, until it sends a *Join-Conf* packet back to the chosen session member.

Since the *Neighbour-Status* table is periodically broadcasted, neighbours are able to detect when they become partitioned from the *shared tree*. When this happens, the disconnected neighbours are required to re-join the tree through the joining mechanism described above. However, since a finite time exists before the node becomes reconnected to the *shared tree*, many packets may be dropped to both the disconnected receiver and its dependencies. Thus, simulations conducted by [61] show AMRIS to deliver less packets compared to other *mesh-based* protocols (described next). In addition, [61] showed AMRIS to deliver only 60% of its' packets (to all intended recipients) when nodes were kept stationary. Lee et al [61] suggested this to be the result of collisions which occurred from the relatively small periodic time interval set by the AMRIS protocol, which is used for the broadcasting of the *Neighbour-Status* table.

2.3.4 Mesh-based Multicasts

Mesh-based protocols were developed to provide multiple multicasts paths to *group* members, instead of a single path, as discussed in previous schemes. The reason for do so becomes apparent when one considers the likelihood of link breakages in the presence of mobility. If multiple paths are established and one of these paths becomes invalid, data will still be sent along the remaining paths. This of course leads to extra overhead, since data is unnecessarily duplicated along all established paths, but does allow data to be received by the *group* in the presence of frequent link breakages, such as found in highly mobile ad-hoc environments. Also, since multiple paths exist, *mesh-based* protocols result in the mesh (multi-connected *spanning tree*) being repaired less often, reducing any overhead experienced during *reconstruction*. An example of this is the Core Assisted Mesh Protocol (CAMP).

2.3.4.1 CAMP (Core Assisted Mesh Protocol)

CAMP [74] requires both the multicast source and destination *group* members to join to the mesh, which is established through the use of one or more designated RP's. Nodes join to the mesh in one of two ways. The first of these is to check if any neighbouring nodes are already part of the mesh. If so, then all that is needed is a simple join request/reply exchange, or else nodes are required to send join requests to one of the designated RP's, through the use of its underlying unicast protocol. However, if this RP cannot be found, an *expanding ring search* is employed to find at least one *group* member node. Once the request reaches either the RP or the *group* member, a reply is returned, causing a multicast path to be established back to the requesting node. Since, multiple join options exist, mesh protocols do not suffer from a single point of failure. In addition, CAMP requires each designated RP to send explicit join messages to each other, ensuring connectivity.

In CAMP, nodes are also required to maintain an *anchor table*. This table is used to determine which neighbouring nodes depend on it, with respect to the forwarding of multicast data. Thus, whenever membership information alters, a MRU (Multicast Routing Update) needs to be transmitted amongst neighbouring nodes to ensure integrity. The MRU contains a list of all awaiting anchors, as well as, any newly discovered sources. When a node wishes to leave a particular multicast *group*, it is required broadcast a *quit* notification, allowing neighbouring nodes to adjust their *anchor table*. However, if, after adjusting the *anchor table*, there are still

nodes depending on the quitting node, multicast data will still be forwarded to this node and will only cease when all dependencies have be removed. Hence, the objective of the *anchor table* is to ensure routers are not prematurely removed from a multicast *group*, which may result in loss of multicast data.

Since CAMP makes use of a mesh, when links become broken, a valid route is expected to still exist. However, if one does not, the disconnected multicast receiver is required to re-join the mesh, using the join mechanism described earlier. In addition, multicast receivers need to periodically ensure that the established mesh contains the shortest hop path between itself and every other multicast source. Receivers are able to determine this by comparing the shortest-hop information (obtained from the underlying unicast protocol) to that found within the nodes' *message cache*. The *message cache* is a buffer of memory that is used to store a list of recently received data packets. Hence, the performance of CAMP relies on the ability of the unicasting protocol employed. When a mismatch in the path length occurs, receivers are required to send one of two types of join messages, depending on whether the neighbouring node is already multicast member or not. If the neighbouring node is already a multicast member, a *heartbeat* is sent; else a *push join* is transmitted. Like the ordinary join process, these join messages simply add an extra path to the receiver. An example of the *push join* mechanism is given in Figure 36 below.

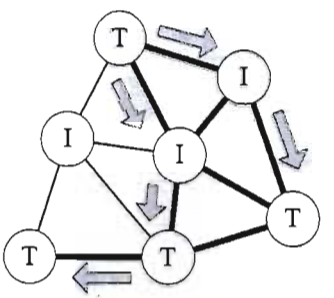


Figure 36(a): Showing a situation in which multicast traffic is being received on a sub-optimal mesh.

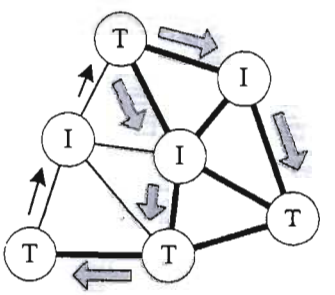


Figure 36(b): Showing a push-join message being sent by a multicast member, in order to include the shortest path into the established mesh.

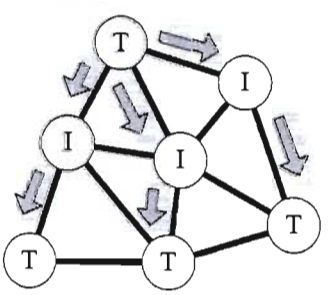


Figure 36(c): Showing the resultant mesh that now includes the shortest path to all group members (T).

Simulations performed by Garcia-Luna-Aceves et al [74] have shown CAMP to outperform wired *source-based* multicasts, such as DVMRP and PIM-DM. Since CAMP contains multiple alternate paths, CAMP has also been shown to perform better than *tree-based* protocols. In addition, [74] proves CAMP to scale better than ODMRP, when the number of *group* members is increased.

2.3.5 Flooding-based Multicasts

Flooding-based multicasts simply broadcast multicast data to all nodes within the network. These protocols are usually employed in highly mobile environments, where previous schemes are inadequate [75], due to the production of large protocol overheads and poor data delivery ratios. Hence, new algorithms were devised to counter these deficiencies. Literature [76, 77] on *flooding-based* protocols has classified these schemes in four main families – namely *blind flooding, probability-based* floods, *area-based* floods and *neighbour-knowledge* floods.

2.3.5.1 Blind Flooding

Blind flooding is by far the simplest of these schemes. It requires all newly received data to be re-broadcasted to neighbouring nodes. Thus, a network consisting of *m* nodes will experience *m* re-transmissions. However, since all *m* nodes may be reached after *n* re-transmissions (Figure 37), *m-n* redundant transmissions occur, resulting in poor bandwidth utilization and a waste of unnecessary battery power, especially as the density of the network is increased (see [77]).

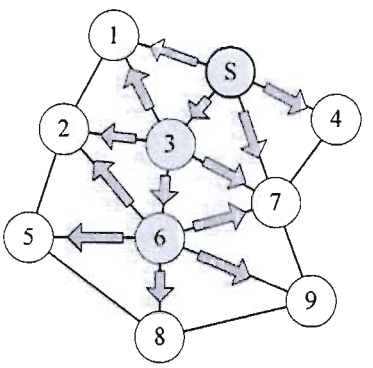


Figure 37: Showing how m=10 nodes can be reached after n=3 re-transmissions.

In addition, flooding schemes suffer from the *hidden terminal* problem [78]. The occurs when two out-of-range nodes wish to send data to a common receiving node, since both transmitters will evaluate the channel to be free and thus cause a collision to occur at the receiver. An illustration of this is given in Figure 38, below. One may think that such occurrences should happen for all modes of communication and not just for broadcasted traffic. Although this may be true, mechanisms have been developed to reserve the channel for unicast transmission (see "virtual carrier sensing" in section 3.5.1.2), causing less collisions to occur in this mode.

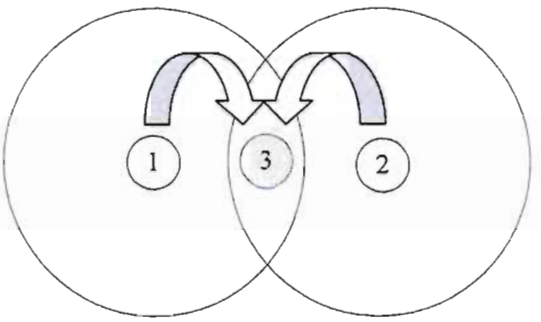


Figure 38: Showing the *hidden terminal problem*, in which nodes 1 and 2 sense the channel to be free and thus transmit data simultaneously, only to cause a collision to occur at node 3.

However, reservation of the channel for multicasting and broadcasting purposes is non-trivial, since it is dependent on both the number of next-hop neighbours available, as well as, the underlying routing protocol employed. Hence, reservation for broadcasting is not performed by the IEEE 802.11(b) MAC specification (used herein, see Chapter 3), resulting in the presence of many unnecessary contentions and collisions, known as the *broadcast storm problem* [76]. Hence, more sophisticated methods are required to reduce this redundancy.

2.3.5.2 Probability-based Floods

Probability-based protocols are similar to *blind flooding* schemes, except that they only re-broadcast a packet with a probability equal to some value. An example of such a method is the Probabilistic scheme.

2.3.5.2.1 The Probabilistic Scheme

The Probabilistic scheme, defined in [76], simply re-broadcasts a packet with a hard-coded probability of x. If x is set to one, a *blind flood* is performed; otherwise a node has a probability of $1\text{-}x$ of not receiving the packet from its' immediate previous-hop neighbour. However, the actual probability of a node receiving a broadcast is proportional to the number of neighbouring nodes which surround it. This can be easily seen by the following example:

If node a has two neighbours, each of which may broadcast data with a probability of x, then the probability that a will receive the packet is:

$$x + (1-x)x = 2x - x^2$$

However, if a had three neighbours, then the probability would have been:

$$x + (1 - x)x + \mathbf{(1 - (x + (1 - x)x))x} = 3x - 3x^2 + x^3$$

Leading to a probability increase of $x - 2x^2 + x^3$, with the addition of a single neighbouring node.

Hence, *probability-based* protocols are usually employed in dense networks, where nodes may contain many neighbours, thus reducing the amount of redundancy that would have resulted had a *blind flooding* scheme been utilized. Since there is a reduction in the amount of re-broadcasted data, less congestion is also experienced, resulting in a higher network throughput.

However, since probability schemes cannot guarantee that nodes will receive the broadcasted data, they are unreliable. This is especially true for the nodes that surround the edge of network, since they contain fewer neighbouring nodes and hence obtain a lower probability of receiving

data. Suggestions have been made by [77] to employ an adaptive scheme, whereby the probability is based on the number of neighbouring nodes that are present. Unfortunately, to-date no results have been found with this modification included.

2.3.5.3 Area-based Floods

Area-based flooding make use of thresholds to determine whether or not to re-broadcast a packet, which is derived from an equation known as the EAC (Expected Additional Coverage) [76]. The EAC basically gives a node an indication as to the amount of additional area that would be covered through the re-transmission of the packet. If this area is found to be above a preset value, the packet is re-transmitted, or else it is dropped. Hence, a packet is only re-transmitted if it going to traverse a certain amount of additional radio coverage.

Examples of such protocols include the Counter-Based, Distance-Based and Location-Based schemes.

2.3.5.3.1 The Counter-Based Scheme

The Counter-Based scheme [76] counts the number of times a particular packet is heard from its' neighbours and, depending on the value of this counter, makes a decision on whether the packet should be re-broadcast or whether it should be dropped. The basis for doing so is derived from the fact that the additional area covered by a nodes' transmission radius decreases as the number of re-transmissions from neighbouring nodes increases (see Figure 39). Ni et al [76] showed, through simulation, that, after hearing the same packet four times, the EAC of a node is below 0.05%.

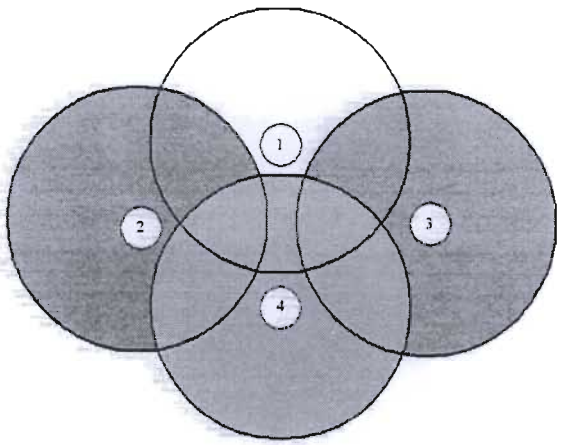


Figure 39: Showing how the additional area gained by node 1 (the white region) decreases, as neighbouring nodes re-transmit the broadcast data.

Williams et al [77] suggested two practical ways for the implementation of the Counter-Based scheme. The first of these is to simply pass the packet to the MAC layer, where it would be buffered in an interface queue. A packet would then remain in this queue until all prior packets have been sent and the channel becomes clear again for it to be transmitted. Thus, during this time period, neighbouring nodes may re-broadcast the same awaiting packets. Hence, every time such a broadcast is heard, a count is kept, and, if the counter reaches above a particular value, the packet is removed from the interface queue, preventing it from being transmitted.

The second method involves delaying the packet for a random amount of time, prior to sending it to the MAC layer for transmission. During this time, a count is kept of neighbouring

transmissions for that particular packet, and, if above a particular value, the packet is discarded, preventing it from being sent to the MAC.

However, since the Counter-Based scheme uses thresholds, it is unreliable. Nevertheless, the Counter-Based scheme is superior to the Probabilistic scheme, since it "listens in" on transmissions and hence is able to adapt automatically to the number of neighbouring nodes which surround a node.

2.3.5.3.2 The Distance-Based Scheme

The Distance-Based scheme, given in [76], examines the distance of each neighbouring node. If all distances are below a threshold value, the packet is dropped, or else it is re-broadcasted. Hence, the Distance-Based scheme considers whether the additional area covered by a transmission is above some threshold value, since this area is proportional to distance, as indicated in Figure 40.

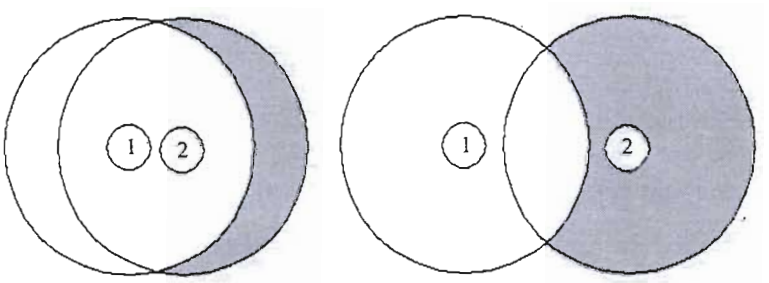


Figure 40: Showing how the distance between two nodes is proportional to the additional area gained, as indicated by the shaded area of node 2.

[77] suggested using the same two implementation methods that was employed for the Counter-Based scheme, except, instead of counting each packet, the distance of each neighbour is sampled, just prior to a packets' transmission. Hence, if this sample is determined to be above the defined threshold, the packet is transmitted; otherwise it is dropped.

However, since the Distance-Based scheme only determines the additional area that would be gained and not whether all nodes are able to be reached, it too is unreliable.

2.3.5.3.3 The Location-Based Scheme

The Location-Based scheme, developed in [76], is similar to the Distance-Based scheme, expected that it uses a more accurate mechanism to determine the EAC.

In the Location-Based scheme, each transmitted packet is expected to contain the location of the node which re-transmitted the packet. Upon receiving the packet, neighbouring nodes calculate the actual additional area that would be covered, if the packet were to be transmitted by the receiving node. If this value is determined to be above a certain threshold, it is sent for transmission, through the use of one the two methods described previously. Since both these methods cause the packet to be delayed, duplicate packets may be received from other neighbouring nodes. When this happens, the actual additional area is re-calculated. If this calculation causes the additional area to drop below the predefined threshold, the packet is dropped; otherwise the packet is kept untouched, until it is finally transmitted.

Since the EAC is continuously re-calculated using the position of each node, the Location-Based schemes is very computationally expensive, but results in an improved estimate of the

additional area covered. Hence, the Location-Based protocol is by far the most superior *area-based* protocol developed (see [76]), but, since it makes use of a threshold value, it is also unreliable.

2.3.5.4 Neighbour-Knowledge Floods

Neighbour-knowledge floods keep track of the state of neighbouring nodes to broadcast data more effectively to all nodes within the network. Thus, these protocols are as reliable as blind floods, but are more computationally expensive. However, since redundancy is kept to a minimum, *neighbour-knowledge* techniques are far more efficient than *blind flooding*.

Examples of these techniques include Flooding with Self-Pruning, the Scalable Broadcast Algorithm, Multipoint Relays and the Ad Hoc Broadcast protocol.

2.3.5.4.1 Flooding with Self-Pruning

Flooding with Self-Pruning, given in [79], (known hereafter as FSP) requires each node to keep knowledge of its' 1-hop neighbours, which is achieved through the use of periodic *hello* messages.

When a packet is broadcasted in FSP, the transmitting node is required to include a list of all its' 1-hop neighbours within the header of the packet. Once this packet is received, nodes are required to mark-off (*prune*) all neighbours that have been included in this list. If there are any neighbours that still exist after the *prune*, the packet is re-broadcasted; else the packet is determined to be redundant and hence will be dropped.

However, since FSP assumes perfectly circular transmissions, it cannot account for collisions, which may cause a set of neighbours to be marked as having received the packet, when in fact they have not. This is especially true in regions where obstacles like trees, buildings, motor vehicles, etc. may interfere with transmission patterns.

2.3.5.4.2 SBA (Scalable Broadcast Algorithm)

SBA [80] is similar to FSP, except that it is aware of neighbours within a 2-hop radius.

Like FSP, neighbour knowledge is achieved through the use of *hello* packets. However, in this case a *hello* packet is required to contain a list of all known neighbouring nodes, instead of just the ID of the sending node. Thus, when the packet is received by neighbouring nodes, a two hop topology can be determined for each neighbour.

When a broadcasted packet is received, a node is required to examine its' established two-hop table to determine which neighbours should have all received the packet. If, after examining the table, there are still neighbours outstanding, the node that received the packet is required to re-broadcast it; else the packet is simply dropped. However, instead of simply re-broadcasting the packet out, a node is required to delay the broadcast for a random amount of time, where the upper limit of the random number is selected to be:

$$\frac{(d_N \max)}{(d_N)} \qquad (2)$$

Where:

d_N max - is the maximum number of neighbouring nodes a node contained,
d_N - is the number of neighbours a node currently contains.

Hence, nodes with more neighbours will generally re-broadcast a packet quicker than nodes containing fewer neighbouring nodes, thus prevent as much redundant packets as possible. If a duplicate packet is received from another neighbour during the delay period, a node is required to re-examine its two-hop table to determine if all nodes have now been covered. If this is the case, the packet is simply dropped; otherwise the packet is allowed to continue, until it eventually expires, causing it to be sent to the MAC layer for re-broadcasting.

Results conducted by [77] show SBA to be far more efficient than *blind floods*, *probability-based* floods and *area-based* floods. However, Williams et al [77] did show SBA's performance to be highly dependant on the accuracy of the 2-hop information contained within its' table. In addition, SBA, like FSP, assumes perfect transmissions, rendering it unreliable in environments where obstacles may cause interference.

2.3.5.4.3 MPR (Multipoint Relays)

Like SBA, MPR [81] makes use of two-hop neighbour information to flood data to all nodes within the network. However, instead of scheduling the broadcasted data, MPR explicitly selects the next-hop neighbours that are required to cover all 2-hop neighbouring nodes. Thus, only the chosen nodes, known as *multipoint relays*, are allowed to re-broadcast data.

In MPR, *multipoint relays* are chosen through the use of the following algorithm:

- First select next-hop neighbours that contain neighbours which can only be reached by the next-hop neighbour in question and make them *multipoint relays*.
- Then, for all remaining neighbours, select those which cover the most number of neighbours that are not covered yet and make them *multipoint relays*.
- Finally, repeat the previous step, until all nodes have been covered.

Once the *multipoint relays* have been chosen, nodes are required to send *hello* packets to neighbouring nodes, informing them of the selection. Thus, when data arrives, nodes are required to look-up whether they are a *multipoint relay* for a particular transfer. If they are, they are required to re-broadcast the packet, or else the packet is simply dropped.

However, since MPR also gathers its 2-hop information from *hello* messages, it too suffers from the same problems described in SBA.

2.3.5.4.4 AHBP (Ad Hoc Broadcast Protocol)

AHBP, described in [82], is similar to MPR in that it selects next-hop neighbours to re-broadcast data. In fact, ABP adopts the same algorithm to choose these nodes, but calls them Broadcast Relay Gateways (BRG's). However, the difference between AHBP and MPR is the way in which BRG's are informed.

In AHBP, BRG's are informed on a per-packet basis. Thus, each selected BRG is required to be appended to the broadcasted data. On reception of the packet, neighbouring nodes are required to look in the header to determine whether they are a BRG or not. If not, the packet is simply dropped; otherwise these neighbouring nodes are required to re-apply the *multipoint relay* algorithm to determine the next set of BRG's, which is appended to the re-broadcasted packet.

Thus, AHBP is more computationally expensive than MPR, but allows up-to-date information (regarding neighbouring nodes) to be used in the calculation of the BRG set, at each successive hop. In addition, AHBP is able to account for neighbours which have already been covered by the received packet, allowing them to be excluded in the calculation of the next BRG set, thus improving redundancy. Also, if mobility causes a node to receive a packet for a next-hop neighbour for whom it did not expect, it is required to automatically assume BRG status and thus re-broadcast the packet. However, once the *hello* message reaches the confused node, it will be informed of the altered topology and hence will be able to account for the new neighbouring node on the next packet arrival.

A survey conducted by [77] showed AHBP to perform better than SBA in semi-static and highly congested environments, but not in highly mobile scenarios. This is because AHBP is unable to recover from situations which result when a chosen BRG moves out of range of the transmitting node, causing the selected BRG set to fail (with regards to covering all expected next-hop neighbours). However, since no BRG's are selected in SBA, it does not experience this shortcoming.

2.4 Conclusion

Current unicast ad-hoc networking protocols are able to determine a path to a destination through either proactive next-hop tables or reactive routes. While, reactive schemes are required to search the entire network space for a route before a single packet can be sent, proactive protocols are simply required to lookup the address of the neighbouring node that will aid in the forward progression of a packet to a particular destination. This is because the next-hop table of proactive schemes are continuously updated, allowing routing information to be readily available. Hence, proactive methodologies are capable of forward data faster, since a routing path to every destination is defined implicitly.

The same is true for *location based* protocols. Here, forwarding decisions are established through the position of neighbouring nodes, which are determined by some location *service*. But, unlike the proactive schemes, *location based* protocols should never be used in isolation. This is because results have shown that routing in this manner causes large protocol overheads to result, and the potential for *broadcast storms*. However, if location information is used correctly, papers have shown that the overall performance of a protocol can be improved [21, 25, 83].

Nevertheless, reactive protocols have been shown to route packets more efficiently than their proactive counterparts. The main reason for this is due to the *on-demand* strategy employed by the reactive schemes, which incurs protocol overhead on an "as needed" basis. Since this brings scalability into a protocol, reactive schemes supersede proactive schemes in that they are able to be deployed in networks that consist of a large number of nodes.

When looking at multicasting, however, a different outcome was found. Multicasting relies on the use of a *group* address, which defines a particular set of logically connected destination

nodes through the use of a single ID. Since the wired IP infrastructure could only support one destination address at a time, *group* addresses solved the problem of having to address multiple receivers simultaneously. In addition, *group* addresses also allowed multicast sources and receivers to be independent of each other, which permitted nodes to dynamically connect and disconnect from a *group* without affecting the delivery of data to other *group* members. However, in order for such a forwarding system to operate correctly, some kind of *spanning tree* was necessary.

Spanning trees give routers the ability of knowing how to forward multicast data. But, since these trees are required to be up-to-date, some kind of *table-driven* scheme is generally required, although schemes try to establish and maintain the tree using triggered events, thus, bring an *on-demand* "feel" to multicasting. Nevertheless, *groups* require the use of large transmission and storage overheads, which, in the presence of mobility, were unable to deliver an acceptable number of packets to all multicast members. Since bandwidth and other resources were being consumed with little befit, alternate solutions were required for multicasting in mobile ad-hoc environments.

One field that developers explored was flooding. Previously, flooding was done only through the *blind-flooding* technique. However, this scheme produced many unnecessary re-transmissions, which lead to the development of various flood-limiting methodologies. Although these strategies provided a simple means for data to be sent all nodes of a network, they can only be employed in large networks. This is because flood-limiting protocols will approach that of *blind-flooding*, as the number of nodes in a network is reduced. Williams et al [77] showed that the distinction between a large and small network (for flooding purposes) was approximately 20 nodes. Thus, if a network contained 20 nodes or less, *blind-flooding* should be used instead of any of the other flood-limiting techniques, as very little performance would be gained at the expense of additional storage and processing costs (needed by each flood-limiting scheme).

Therefore, in a network that consist of 20 nodes or less, little research has been conducted to forwarded multicast data over structures that do not require the use of *spanning trees*. Since such a solution was necessary for the operation of the PCS (section 1.2), a novel routing technique called LAMP[3] was developed to overcome this limitation.

This scheme used a modified version of the DSDV protocol to identify neighbouring nodes that were deemed suitable for the forwarding of multicast data. But, since mobility may cause neighbouring links to become temporarily invalid, a *location based* scheme (adopted from LOTAR implementation of LAR) was used to find an alternate route around each broken link. This way, a localized flood was employed only during link failures, thus, ensuring congestion was kept to a minimum. However, for more information on why these schemes were selected, refer to Chapter 5.

Thus, since the selected protocols were derived from three different research areas (proactive, reactive and *location based* schemes), this survey needed presentation of each. In addition, as this dissertation was concerned with the details of multicasting, these schemes were given to show why current schemes were deemed inapplicable to the requirements of the PCS.

---

[3] Location Aided Multicasting Protocol.

The Physical Environment

3.1 Introduction

This chapter describes the physical devices and the selected operating system that was used to implement the proposed routing scheme. The rationale for the inclusion of this chapter lies in the fact that extensive research was required to understand how packets were being processed at each layer of the operating system. This understanding was necessary in order to define the requirements needed by the ad-hoc routing algorithm. Hence, this chapter provides a level of detail sufficient to allow the reader to assess the complexities associated with the different services that can be found within each layer. Once this understanding has been attained the details relating to the manipulation of this framework to incorporate the newly proposed protocol can be provided.

3.2 The Mobile Handheld Devices

To provide the necessary processing power for each node within the PCS, sixteen commercially available HP (Hewlett-Packard) 3870 iPAQ PDA's were purchased. These devices came standard with the following "off-the-shelf" features [84]:

- 240 x 320 pixel (5.7 x 7.7 cm) 16K (65,536) color touch-sensitive reflective TFT LCD,
- 206 MHz Intel StrongARM SA-1110 32-bit RISC processor,
- 64 MB SDRAM with an additional 32 MB Flash ROM,
- 1400 mAh Lithium Polymer rechargeable battery,
- Compaq Expansion Pack System that allows Type II PCMCIA cards to be connected,
- Numerous supported interfaces that include a microphone, a speaker, a 5-way joystick, a handwriting recognition system, a software keyboard and a serial/USB slave port,
- Integrated Bluetooth™ technology,
- Microsoft® Pocket PC 2002 operating system.

An illustration of the 3870 iPAQ PDA, with all its' features, is depicted in Figure 42 (over the page).

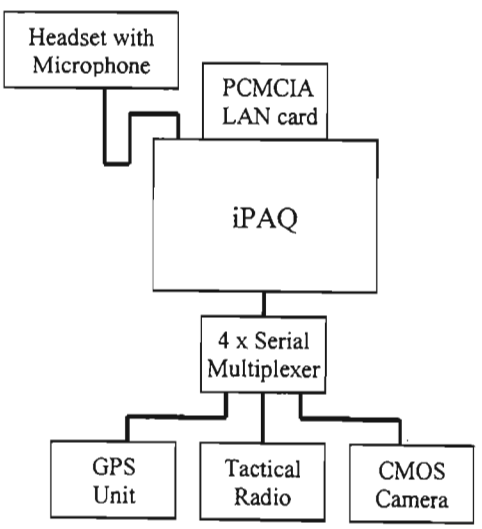
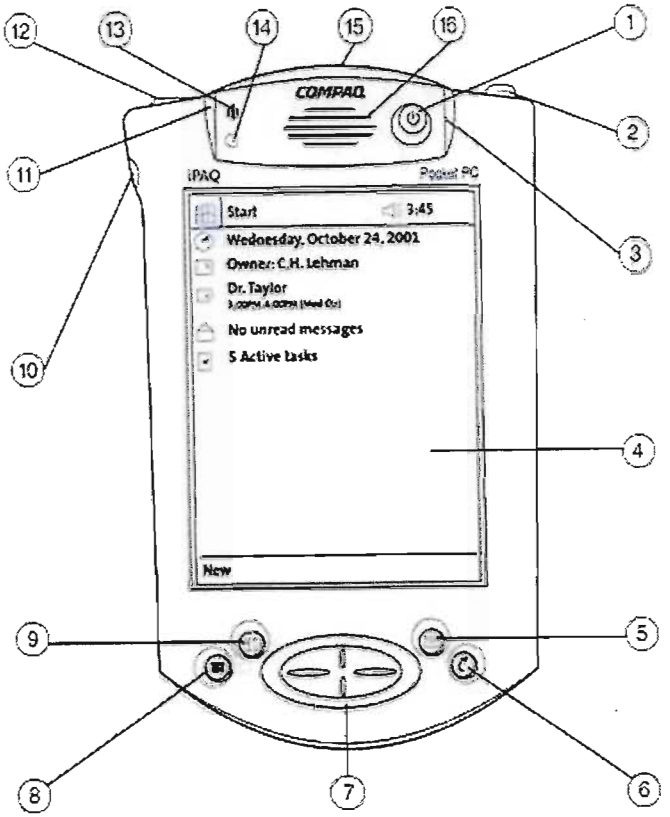


Figure 41: Showing the iPAQ sub-system.

However, in addition to these features, all PDA's were fitted with wireless Cisco Aironet 350 PCMCIA LAN cards, a headset and a 4-way serial multiplexer (Figure 41). The wireless LAN cards were used to give these devices the ability to communicate with each other through the IEEE 802.11(b) MAC protocol, while the headset provided the necessary hardware to perform voice communication. The multiplexer was additionally incorporated to give each iPAQ the capability to receive GPS information, tactical radio data and CMOS camera images from its' on-board serial port.



1. Power Button
2. Stylus
3. Charging/Notification Indicator
4. Color Display w/Integrated Touch Panel
5. Inbox Button
6. iPAQ Task Manager Button
7. Directional Pad
8. Calendar Button
9. Contacts Button
10. Record Button
11. Active Bluetooth Indicator (Bluetooth models only)
12. Stereo Headphone Jack
13. Microphone
14. Ambient Light Sensor
15. SD Slot
16. Speaker

Figure 42: Showing the features of the 3870 iPAQ PDA from HP [84].

3.3 The Selection of the Operating System

An embedded ARM Linux kernel OS (operating system) was selected as the OS of choice. The reason for doing so is due to the open source nature of ARM Linux, allowing the kernel to be easily understood and manipulated, especially when a new routing protocol is required to be incorporated into the existing IP (Internet Protocol) stack.

However, many people question Linux's application to embedded systems. The reason for this is due to the amount of RAM, ROM, flash, and processing power Linux requires [85]. Although this may be true, Linux's inclusion of a fully standards-compliant TCP/IP stack and GUI makes it very attractive to high-end embedded systems, such as the iPAQ [85]. Previously, such high-end devices did not exist, forcing embedded producers to adopt custom built real-time operating systems (RTOS), aided by compact software development environments such as eCos [86], VxWorks [87], QNX [88], and LynxOS [89]. But with decreasing memory prices and increased processor speeds, custom built RTOS are no longer necessary, since well-known commercial "off-the-shelf" operating systems are able to be ported to such environments. This is further emphasized by Microsoft's development of Pocket PC, which is simply an embedded version of the ordinary Windows operating system for the iPAQ.

Having said this, however, many embedded Linux distributions exist. The reason for this has been largely through HP's (Formerly Compaq) continual support for the development of a reference distribution of the Linux kernel for the iPAQ, known as `hh` [90]. This distribution originally consisted as a ported version of the XFree86 implementation of the X Window system for the StrongARM (SA 11xx) processor, which was released in May 2000 [85]. In addition to this, HP also sent free iPAQ devices to major Linux and open-source players, and hosts a community resource website at http://www.handhelds.org, which provides handheld developers with Web, FTP CVS and mailing-list support to all its' open-source solutions [90]. This generated much interest within the larger global community resulting in the rapid development of an embedded Linux solution for the iPAQ. In fact, solutions developed so quickly that HP couldn't keep up, eventually forcing HP to replace its' reference distribution with that of Familiar [91].

The Familiar Project consists of a small group of developers that are dedicated to the creation of a stable OS and supporting applications for the iPAQ. Initially, the project customized `hh` to include Python and anti-aliased font packages, which could be accessed within its' Blackbox window manager [90]. Currently, however, this has been modified to include a user-friendly packaging support system called `ipkg`, which allows packages to be loaded, updated and upgraded as required. Collectively, these components form the Familiar Distribution, which has now become the new `hh` reference distribution [90].

While the Familiar Project focuses on the development of a stable OS, at least three other vendors have focused on the implementation of alternative GUI's. These being Century Software, Trolltech and Transvirtual Technology [85, 90].

Century Software [92] builds on the `hh` distribution to equip it with a GUI, which consists of a number of layers. At the base of this distribution lies Microwindows, a windowing environment that contains two API's – Microwindows and Nano-X – that resembles the Windows GDI and the X system [85] respectively. However, Century Software decided to focus on its' Nano-X API, since it wanted to preserve X's option of running the OS in a client-server environment. Thus, to Nano-X, a FLTK (Full Light Toolkit) layer was added, called FLNX (Full Light Nano X). FLTK provides all the necessary widgets that are required to interact with text boxes, push buttons, menus, etc. [85]. Hence, Century Software allows ordinary FLTK calls to be translated by FLNX to Nano-X calls, which in turn are converted by Microwindows to Linux frame buffer calls. In addition, Century Software wanted to incorporate the fully standards-compliant KDE (K Desktop Environment) 1.0 browser into its distribution, due to its' compactness. But, KDE required Trolltech's Qt application framework [85]. Thus, Century Software developed another layer, known as the Qt compatibility layer, allowing Qt calls (made by the KDE widget) to map to the corresponding FLTK calls. However, the performance of this layer is questionable, since each call will be re-translated five times, before actually being processed. Nevertheless, Century Software provides a windowing GUI distribution that is fully FLTK and web browsing compliant.

Like Century Software, Trolltech [93] also incorporated GUI facilities into the `hh` reference distribution, which is independent of the X environment itelf. However, being the original developers of the Qt Palmtop Environment (QPE), Trolltech's framework is conceptually simpler than that of Century Software. Instead of using multiple layers, Trolltech simply translates all Qt calls directly to Linux, through its Qt/Embedded framework (a compact version

of Qt) [85], making it ideal for Qt applications, such as KDE. However, Trolltech's simplicity may also be its downfall, since it is unable support as many API's as that found with Century Software. Nevertheless, it does offer a compact GUI, with increased performance. However, although Trolltech's distribution is free and open, it requires a license to develop commercial products [90].

Transvirtual Technology [94], on the other hand, takes a rather different approach. Although it builds on the hh reference distribution, it aims is to provide the iPAQ with a Java Virtual Machine (JVM), allowing applications to be rapidly deployed without regard to platform considerations [85]. However, since Java requires the use of a JIT (Just-In-Time) interpreter, many people question Java's applicability to an embedded operating system. Nevertheless, being founded by the authors of Kaffe Java [95] (one of the best open-source JVM developed for the PC), it is expected that Transvirtual Technology's distribution, called PocketLinux, will provide long-term support to the iPAQ [85].

Nonetheless, since the above three vendors all base their distributions on the hh reference, the Familiar Distribution will always be "one step ahead of the pack", causing it to be selected above all other vendors, especially for the adoption of a new routing protocol.

3.4 Linux and its' Networking History

Networking, and the Internet, started its roots in 1962 through the U.S. Department of Defense's (DoD) Advanced Research Project Agency (ARPA). The focus of this agency was on computer research, initiated by "U.S.'s reaction to the then Soviet Union's launch of *Sputnik* in 1957" [96]. Through the developments made by ARPA, the idea of an interconnected community was born, leading to the creation of the ARPANET (Advanced Research Projects Agency Network) in 1968. Initially the ARPANET consisted of IMPs (Interface Message Processors) connected at UCLA (University of California Los Angeles) and SRI (Stanford Research Institute) [96]. Once installed, group meetings were held to discuss technical issues surrounding the interconnection of these cites in 1969. This group became know as the Network Working Group (NWG) [96], which distributed meeting notes to keep members informed. These notes, known as RFC's (Request For Comments), were made public in order to promote an informal discussion of issues that would normally not be found in papers. Through this openness, many fundamental networking problems were addressed, which lead to the construction of the global Internet found today [96].

At about the same time of ARPANET, the UNIX operating system was born. It started initially in 1965 as the MULTICS (Multiplexed Operating and Computing System) project, to build a dependable timesharing operating system through the collaboration of GE (General Electric), MIT (Massachusetts Institute for Technology) and Bell Laboratories, which failed in 1969 [97]. Through the efforts of Ken Thompson, Dennis Ritchie, Doug McIlroy, and J. F. Ossanna, Bell Laboratories re-looked at the problem and developed a philosophy of using smaller interconnected programs to construct the operating system, leading to the creation of UNICS (UNiplexed Information and Computing System) [98]. Although this operating system was designed with the intent of creating an environment in which to do future work, its name stuck, leading to the shortened form of UNIX. Further research conducted at Bell Labs lead to many innovations, such as the B programming language (1971), the C language (1973) and the concept of a *pipe* (1972) – all of which aided the integration of applications and the production

of network device drivers [4]. Then in 1976-77 Ken Thompson took a six-month sabbatical from Bell Labs to teach UNIX, as a visiting computer science professor at UCB (University of California at Berkeley) [99]. On completion of his sabbatical, students and professors of Berkeley continued to work on UNIX, leading to the concept of *sockets* and the construction of the Berkeley System Distribution (known simply as BSD Unix) in 1978. Later, AT&T introduced the notion of *streams* through the creation of a "rival" Unix-based operating system (known as System V) in 1984. However, due to shortages in RAM, high microcomputer costs and lack of available networking specifications, much of the networking work developed in the 1960's only became feasible to the common user in the mid-1990s [4]. This was due to many factors [4], including the creation of Linus Torvalds' Linux operating system (1991) and the establishment of the GNU project (1984), which supported the distribution of free software to the public domain. The Unix-based Linux operating system was the first fully standard-compliant OS to feature a TCP/IP stack implementation that accompanied free source code to anyone who wished to obtain it [4]. Through this distribution concept, many research groups have gained hands-on "working documentation" that has been vital to the progress of protocols within all areas of the network domain.

3.5 The Linux Networking Model

While engineers of ARPA worked on the development of the networking framework, other organizations constructed generalized conceptual models of its' infrastructure. One such body was the International Organization for Standardization (ISO), which produced the Open Systems Interconnection (OSI) model in 1984 [100], as given in Figure 43, below.

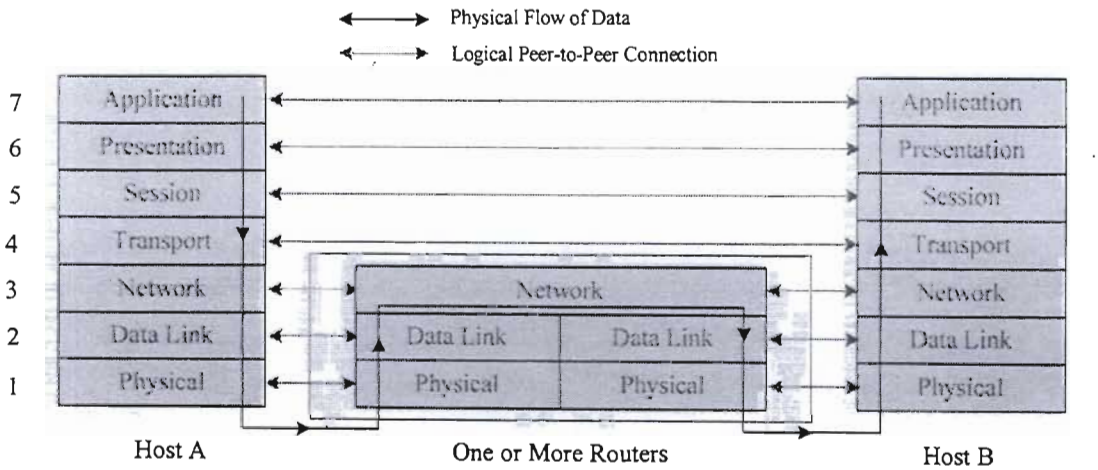


Figure 43: Showing how data flows between two host communication devices, using the seven layered stack of the OSI model developed by ISO.

Apart from providing a common design metalanguage [4], the OSI model introduced the concept of layers, which was used to break the complex networking system up into smaller, more manageable blocks. Once segmented, requirements could be established and assigned to each block, allowing developers to focus on a particular layer independently. Hence, the aim of the OSI model is not to specify the details of each layer, but rather its' task, thus leaving the implementation specifics up to the research communities at large [4]. However, in order for each layer to work independently, a process known as encapsulation was required.

Encapsulation allows two peer layers (see Figure 43) to convey control information to each other, without affecting the layer below. It achieves this through the placement of a header (and/or trailer) [100], which is appended to the beginning (or end) of each packet, just prior to it being passed down to the layer below it. Since each layer is independent, this lower layer will simply treat both the header (and/or trailer) and data fields as ordinary data, ensuring that this new data combination is delivered correctly to its peer layer. An illustration of this is given in Figure 44, below.

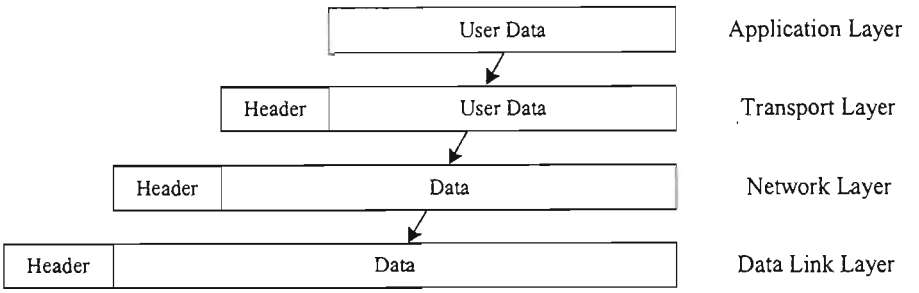


Figure 44: Showing the encapsulation process, in which the header and data fields from one layer form the "data" of the layer below it. [4]

Once the packet has been fully encapsulated, it is transmitted over a physical transportation medium, which may be a set of wires, optic-fibers, or air. On the receiving end, the packet undergoes decapsulation, which is simply the reverse of the encapsulation process. In other words, the Data Link layer will first remove its control header information, before passing the remaining data packet to the Networking layer. The Network layer will then read this data to retrieve relevant fields from its' header, which are used to determine whether the packet is destined for this machine or not. If not, the packet is re-encapsulated and sent down to the Data Link layer, so that it may be re-transmitted on towards the next-hop node. However, if the packet did reach its' destination, the header of the Network layer is removed and the resultant packet given to the Transport layer. This process then continues, until the packet is finally received by the corresponding peer application.

However, when looking at real networking systems of today, one will notice that not all seven layers of the OSI model are ever implemented. The reason for this is simply because encapsulation of seven layers is unnecessary, since it would produce a significant amount of processing and control overhead. Hence, a more efficient approach is to just to combine some of the layers within the OSI model. This is exactly what the National Research Council (NRC) did in 1994 [101], with the development of their Open Data Network (ODN) model (Figure 45).

[4] Note, however, that only four layers are represented and not all seven, as found in the OSI model. The reason for this will become clearer, with the development of the ODN model.

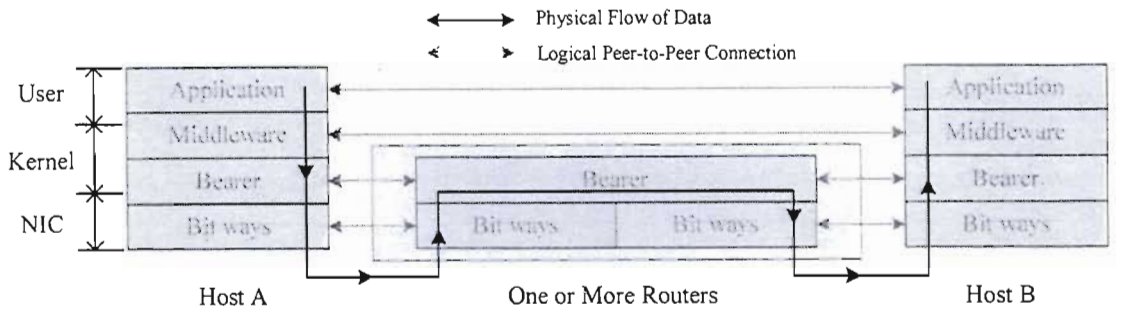


Figure 45: Showing how data flows between two host communication devices, using the four layered stack of the ODN model developed by the NRC.

Apart from reducing overhead, the ODN model was developed to give a better representation of how any network may be interconnected. This was done to help understand how all networks function, in the hope of merging all these systems into the next generation of networks. Hence, this model can be used to represent telephone, satellite, cellular, television, and computer networks [6]. In particular, this model can be used to help understand the Linux networking protocol stack. But, before doing so, a short description of each layer (or service) is necessary.

3.5.1 The Bit Way Service

The Bit Way service is responsible for transmitting individual bits over a point-to-point link [6]. Knowledge of this service is needed, so that an understanding can be gained of the transmission difference between a unicast and broadcast packet. This is because this information becomes pertinent in Chapter 5, where an explanation is given on why additional structures were required to acknowledge multicast traffic. In addition, this service explains why large latencies can be observed in congested networks.

Generally, on computer networks, the Bit Way service is handled using Network Interface Cards (NIC's), which are employed with a particular MAC (Medium Access Controller) specification. In the case of this project, Cisco's wireless Aironet 350 PCMCIA PC LAN cards, operating on the IEEE 802.11(b) protocol, were used.

The IEEE 802.11(b) wireless MAC protocol [102, 103] defines a means for location-independent devices, such as PDA's or laptops, to communicate over an unlicensed 2.4 GHz ISM (Industrial, Scientific and Medical) radio band. On traditional wired interfaces, the IEEE 802.3 Ethernet standard [104] was developed, which used CSMA/CD (Carrier Sense Multiple Access with Collision Detection). However, since the *hidden terminal* problem (section 2.3.5.1) makes wireless collision detection difficult, the IEEE modified its CSMA/CD mechanism to that of CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance), referred to as the DCF (Distributed Coordination Function). The DCF is comprised of a random back-off algorithm, virtual and physical carrier-sensing, as well as, two different transmission techniques [102]. The sub-sections that follow look into these schemes, separately, since they become relevant to the understanding of the *broadcast storm problem* [76] and why multicast protocols need to handle acknowledgement procedures within the routing layer, in order to match a unicast transmissions' reliability.

3.5.1.1 The Back-off Timer

The back-off algorithm is employed to prevent collisions from occurring, once the air medium switches from the *busy* to the *idle* state. This is because all other awaiting hosts will sense that the transmission medium has become vacant and hence will attempt to contend for the air channel, simultaneously. However, such contentions can be avoided if a host first waits for a random amount of time before it transmits. This is the task of the back-off timer.

The IEEE 802.11 back-off timer work as follows: When a host detects a *busy* to *idle* state transition, it is required to wait for a period defined by Equation 3 (unless the back-off time was previously set, in which case the wait period is set to the time remaining on the back-off timer). Once a value for the back-off timer has been allocated, the timer will decrement by `aSlotTime` for every back-off slot found to be idle [102] (see Figure 47).

$$\text{Back-off Time} = \text{Random}() \times \text{aSlotTime} \qquad (3)$$

where:

- `Random()` is a function that returns a pseudorandom integer in the interval [0, `CW`], with `CW` (Contention Window) being in the range `aCWmin` < `CW` < `aCWmax`. When a host first tries to contend for the channel, it sets `CW` equal to `aCWmin`. `CW` is then increased to the next power of two (minus 1) each time an unsuccessful data transmission occurs, until `aCWmax` is reached, in which case `CW` is no longer increased (see Figure 46). `CW` remains set to this higher value, until it is reset due to a successful transmission.

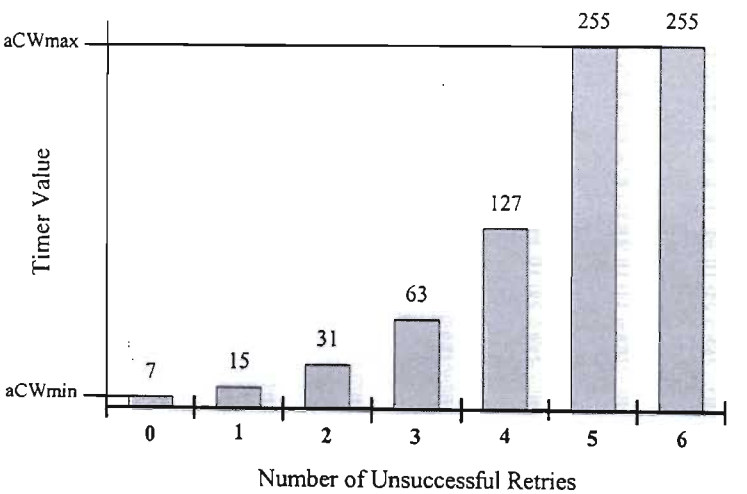


Figure 46: Showing how CW increases, with each unsuccessful retry [102].

- `aSlotTime` is the minimum time it takes for the NIC to determine the state of the wireless medium, i.e. whether the channel is *idle* or *busy*. Vendors usually set `aSlotTime` to 20 micro-seconds.

Note that as the channel becomes congested, an exponential increase in the latency of a packet is observed. This information becomes vital to the understanding of the maximum delay bound of a packet during broadcasting, as will be seen in the results of Figure 70 in Chapter 5.

To illustrate Equation 3, image two nodes exist (*a* & *b*), each initially containing a back-off time interval of 15 x `aSlotTime` and 20 x `aSlotTime`, respectively. Node *a* will transmit its' data first, since it is required to wait for a smaller time period than *b*. Hence, after 15 x `aSlotTime`'s, *a* will begin to transfer its' data, while *b* will stay *idle* with five slots still remaining, since it will sense the air medium as being *busy* during this time period. Once *a* has completed its' data transfer, the channel will switch from the *busy* to the *idle* state, causing *a* to randomly select a new back-off time interval. However, as *b* already has 5 x `aSlotTime` seconds left on its' clock, it does not select a new back-off interval, causing *b* to transmit its' data next, depending on the random value chosen by *a*. Hence, this strategy allows multiple devices to share the same physical channel, but causes periods where no data is communicated; resulting in lower link capacities, as described in Section 1.1. This process is shown in Figure 47, below.

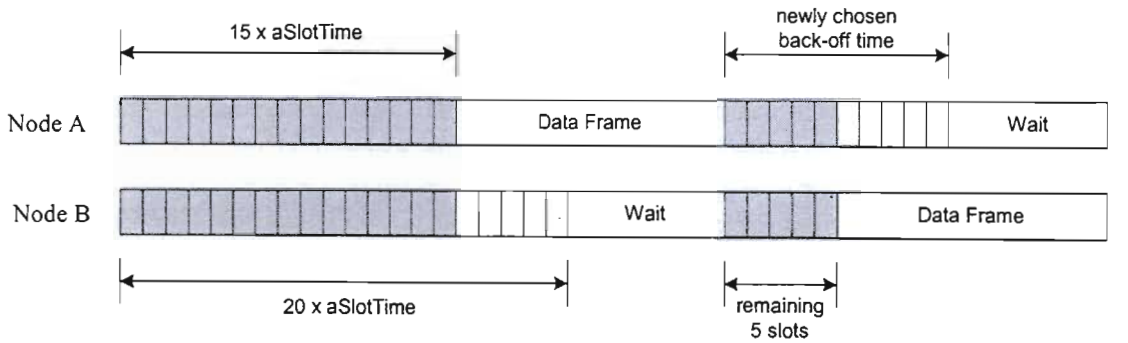


Figure 47: Showing the back-off mechanism of two nodes *a* and *b*, using the IEEE 802.11(b) MAC protocol.

3.5.1.2 Virtual Carrier-Sensing

Virtual carrier-sensing is achieved through the use of RTS (Request To Send) and CTS (Clear To Send) reservation frames, known collectively as the NAV (Network Allocation Vector) [102]. RTS/CTS control frames indicate to other neighbouring hosts that the medium is to be reserved for a duration of time, which is calculated to be the sum of the time that it takes for the transmitting host to send a data frame and to receive an ACK (Acknowledgement). Hence, hosts awaiting the use of the medium are informed to delay their transmission for this period of time. However, this reservation mechanism can only be employed for unicasts, as broadcast and multicast transmissions contain multiple next-hop destinations, resulting in multiple CTS response frames. Since the MAC layer has no way of determining whether all the CTS frames were received correctly (as there is no interaction between it and the routing layer), this reservation procedure cannot be used in multi-destination transmissions. Similarly, ACK frames can only be used for unicasting purposes, thus reducing the reliability of broadcast and multicast data (see Section 2.3.5.1). An illustration of the different frame structures employed for unicast and broadcast data types are given in Figure 48 and Figure 49, respectively.

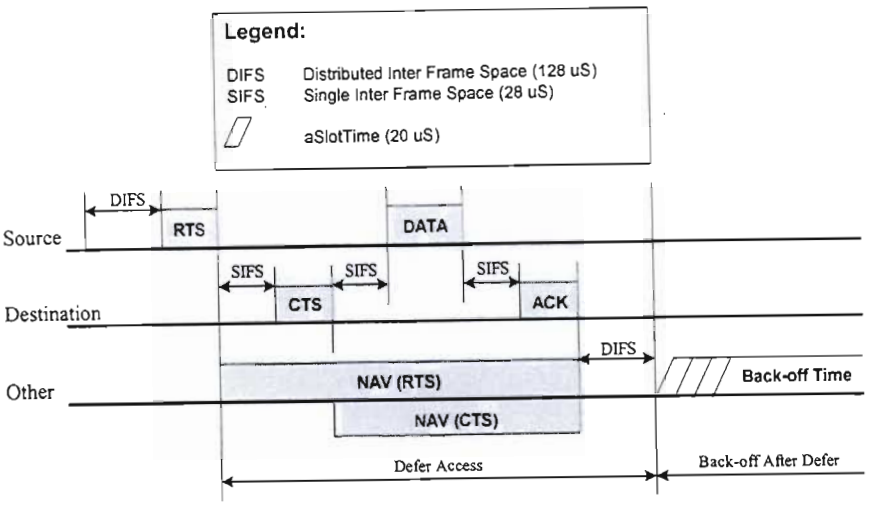


Figure 48: Showing the virtual carrier-sensing mechanism used for unicasted traffic [102].

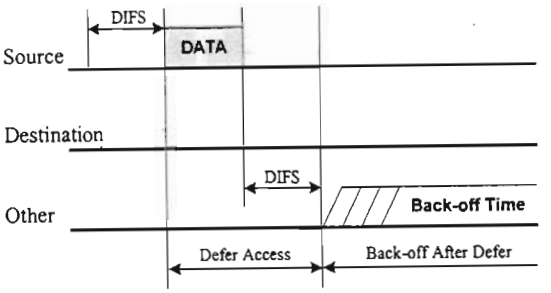


Figure 49: Showing the virtual carrier-sensing mechanism used for multicasted traffic [102].

Hence, in order for a multicast protocol to attain the same reliability as unicasted traffic, these acknowledgement sequences are required to be handled within the routing layer, as was done in Chapter 5.

3.5.1.3 Physical Carrier-Sensing

Physical carrier-sensing is accomplished through the CCA (Clear Channel Assessment) mechanism [102]. The CCA varies depending on the transmission mode used, which may be either FHSS (Frequency Hop Spread Spectrum) or DSSS (Direct Sequence Spread Spectrum). But, basically, if the carrier signal is sensed above a certain energy level, the medium is assumed to be *busy*, or else it is *idle*. Through the combination of the NAV, the CCA and the NIC's transmitter status, the state of the medium can be determined. However, for more information in this regard, refer to [102].

3.5.2 The Bearer Service

The Bearer service is required to handle the networking aspects of the system [6]. Hence, this service resembles the Network layer of the OSI model. In ordinary computer networks, this is generally handled by the Internet Protocol (IP), which is implemented within the kernel.

The main task of IP is to identify the logical source and destination addresses of the sending and receiving hosts, respectively [38]. However, in addition to these services, IP is also used to limit routing loops and to provide fragmentation of packets greater than 64K bytes. But, due to the way in which the new routing algorithm was inserted into the Linux operating system (described in Section 3.7), IP fragmentation was not permitted and hence is not discussed further. The 20-byte header of IP (used during the encapsulation process) is shown in Figure 50.

Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
TTL		Protocol	Header Checksum	
Source Address				
Destination Address				

Figure 50: Showing the IP protocol header [38].

The following is a brief description of the IP fields that were manipulated to incorporate LAMP:

`IHL (4 bits) -` The IP-Header-Length in 32-bit *words*. This value depends on whether IP options are used or not. Since this project does not explore the use of IP options, this value is fixed to 5, indicating 20 bytes (5 x 32-bits).

`Total Length (16 bits) -` The size (in octets) of the IP header + encapsulated data.

`TTL (8 bits) -` The Time-To-Live. This value is used to prevent infinite loops from occurring. It is set to the worst possible number of hops by the source, after which it is decremented by one at each router. If a value of zero is found, the packet is discarded.

`Protocol (8 bits) -` This field is used to indicate the receiving layer 4 protocol. This value is 17 for UDP and 6 for TCP.

`Header Checksum (16 bits) -` The 16-bit one's complement of the IP header. During checksum calculations, this field is set to zero.

`Source Address (32 bits) -` The logical address of the sending host.

`Destination Address (32 bits) -` The logical address of the destination host. If this value is set to all 1's, then a broadcast is sent out.

However, before concluding this sub-section, a brief description of how ARM Linux handles multi-byte (more than 8 bit) fields is required.

When individual characters (single bytes) are transmitted, they are sent out in a left-to-right fashion, starting from the top of Figure 50. This format is called network-byte-order, but is also known as *forward* or *big-endian* orientation [4]. Here the most significant 8-bits of multi-byte variables are placed in the lowest memory addresses. However, not all processors use multi-byte variables in this manner, especially those based on the Intel 80x86 architecture. These, instead, place the least significant 8-bits in the lowest memory addresses. This format is called host-byte-order, but is also known as *backward* or *little-endian* [4]. Hence, any processor operating in host-byte-order (such as the Intel SA 11xx) will need to convert variables between these formats, when manipulating multi-byes fields received in network-byte-order. However, Linux provides the following four library functions to perform these conversions automatically [4]:

`htonl()` – Host- to network-byte-order conversion of a 32-bit (4 byte) variable.
`htons()` – Host- to network-byte-order conversion of a 16-bit (2 byte) variable.
`ntohl()` – Network- to host-byte-order conversion of a 32-bit (4 byte) variable.
`ntohs()` – Network- to host-byte-order conversion of a 16-bit (2 byte) variable.

In addition to byte orientation, some processors also require that multi-byte variables be aligned to word (4 byte) boundaries [4]. One such processor is the Intel SA 11xx, used within this project. It prevents multi-byte variables from being manipulated at arbitrary boundaries. Hence, de-referencing a multi-byte variable at a non-word boundary causes undesired results. This is shown in Figure 51.

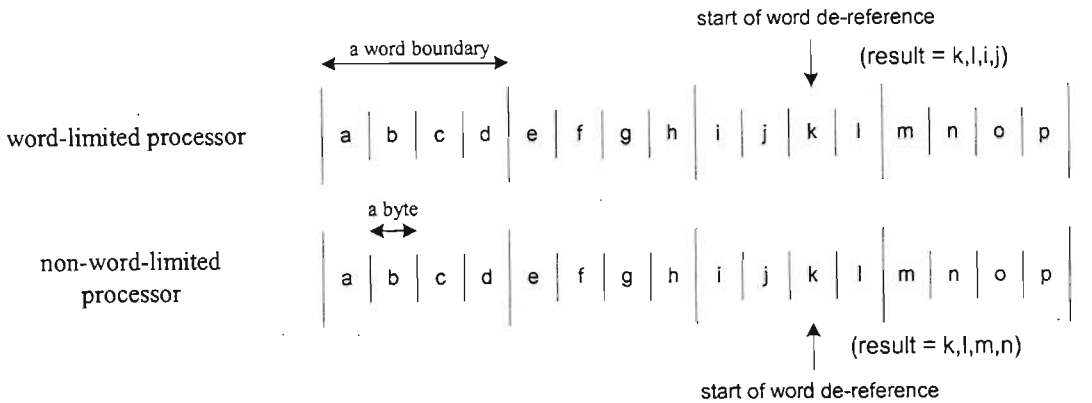


Figure 51: Showing the difference between a word-limited processor and a non-word-limited processor.

One means of preventing such errors, is to copy each desired byte into a temporary word-aligned memory space, before de-referencing. This way, data will always be recovered correctly, but will result in an increased processing overhead.

3.5.3 The Middleware Service

The Middleware service is responsible for handling any other kernel tasks that may be necessary, in order for data to be transmitted correctly [6]. In Linux, this layer constitutes the socket and INET (Internet) mechanisms. Sockets are used to handle multiple instances of applications, allowing the networking structures to be multiplexed among different kernel processes, while INET protocols are employed to ensure data is received correctly. In this project, this is handled by the User Datagram Protocol (UDP).

UDP allows datagrams (connectionless data) to be sent to/from multiple applications [105]. But, UDP does not provide protection from duplication, nor does it guarantee that transmitted datagrams will be received by the intended destination(s). If these requirements are needed, then either the end-to-end application will have to handle these procedures or UDP should be replaced by TCP (Transmission Control Protocol). However, the use of TCP is not considered for two reasons. Firstly, TCP can only provide unicast services and thus does not permit the use of broadcasting or multicasting, which is crucial to the research conducted herein. And secondly, evaluation of a routing protocol cannot be conducted through the use of TCP, since TCP will improve the delivery ratio of packets (due to its reliable protocol engine), giving a false reflection of the actual performance of the underlining routing algorithm, when viewed from an end-to-end application perspective. The 8-byte protocol header of UDP is given in Figure 52.

Source Port	Destination Port
Length	Checksum

Figure 52: Showing the UDP protocol header [105].

The field descriptions that were manipulated to incorporate LAMP include:

`Source Port` - The address of the port to which the process of the sending application is attached.

`Destination Port` - The 16-bit port address of the receiving application process.

`Length` - The size (in octets) of the UDP header + the application data.

`Checksum` - The 16-bit one's complement of the pseudo header (known also as the *fake* header, see Figure 53) and user data. Note that if this value is calculated to be zero, all 1's are sent instead, since a zero checksum indicates that no checksumming was performed by the transmitter (used mainly for debugging purposes).

Source IP Address		
Destination IP Address		
Padded Zero's	Protocol field from the IP header	Length field from the UDP header

Figure 53: Showing the pseudo (*fake*) header [105].

3.5.4 The Application Service

The Application service simply provides the GUI, allowing users to easily interact with each other. Since the iPAQ contains a relatively small screen (240 x 320 pixels), GUI's need to be designed carefully and ergonomically. The area (in Linux) where applications reside is known as user-space or (alternately) user-land.

3.6 The Linux Networking Protocol Stack

This section describes the way packets are sent over the Linux IP stack, starting with the receiver. Prior to doing so, however, an illustration of the Linux encapsulation process is required. Note that Figure 54 resembles very closely to that depicted by the ODN model.

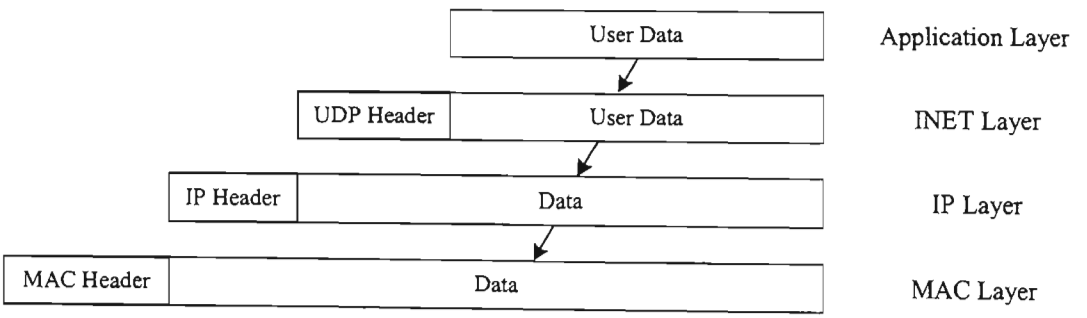


Figure 54: Showing the encapsulation process used within the Linux kernel.

3.6.1 The Reception of a Packet within the Linux Kernel

With reference to [106], when a packet is received by the NIC, the first six bytes are examined to determine the packets' destination MAC address. If this MAC address corresponds to either the broadcast MAC address (FF:FF:FF:FF:FF:FF) or the uniquely assigned MAC address of the NIC, it is passed to the network driver, or else the packet is ignored and no further processing is done to it. Since many routing layer protocols may exist on the host, it is the duty of the network driver to determine which receiving function is responsible for handling the packet next. This is done by analyzing the *type* field of the MAC header, which is generally set to IP. Assuming this is the case; the protocol receiving function will determine where to send the packet next, by examining the destination IP address. If this IP address happens to match that of the hosts' IP address, the payload of the packet is sent to either the UDP or the TCP protocol receiving functions, where it is given to the intended application through various socket dependent handling functions; else the forwarding route of a packet is looked up in the routing table of the host, where it is sent back down the IP stack for re-transmission. Note, however, that as the packet travels up the IP stack, more and more information regarding it is discarded. Furthermore, if the application does not open the correct socket type with corresponding parameters, the received packet is dropped by the operating system.

3.6.2 The Transmission of a packet within the Linux Kernel

A host may also wish to send data to another host. Since Linux performs most I/O handling through the basic file operations `open()`, `read()`, `write()` and `close()` [107], the host application begins the transmission of data by making a system call to the kernel, with the required file descriptor. File descriptors are simply integers that associate one open "file" from another, where a "file" may be a FIFO buffer, a network connection, a *pipe*, a *terminal* or even an actual file open on the disk-drive [108]. Hence, the first task of the kernel is to determine the type of "file" requested, and call the appropriate handling function. Since we wish to send data to a network connection, this request is given to the socket transmission function, which transfers the packet to the INET layer [109]. Depending on whether the file descriptor corresponded to a connection (*stream*) or connectionless orientated socket [108], the TCP or the UDP protocol transmitting functions are called, respectively. These protocols perform the necessary operations to ensure that the data is given to the receiving application correctly, after which the packet is given to the IP layer for routing. This layer fills in the source address information and looks up the outgoing route of the packet, through the hosts' routing table. Once a route is found, the next-hop MAC address is determined using ARP (Address Resolution Protocol) and the packet is transferred to the NIC for transmission.

3.7 The *Netfilter* Framework

Paul "Rusty" Russell started a project called *netfilter* in 1998 [107] to provide a packet manipulation (known as mangling) framework for the 2.3.x Linux operating system. The aim of the framework is to allow complex packet filtering (known as firewalls) and modification (called Network Address Translation, or simply NAT), outside of the of ordinary Berkeley socket interface. Thus, *netfilter* offers a means for packets to be altered (or stopped altogether), in addition to the ordinary functioning of the Linux IP stack.

Previously, mangling of packets was done using the Linux's 2.2.x *ipchain* framework [107], which was also developed by Rusty Russell. It consisted of a portion of code that resided within the Linux operating system, as well as, a command-line utility (called *ipchains*), which provided the rules to configure a Linux firewall, allowing unwanted packets to be dropped. However, no application programming interface (API) was developed to access the *ipchain* infrastructure from user-space applications, preventing rules to be written from languages such as Tcl, Java, Perl, C or C++ [107]. Also, *ipchains* was not flexible, since rules could not be established on a per-user basis nor could packets be filtered at the MAC level. In addition, Rusty Russell had also developed another independent project, known as NAT, which was used to alter the destination (DNAT) and/or source (SNAT) addresses of a packet [107]. NAT allowed one to perform many interesting "tricks", since both routers and hosts could now be "fooled" into believing packets originated from PC's which did, or did not, exist. One main application of this technique is in *masquerading*, which allowed different PC's to connect to the Internet through the use of a single ISP (Internet Service Provider) address (see [110]). Hence, an individual applies (and pays) for one connection, but is able to use this connection to run a small network of PC's, simply by using NAT on both the incoming and outgoing Internet packets. However, Rusty felt that a more flexible API was required, which incorporated both NAT and *ipchains* into a single package. Hence, for these and many other reasons, he wrote *netfilter* [107].

Netfilter consists of protocol specific hooks, which are placed strategically within the IP stack. To-date, hooks have been developed for IPX, IPv4 and IPv6. However, since this project deals with IPv4, only details relating to this protocol will be considered. A conceptual illustration of where these hooks are placed is given in Figure 55.

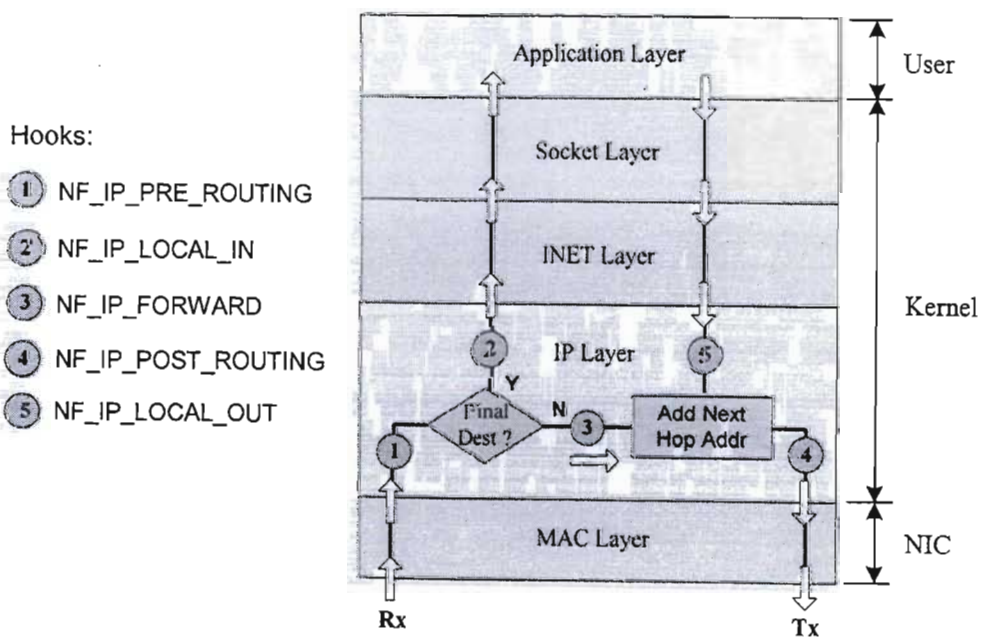


Figure 55: Showing the position of the five IPv4 *netfilter* hooks, found within the Linux kernel.

The definitions and responses for each hook can be found in `/include/linux/netfilter_ipv4.h`, which is located off the main Linux 2.4.x source directory. Essentially, when a packet is received from the NIC, *netfilter* allows the packet to be first captured by the NF_IP_PRE_ROUTING hook. Rules can then be applied to a packet, allowing a response to be generated. The responses that exist for an IPv4 packet are [107]:

- NF_DROP (which causes a packet to be dropped, thus preventing it from traversing the IP stack any further)
- NF_ACCEPT (which causes a packet to continue to traverse the IP stack)
- NF_STOLEN (which causes a packet to be taken over by the hook and hence will be processed without the use of the IP stack)
- NF_QUEUE (which causes a packet to be placed onto a protocol specific queued, so that it may be transported in a user-space application)

Hence, the NF_DROP and NF_ACCEPT rules provide a means for a firewall to be constructed, while the NF_QUEUE rule allows packets to be analyzed via a background user-space application (known as a routing daemon). An illustration of this is given in Figure 56, below.

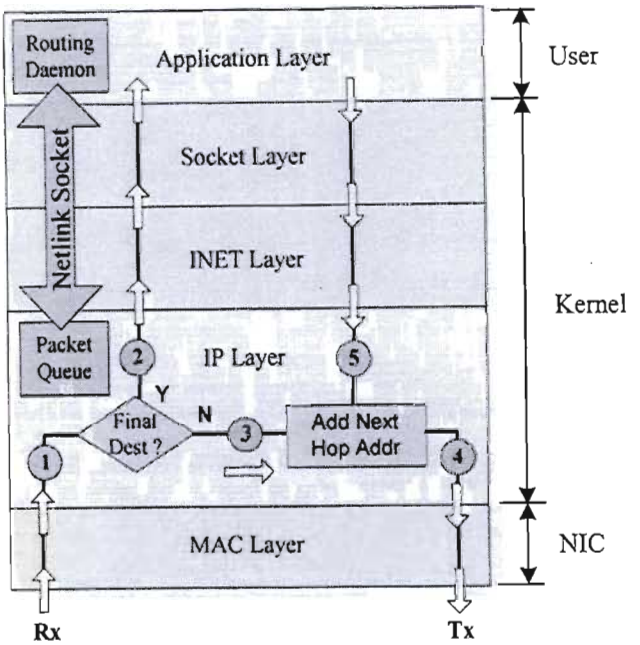


Figure 56: Showing the queuing mechanism used by *netfilter*.

Once, the routing daemon has processed the packet, it may do one of three things to the packet [107]:

- It may NF_DROP the packet, causing the packet to be flushed off the protocol specific queue,
- Or it may NF_ACCEPT the packet, causing the packet to be placed back within the hook from which it was captured,
- Or it may give the protocol specific queue a replacement packet, which is also placed back into the hook from which the original packet originated.

Hence, NF_QUEUE allows flexibility to be introduced into *netfilter*, since the user-space application is free mangle the received packet as it sees fit, before it is passed back to the Linux IP stack for processing. Thus, it is here where NAT can take place.

However, this project used the routing daemon for a different purpose. Instead of performing NAT, the routing daemon was used to manipulate packets based on the ad-hoc protocol employed. To this end, the daemon was used to add, edit and remove an ad-hoc specific header, which was employed to aid the decision making of each router.

Once a packet has passed through the NF_IP_PRE_ROUTING hook, it is examined to see whether the packet was destined for this host. If it was, it is sent via the NF_IP_LOCAL_IN

hook to the receiving application; otherwise the packet is given to the NF_IP_FORWARD hook, before being routed. Routing in Linux simply means looking up the next-hop IP address from the routing table of the local host. Once this is known, the packet is given to the NF_IP_POST_ROUTING hook, where it is sent to the NIC for transmission.

If the packet was created locally, however, it is first sent through the NF_IP_LOCAL_OUT hook. Once returned, the packet is routed and then sent via the NF_IP_POST_ROUTING hook, for transmission.

Even though this framework was used to manipulate packets for the purpose of developing an ad-hoc networking infrastructure, specific details concerning the IP stack is still required, as this will clarify exactly how packets need to be handled within the routing daemon. For example, do the UDP and IP checksums and data lengths need to be re-adjusted after a packet is altered or do these fields get re-calculated automatically after each hook? These questions can only be answered by conducting a trace through the Linux IP stack, since such documentation could not be found. Hence, this work was conducted in conjunction with the implementation and can be found in Appendix A.

3.8 Conclusion

The iPAQ PDA from HP is a high-end embedded device that is suited to the ARM Linux operating system. The Linux operating system is open, flexible and free, making it the perfect environment for the incorporation of a new routing protocol.

To-date many projects are being initiated to give the Linux kernel extra features. Companies such as Century Software, Trolltech and Transvirtual Technology, have looked to develop embedded GUI's, while projects such as *ipchains*, NAT and *netfilter*, have explored packet manipulation techniques for the kernel.

In particular, *netfilter* provides a means for ad-hoc protocols to be incorporated into the existing Linux IP stack, via a background user-space application. Being an application, routing daemons have access to ordinary programming libraries and hence are easy to both compile and debug. In addition, since packets can be manipulated before the IP and UDP calling functions, ad-hoc protocol specific headers can be add/removed without affecting the existing function calls of the IP stack. Hence, ad-hoc "features" can be simply added to the existing Linux architecture.

However, apart from the architecture, this chapter also described the difference between a unicast and a broadcast transmission. In a unicast transmission, virtual-carrier sensing is used to reserve the channel for a data transfer, after which an acknowledgement packet is sent to notify the transmitter of a successful reception. On the other hand, a broadcast packet simply uses physical sensing to detect if the channel is *idle*, after which the data is transmitted. Since no acknowledgements are used for a broadcast, the transmitter has no way of determining whether the data was successfully received by the intended neighbouring node. Hence, in order for a broken link to be detected (i.e. the loss of a neighbouring node, due to mobility), additional structures are required by multicast schemes to acknowledge the reception of packets. In addition, note that since virtual-carrier sensing is not employed, broadcast schemes suffer from a greater probability that a collision will occur, due to the *hidden terminal problem* (described in 2.3.5.1). However, this discussion will re-surface in Chapter 5, where details are given on the acknowledgement structures that were used in LAMP.

The Simulated Environment

4.1 Introduction

Developing a network protocol requires three main steps:

1. Logic construction,
2. Logic testing, and
3. Logic implementation.

However, each of these steps may need to be repeated several times, before the final protocol has been developed. For instance, during the testing phase, one may realize that some aspect of the protocol is not working as expected and thus requires modification. The same is true during implementation, except in this stage, protocol modification often occurs as a result of hardware constraints, which were not considered (or known) during the construction and testing phases [111].

This chapter explores just the testing phase and shows how a network simulator, known as *ns-2* (Network Simulator version 2) [112], can be used to provide such facilities.

4.2 The Network Simulator

Writing a custom-built simulation environment has a few disadvantages. One such drawback is the creation of existing protocol models, which have already been build (and validated) by either the author or experts working in that field. Another problem transpires when results need to be compared, since papers do not always detail every aspect of a protocol, causing different implementations to lead to different results [45]. Thus, to address these issues, a drive to develop a shared community-based simulation framework was initiated by the VINT (Virtual InterNetwork Testbed) project[5], which permits all types of networking traffic to be investigated through the use of common protocol models. VINT called this framework the Network Simulator (NS), which is now used by approximately 10,000 users from 600 institutes (across 50 countries) (see http://www.isi.edu/nsnam/ns/ns-research.html).

NS is a freely distributed, open-source simulator that focuses on modeling network protocols from the link-layer and above. NS is coded using the event-driven C++ language, which is employed to handle the transmission and reception of multiple, simultaneous discrete packets. Presently, NS is supported on UNIX (or UNIX-like) environments (such as FreeBSD, Linux, SunOS and Solaris), as well as, on most Windows-based platforms. However, due to the open-source nature of NS, support is geared more to the UNIX-based platforms than to others.

4.3 The History of NS

NS was developed by LBL (Ernest Orlando Lawrence Berkeley National Laboratory) in 1995 through the support of DARPA (the department of Defense's Advanced Research Project Agency). Initially, NS began as variant to the 1989 REAL (REalistic And Large) network simulator [112].

[5] A collaboration between LBL, Xerox PARC (Palo Alto Research Center), UCB (University of California, Berkeley) & USC (University of Southern California) / ISI (Information Science Institute).

REAL was developed to specifically simulate a packet switched, store and forward network, so that scheduling disciplines (such as the Fair-Queuing Gateway Algorithm, First-Come-First Serve and Hierarchical Round Robin) could be compared [113]. REAL was built using the University of Columbia's NEST (NEtwork Simulation Testbed) simulation toolkit, which incorporated a centralized simulation server that could be controlled using multiple client workstations [114]. However, in 1995 the VINT project was formed to evolve NS into a generic simulation platform that could be used for both network emulation and education, in addition to research [112].

Emulation allows NS to be used in conjunction with a live network, thus permitting actual network traffic to be processed and/or generated. To-date, live packets are unable to be interpreted by the simulator and hence may only be dropped, delayed, re-ordered, or duplicated. However, work is still being conducted to resolve this issue [115].

Nevertheless, since the initiation of the VINT project, NS has been extended with satellite (by the UCB Daedelus Research Group in 1997) and wireless models (by the CMU Monarch project in 1999), making it the standard framework on which ad-hoc routing protocols are tested.

4.4 The Components of NS

NS is comprised of three main components:

- The simulator,
- The pre-processing tools, and
- The post-processing tools.

The simulator is an executable application that encompasses pre-built C++ simulation models, which are configured through various OTcl (Object-orientated Tcl) scripts. Details describing how these models interact with each other are given in section 4.4.1.

In addition to these elements, NS also requires the use of externally written scenario files that describe the way nodes move, as well as, the traffic profile of each node. This is done through various generators, which come as part of the NS distribution. Together these tasks form the pre-processing phase, described in section 4.4.2.

Once NS has processed a particular scenario, results may be gathered and analyzed through two output files. The first of these files allows the simulation to be viewed through a graphical environment known as NAM (Network AniMator), while the second provides an ASCII trace of every packet handled by the simulator. This trace file can then be used in conjunction with tools such as Awk, Perl, Tcl, and MATLAB to provide vital simulation statistics, which can used to compare protocols against each other. This task is known as the post-processing phase and is described in section 4.4.3.

4.4.1 The Architecture of the Network Simulator

NS makes use of two languages [115]:

- C++ for the pre-complied objects, and
- OTcl for the frontend.

OTcl is an objected-orientated Tcl language, which was written in C++ by the same authors of NS. The reason for developing a new language for the frontend of the simulator is due to

drawbacks found in the C++ language. C++ compiles code efficiently, but, once complied, objects are difficult to manipulate. For instance, if a variable is used to indicate the number of packets that are required to be transmitted by each node, this variable can either be hard-coded or passed as a parameter from the prompt. If hard-coded, the object will need to be re-complied each time the variable needs to be altered. If, instead, the variable was passed as a parameter on the prompt, then object re-compilation will not be required, but, since a simulation uses multiple objects (each containing multiple variables), passing parameters on the prompt is undesirable. This is the task of OTcl. Instead of passing parameters over the prompt, NS makes use of a simple scripting language to configure objects for a particular simulation. However, to achieve this, a process known as *shadowing* is required.

Section 4.4.1.1 describes the operation of selected pre-complied C++ objects, while section 4.4.1.2 explains the *shadowing* process used to configure these objects.

4.4.1.1 The C++ Objects

In NS, two main types of objects exist [115]:

- Classifiers, and
- Connectors.

Classifiers are simply multiplexers that are used to connect many objects together, while connectors are used to manipulate packet events. Through the combination of these two object types, models can be constructed. Section 4.4.1.1.1 details the wired models, while the wireless models are handled in section 4.4.1.1.2.

4.4.1.1.1 The Wired Models

By combining many classifiers and connectors together, NS is able to artificially model the networking architecture found in many physical systems, such as Linux. An example of such a model is shown in Figure 57.

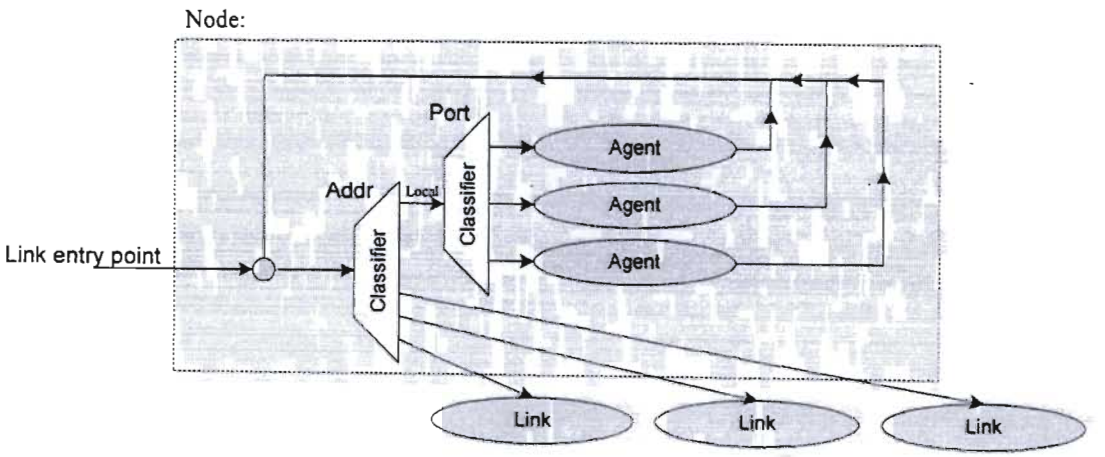


Figure 57: Showing the NS model used to represent a wired node [115].

A node in NS consists of two types of classifiers. The first of these is an address classifier. The node depicted in Figure 57 contains three other downstream links. Hence, the task of the address classifier is to examine the address of a packet to determine which link the packet is required to be transmitted over next. If this address happens to be equal to the local address of the node, the

packet is sent to the port classifier; else the packet is sent to a neighbouring node via its corresponding link.

The port classifier is used to connect multiple agents together. Agents are objects that are able to manipulate packet fields in some defined way. Hence, NS uses a combination of agents to implement the different protocols found at each layer of the ODN model (section 3.5). Some of the pre-existing agents of NS include TCP, UDP, FTP (File Transfer Protocol), CBR (Constant Bit Rate) and Telnet protocols. In addition to these, a special agent, known as a routing agent, is used to manipulate the address of the packet, allowing the address classifier (and hence the next-hop destination of the packet) to be controlled. Additionally, agents are able to send a packet from one layer to another by altering the port address of the packet, allowing inter-layer, as well as, peer-to-peer communication to be simulated.

Links, on the other hand, are used to model the congestion and delay characteristics of packets. A conceptual illustration of a link in NS is drawn in Figure 58, below.

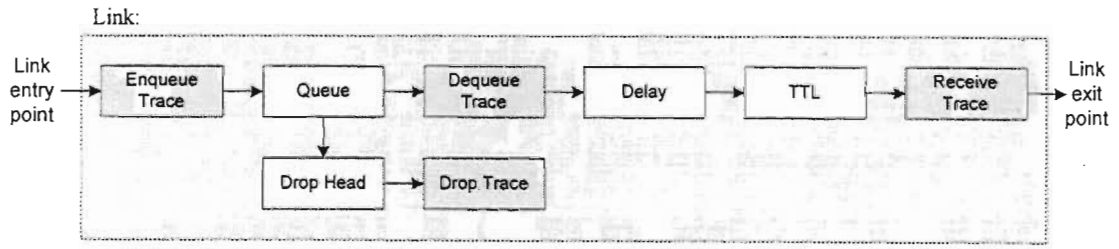


Figure 58: Showing the NS model used to represent a wired link [115].

When a packet arrives at a link, it is placed into a queue connector. However, just prior to queuing, the packet is written to an output trace file for statistical and debugging purposes (see the post-processing phase given in section 4.4.3). NS allows the use of many different queuing disciplines, such as FIFO (known in NS as a droptail queue), Priority-based droptail, CBQ (Class-Based Queues), RED (Random Early Detect), FQ (Fair Queues), SFQ (Stochastic Fair Queues, and DRR (Deficit Round-Robin). If any of these schemes cause the packet to be dropped from the queue, the packet is sent to the drop head.

Drop heads are simply objects that are allowed to manipulate the dropped packet, before it is written to the output file (through the drop trace connector). If, however, the queuing scheme caused the packet to be de-queued instead, it is first traced and then sent to the transmission delay connector.

The delay connector just multiplies the size of the packet by 1/bandwidth (of the link) to calculate the amount of time the packet is required to be delayed. Once delayed, the packet is given to the TTL (Time-To-Live) connector, to prevent routing loops.

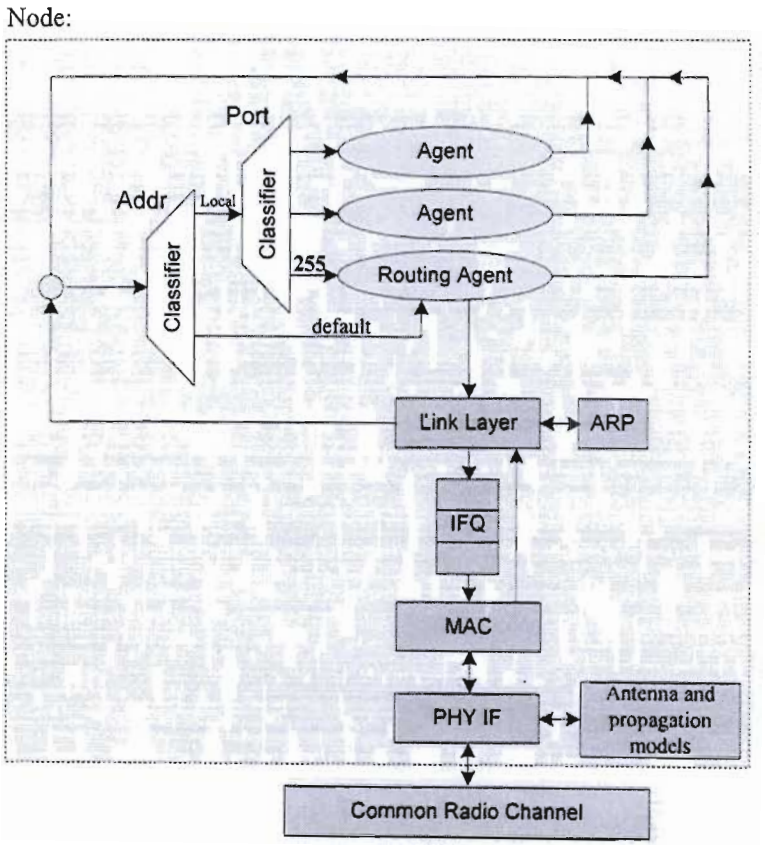
The TTL connector simply reduces the TTL field of the IP header (see 3.5.2) by one, and if found to be zero, the packet is dropped, or else it is traced (to indicate a packet reception) and delivered to the corresponding downstream node, where it will be received by the neighbouring address classifier.

However, in order to model congestion properly, a feedback mechanism is required to notify the queue when it may send the next packet. This is accomplished through the use of a *Boolean* flag, which is set to *true*, during de-queuing, and *false*, once the packet has been handled by the

delay connector. Thus, subsequent packets are blocked, while another packet is still in transient, allowing queues to (possibly) overflow and hence experience outages.

4.4.1.1.2 The Wireless Models

The wireless models of NS were developed by the Monarch project [45] as an extension to the existing wired architecture. But, to do so, [45] needed to introduce new node models, which could account for node mobility and the shared nature of radio communication. One such example is *MobileNode*, given in Figure 59.



ARP (Address Resolution Protocol)
IFQ (InterFace Queue)
MAC (Medium Access Controller)
PHY IF (PHysical InterFace)

Figure 59: Showing the *MobileNode* model used to represent a wireless node in NS [115].

As one can see, the basic structure of the NS node has been maintained. However, comparing Figure 57 to Figure 59, one notices that the *link* model has been replaced with a number of underlying layers (the darker shaded sections of Figure 59). In addition, the address classifier is no longer used to forward packets to neighbouring nodes. This task has been shifted to the routing agent, which was specified to be located on port 255. In fact, the Monarch project developed another wireless node model, called the *SRNode* (Figure 60), which removes the address classifier altogether. Since this model explicitly uses the routing agent to handle all addressing decisions, any packet de-multiplexed on port 255 (by the port classifier) is simply discarded.

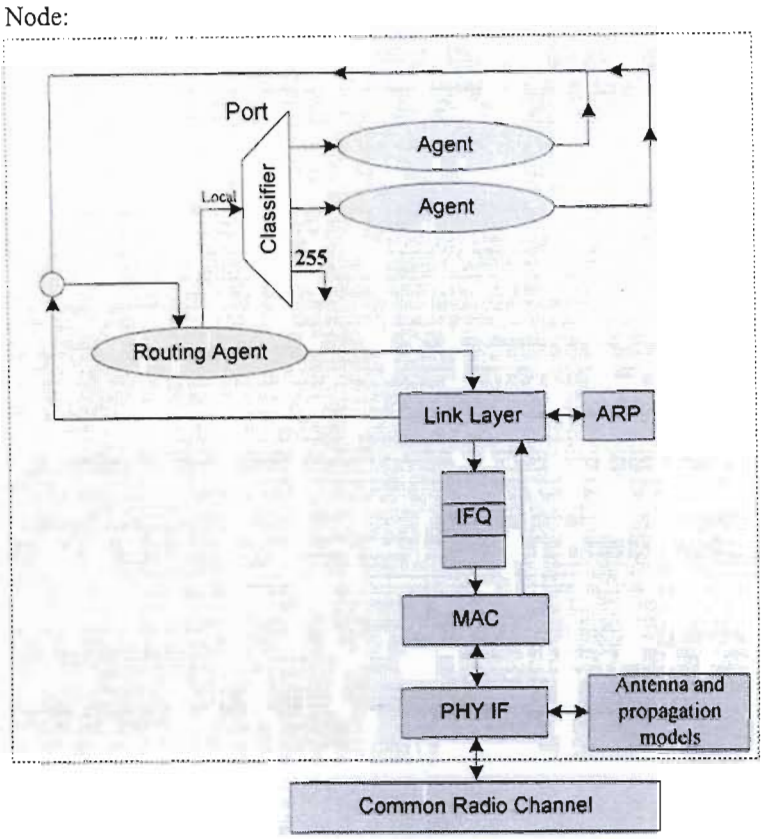


Figure 60: Showing the *SRNode* model used in NS [115].

The darker shaded objects of Figure 59 and Figure 60 model the NIC (Network Interface Card), found in computer networking systems, such as Linux (see section 3.5.1). At the top of this structure lies a link layer, which [45] based on the IEEE 802.2 Logic Link Controller [104]. Essentially, this layer is responsible for translating logical addresses to physical MAC addresses, using a BSD based ARP protocol [115]. Once an appropriate physical address has been found, the packet is placed onto an interface queue, where it is buffered for transmission.

The interface queue is mapped directly to the queue found in Figure 58 and, hence, NS allows any queuing discipline to be employed here. However, since wireless routing protocols make use of time-critical data (such a route-request or reply), interface queues are generally implemented using the Priority-based droptail buffer [115], which allows routing specific data to jump to the head of the queue. Once a packet is removed from the buffer, it is sent to the MAC layer for transmission.

[45] implemented two different MAC protocols into NS, namely:

- The IEEE 802.11(b) MAC protocol, and
- The Preamble based TDMA (Time Division Multiple Access) scheme.

The IEEE 802.11(b) MAC protocol (described in section 3.5.1) uses RTS/CTS/DATA/ACK sequences for unicasted transmissions, and only the DATA sequence for all broadcasted communication. On the other hand, the preamble based TDMA protocol [115] uses a dedicated time slot for each node and a preamble sequence to indicate the intended next-hop destination. Research is currently being conducted to avoid preamble contentions and to re-use unallocated time slots. However, since this project did not explore the use of a TDMA based system, details

concerning this protocol are left to the relevant literature. Nevertheless, in order for each sequence to be transmitted, some kind of transmission technique is required by the physical interface.

The Monarch project based the characteristics of the physical interface on the DSSS Lucent WaveLan Card, which was used to stamp packets with transmission parameters, such as power, wavelength, etc. These parameters were then applied to a unity-gain, omni-directional propagation model, in order to calculate minimum power sensed at each node (connected to the common radio channel). [45] then used this power reading to determine the effective transmission range of a node, which also enabled him to detect packet collisions.

The propagation model used by Broch et al [45] involved a combination of the Friss-space and Two-ray Ground attenuation models. The Friss-space model attenuates the power of a radio transmission at $1/r^2$ (where r is the distance between two antennas), while Two-ray Ground uses $1/r^4$. [45] found that radio engineers generally use the Friss-space model for short distances and the Two-ray Ground model for distances found to be larger than a point known as the reference distance [115]. Typically this distance is approximately 100m (see [116]), hence, the Monarch project simply switched between these two models, depending on the distance between the two nodes in question.

However, applying the above calculations to each node (for every transmission) results in high computational overhead. Hence, [45] decided to divide the area (covered by the mobile nodes during the simulation) into a number of squares, thus forming a grid (as shown in Figure 61). By superimposing a pre-calculated theoretical maximum transmission radius (passed as a parameter from the OTcl environment to the physical interface at the start of the simulation) on top of this grid, Broch et al [45] was able to quickly identify a number of squared-regions, which constitute a particular nodes transmission range (the shaded areas of Figure 61). Thus, if any nodes were found to be located in these regions, a copy of the packet (containing the transmission parameters) is forwarded to them; else the packet is discarded. Hence, the number of nodes that actually performed the relevant propagation calculations were able to be reduced, thus increasing the computational speed of NS.

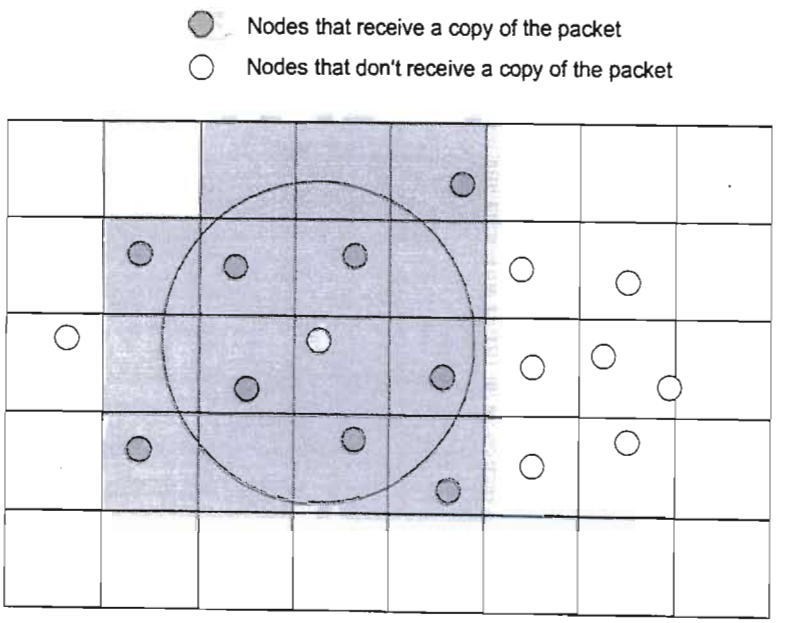


Figure 61: Showing how a grid can be used to decrease the computational complexity of NS.

4.4.1.2 The Shadowing Process

Shadowing provides a means for an object compiled in the C++ environment to be mapped to a corresponding object in the interpreted OTcl environment, so that specific code (constituting procedures and variables) can be manipulated from either environment and yet still point to the same object instance. To accomplish this, however, each environment is required to contain an identical object hierarchy, so that parent-child relationships of the interpreted object can be used to create a *shadow* of the compiled object from the complied hierarchy [115]. For example, take the situation (drawn in Figure 62) in which a DSDV routing agent is created within the OTcl environment, using the OTcl command:

```
set ragent_ [new Agent/DSDV]
```

Here, the `new` command creates a DSDV agent and places it into the "ragent_" class variable, via the Agent/DSDV class constructor. However, since DSDV is inherited from the Agent class, the DSDV constructor can only be initiated once the Agent class constructor has completed. But, since this constructor is dependent on the Connector class (which is inherited from the TclObject class), the TclObject constructor is invoked first.

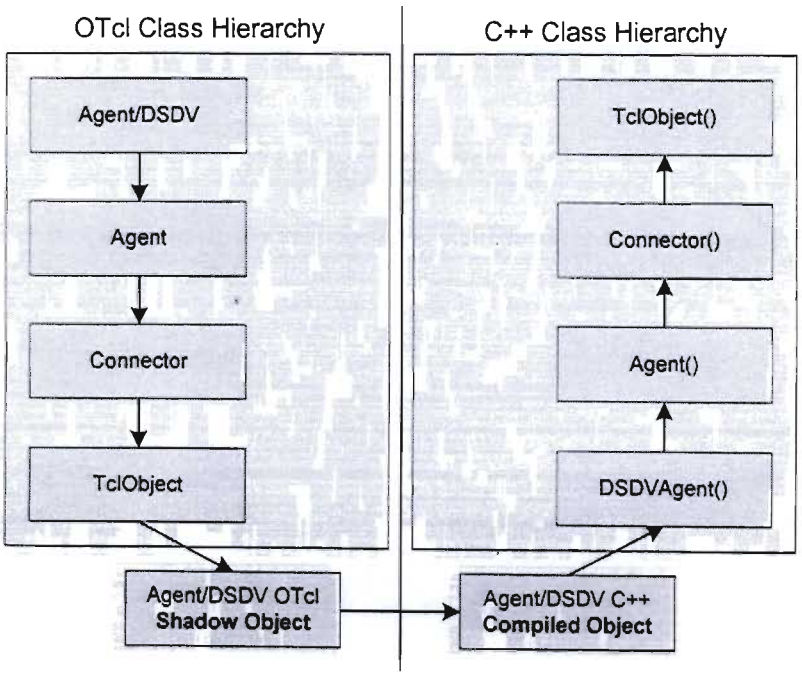


Figure 62: Showing the object hierarchy used to create a DSDV routing agent in NS.

The TclObject constructor is the constructor that is responsible for creating the Agent/DSDV shadow object from the complied Agent/DSDV code. But, in order for the C++ DSDVAgent to initialize correctly, it is required to undergo a multi-stage inheritance (from its complied class hierarchy), before returning the class object to the "ragent_" variable. However, while the C++ classes are being inherited, NS allows specific C++ variables to be bounded to the OTcl environment.

A bounded variable allows a user to manipulate its' value from OTcl, and yet be instantly reflected in the C++ code [115]. Hence, it is through this process that parameters can be passed dynamically to C++ objects. But, instead of requiring the user to specify each and every parameter to all objects, NS makes use of a default OTcl script (found in `/tcl/lib/ns-default.tcl` off the main NS directory), which passes these parameters automatically.

Hence, a user need only edit this default script (or overwrite these values from another startup script) to alter object parameters.

However, binding is more than just a means to pass parameters, since it causes a C++ variable to be mapped directly to the OTcl space. Hence, bounded variables can be monitored and/or manipulated during any stage of the simulation [115], making it easy to debug (and understand) protocols.

In addition to variable manipulation, *shadowing* also provides a means for C++ procedures to be invoked from OTcl, through the use of the `cmd` statement [115]. For example, if one wished to print the routing table of DSDV (using the C++ member function `print_table()`) at some point during the simulation, one may type the following (in OTcl):

```
$ragent_ print_table
```

However, since "print_table" is not a defined OTcl command, OTcl will issue a `cmd` function call on the Agent/DSDV shadow object, passing it an argument vector (`argv`) consisting of the following:

argv[0] – "cmd"
argv[1] – "print_table".

NS automatically maps all `cmd` calls to an objects C++ `command()` member function. Hence, to interact with OTcl, a simple check needs to be made on argv[1] to confirm that the "print_table" argument string was received by the `DSDVAgent::command()` function, before invoking `print_table()` (as given below).

```
int DSDVAgent::command(int argc, const char*const*argv) {

      if (argc == 2) {

            if (strcmp(argv[1], "print_table") == 0) {
             DSDVAgent::print_table();
                  return TCL_OK;
            }
      }

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

Thus, shadowing provides a powerful mechanism, which allows complex C++ models to be fully configured (and manipulated) through a simple scripting language (such as OTcl), without the need for re-compilation. For more information see [117].

4.4.2 The Pre-Processing Phase

In addition to configuring parameters for a particular simulation scenario, NS makes use of two external OTcl scripts to describe the mobility and traffic pattern of each wireless node. The only self generated mobility pattern (currently) supported by NS is the *random waypoint model*, while traffic may comprise of TCP and/or CBR transfers [115]. However, note that since TCP affects the performance of the underlying routing protocol (see 3.5.3), only CBR traffic (which incorporates the UDP mechanism) will be discussed.

The *random waypoint model* [40] is characterized by a parameter known as the *pause time* and is based on the following principal: Each node in the simulation begins from an arbitrary location in the simulated environment space and moves to a randomly selected destination with a speed uniformly selected between 0m/s and some maximum. Once the destination point has been reached, a node pauses for the pause time period before randomly selecting a new destination. This process is then repeated until the end of simulation has been reached.

CBR traffic, on the other hand, involves the transmission of a bit stream at a constant rate, and can be characterized by three parameters:

- The number of packets to be transmitted,
- The size (in bytes) of each packet, and
- The rate (in bits/second) of each transmission.

Thus, a CBR application (in NS) will transmit a packet every ((size) x 8) / rate) seconds, until all packets have been transmitted or the simulation has terminated.

Together, these scripts form the heart of the simulation setup and can be used to evaluate the performance of the underlying routing protocol, under many repeatable transmission scenarios.

4.4.3 The Post-Processing Phase

This phase is initiated at the conclusion of a simulation, which produces two output trace files. These files both contain a time-stamped list of all packets transmitted for each protocol layer, but differ in data they represent.

The first of these files is known as a NAM trace and is used as an input to the Tcl/TK based animation toolkit [115], which is packaged as part of the NS distribution. The animation toolkit has been built to consume as little memory as possible (due to the amount of information that is stored in the trace file) and hence makes use of a strict file format, which resembles the follows:

```
<event-type> -t <time> <more flags>...
<event-type> -t <time> <more flags>...
     :
```

Where:

`<event-type>` indicates the type of event. In wireless simulations, events usually occur as a result of node mobility or due to a packet being received, dropped or sent by a particular protocol layer.

`-t <time>` indicates the time at which the event occurred, with each line corresponding to a new time sequence. NAM also requires each line to be in chronological order, so that it need only read one line at a time, when displaying visual information.

In addition, the animator has the capability of displaying real world packet trace data and is able to convert its visual information into an animated GIF or MPEG movie. However, details regarding these abilities are beyond the scope of this thesis. Thus, for more information, consult [115].

The second trace file gives a more detailed textual list of all the events that occurred during the simulation. Since this file is not formatted for a particular application, this file may contain any

information and hence may be used to debug a protocol. However, apart from debugging, the main use of this trace is to gather statistical data, which is used to assess the performance of a particular protocol (in a particular scenario). Examples of such metrics include (based on [1]):

- The end-to-end network throughput,
- The end-to-end network delay, and
- Protocol efficiency.

End-to-end network throughput is defined as the percentage of data packets that successfully reach their destination, while end-to-end network delay measures the amount of time that elapsed during this period. Since these parameters differ for each transmission, statistical indicators (such as mean and variance) are important. Together, throughput and delay metrics provide an external measure of the data routing performance of a protocol [1].

While, data routing performance is defined as the external measure of effectiveness, efficiency is considered to be an internal measure [1]. This is because two routing protocols may achieve the same level of data routing performance, and yet contain varied amounts of protocol overhead. Hence, the efficiently may be independent to the data routing performance of a protocol, unless the control and data traffic share the same channel, in which case excessive control traffic would degrade the channel's capacity, causing a lower data routing performance to be experienced. Thus, to determine the internal efficiency of a protocol, two ratios are used:

- The average number of control bits[6] transmitted per data bit delivered, and
- The average number of control and data packets transmitted per data packet delivered.

The first ratio measures the protocol's efficiency in using control overhead to deliver data packets, while the second measures the "useful" throughput of the network [1].

Collectively, the output trace files describe the progress of each and every packet within the network, and thus provide a means for protocols to be compared, evaluated and validated (either visually or statistically), under various repeatable conditions.

4.5 Limitations of NS-2

The wireless models of NS contain three assumptions [118]:

- Transmissions are always perfectly circular and therefore are unaffected by intrusions such as buildings, cars, trees, etc.
- All nodes move along a flat surface (x and y directions only) and thus effects resulting from differing antenna heights (z) cannot be explored.
- The maximum velocity of a node is significantly smaller than the speed of light, since all models do not take into account Doppler effects, etc. In addition, the propagation model only samples the signal strength once during the packets traversal and thus may cause unexpected receptions to result, if a nodes moves significantly during the transmission period of a packet.

Apart from these, other issues need to be considered whilst using the simulator (or else problems will arise later when trying to implement the developed routing protocol in the physical environment [111]). One such problem is the use of the *common header*, which is

[6] Note that control bits not only constitute the information found in control packets, but also the header portion of all data packets. In other words, anything that is not raw user data is control overhead, and thus should be treated as such.

appended to all packets for the internal operation of NS. This header contains (among other fields) [115]:

- A unique packet ID (called a UID in NS),
- A timestamp of the packets transmission time,
- A simulated packet size,
- A receiving interface label,
- Packet type (to determine which protocol initiated the packet),
- Previous hop address,
- Next Hop Address,
- And the number of hops the packet has undergone already.

As one can see, this information is very temping to a protocol designer. However, this header is not actually recognized as being part of a packet. This is because it is used to specify the simulated packet size, which excludes itself. Hence, using any information from this header simply means you are receiving some data at zero cost, since it constitutes zero bandwidth. This of course is absurd (on practical networks) and hence protocol designers should rather duplicate (and possibly handle) any required fields from this header to their own protocol header, instead of simply accessing this header at will.

Other issues relating to simulators (in general) include positional information, time, and processor differences [111]. NS operates on a different system than that of ARM Linux (used here) and, therefore, differs substantially in processing speed, power management as well as hard-disk memory sizes and access times. In addition, NS is able to start all nodes in the network simultaneously, since only one system clock is used. This means that all nodes are always perfectly time synchronized. In reality, however, this is a very difficult task to achieve, because even if one could synchronize the clocks of all mobile devices at the start of an application, keeping them synchronized over long periods of time, becomes difficult due to clock skewing [119]. Also the time interval over which protocols function, differ from simulation to that of the real-world. Simulations only run for a few hundred seconds, while real-world applications run indefinitely. Hence simulations generally don't suffer from overflows in variables. This allows protocols to adopt a smaller number of bits for header fields (and other variables), which give better performance in terms of overhead than what is realizable in real-word implementations.

Another issue involves the way positional information is represented in NS. NS makes use of x- and y-coordinates, which are accurately available at all times. However, to make this information accessible to ARM Linux, serial GPS receivers were needed. GPS coordinates are given in degrees latitude & longitude, which (depending on the type and quality of the GPS) contain a certain amount of error and latency. In addition, the calculation of distances between two GPS coordinates is non-trivial due to the geoidal shape of the earth [120] and thus requires mathematical approximations, which, combined with GPS coordinate truncations, can lead to positional errors in the order of tens of meters. Therefore, protocols that rely on such information may experience a degradation in performance when compared to simulation [111].

Hence, great care must be taken when designing a new protocol, or else one may find oneself iterating through many unnecessary development life-cycles, before eventually succeeding with an optimal, yet realizable routing scheme.

4.6 Conclusion

NS is an event-driven packet simulator that uses two distinct programming languages to bring flexibility into its framework. Currently, NS supports wired, wireless and satellite models (as well as combinations of these configurations), through the uses of many interconnected objects. Since these objects may be accessed from both coding environments (through *shadowing*), protocols can be analyzed, compared and debugged with the greatest of ease (both visually and/or statistically).

However, like all simulators, there are limitations. This is because simulators only necessitate the formulation, testing and comparison of a protocol under various repeatable scenarios and hence are unable to represent the real world, no matter how accurate its' underlying protocol models are. Thus, for true protocol verification, implementation is essential [111].

Nevertheless, simulation provides a framework where ideas can be both tested and assessed, and thus remains crucial to the development of new protocols.

LAMP - Location Aided Multicasting Protocol

5.1 Introduction

Previous multicast protocols (as described in section 2.3) make use of one of two different forwarding concepts:

- Groups, and
- Multicast regions.

Groups allow the sender and receiver members to be independent of each other, and thus permit a host to dynamically join and/or leave a group at any stage. On the other hand, multicast regions allow data to be flooded to all hosts that happen to be located within a specific geographical area. However, both strategies suffer from the fact that a source host is unable to determine which nodes are currently connected to a particular group (or are positioned in a particular region) at a specific point in time (unless a request is sent out). Additionally, in military applications, transmitted data (which may be classified) will generally be intended for a specific set of hosts. Thus, for a real-time audio system, the distribution of data to an abstract group (or a geographical region) is undesirable.

Nevertheless, Obraczka et al [75] feels that ad-hoc networks are well suited to multicasting, since such systems make use of omni-directional antennas, which give it the ability to communicate to all neighbouring nodes simultaneously. This is different to the IP world, where the common belief is that multicast routing is built on top of the unicast framework, as the IP infrastructure consists of hard wired point-to-point connections, instead of the point-to-multipoint (radio) communication channels, found in ad-hoc networks. Thus, due to this infrastructure difference, [75] argues that ad-hoc multicasting needs to be addressed differently, and hence the use of tree forwarding (such as *source-based*, *core-based* or *mesh-based*) methodologies may not necessarily be the most favorable solution. This arises from the fact that tree algorithms require multicast state information to be stored at each node, which must be both updated and propagated every time node mobility causes a link change to occur[7]. Since most mobile devices (such as handheld devices) contain limited capacity and processing power, the cost of keeping this multicast state information (especially when 20 nodes or less are used) is intolerable. Hence, with these considerations in mind, a more effective approach was desired.

Obraczka et al [75] indicated that an adaptive flooding algorithm may provide the answer. Adaptive flooding algorithms make use of successive broadcasting, from a subset of the networks nodes (forwarding group), to route data to multicast members. Algorithms of this kind differ in the way in which this subset is chosen. Since each unit in the proposed network enjoys the benefit of a GPS, the author of LAMP decided to choose this subset through the use of location information, as defined by the second algorithm of the LAR protocol [25]. However, as mentioned in section 2.2.1.2.2, this algorithm does not guarantee that a path will be found to a destination (even if one exists), since forwarding data to nodes that are closer to the destination may result in a situation in which all neighbouring nodes are deemed further than

[7] Note that since link changes occur rarely in wired networking domains, tree based methodologies are very suited to such environments.

the transmitting node. Hence, this algorithm was modified with global time properties, as employed by the LOTAR protocol (section 2.2.2.2.7), to overcome such drawbacks.

Nevertheless, flooding every packet throughout the whole network produces a considerable amount of congestion and contention, known as the *broadcast storm problem* [76]. Thus, to reduce the number of nodes which actually forward broadcasted data, flood limiting techniques (detailed in section 2.3.5) were developed. However, as indicated by [77], these techniques require more than 20 nodes to be present. This is because a network with less than 20 nodes does not contain enough neighbouring nodes to give these techniques the opportunity to excel above *blind flooding* and hence causes a node to consume extra processing power with no additional performance gain. Thus, the only feasible option remaining in sparse networks is *blind flooding* (or through some localized flooding technique, such as DREAM [16] or LAR). Therefore, to reduce the congestion of *blind flooding*, an alternate method was needed.

Considering that LAR requires some kind of location *service* to be provided (see section 2.2.1) and the fact that the handheld device was required to display "up-to-date" position information of all neighbouring hosts, it was decided that some form of *table-driven* methodology would be appropriate. This is because *table-driven* algorithms forward data along a single path (from source to destination), instead of via multiple paths, as employed by flooding, and thus are able to reduce congestion. However, *table-driven* schemes suffer from scalability issues, since a record (or table entry) is required to be kept for every node in the network. But, since this system only requires the use of 16 nodes, scalability can be temporarily ignored, until a suitable solution has been found. This fact is what triggered the development of the *on-demand* unicast routing techniques from the *table-driven* schemes, causing AODV to be superior to DSDV [121]. However, AODV cannot be employed where an "up-to-date" location *service* is required, since such a requirement defeats its very incentive of the *on-demand* approach. In addition, *table-driven* schemes also enjoy the benefits of minimal latency[8] (when a path to a destination is known), which is necessary when time-critical data is being handled (such as that found in audio and video transmissions).

Comparing the various *table-driven* schemes together, one notices that DSDV is the most fundamental routing strategy developed, yet it is also the most efficient. This is because both CGSR and HSR require the use of a hierarchical clustering algorithm, which is triggered each time two clusterheads become neighbours or a host becomes isolated. Thus, these schemes induce a large amount of transmission and computational overhead to re-establish the routing hierarchy [34]. Likewise, WRP and GSR require the use of multiple tables, which unnecessarily consumes a large amount of memory and computational overhead. Hence, the DSDV algorithm was selected over the others to reduce the congestion incurred through flooding, since it best matched the constraints of the PCS system. However, the DSDV protocol is only a unicast routing scheme and hence its' next-hop table is unable to account for multiple neighbouring destinations. Therefore, this forwarding scheme required modification, which will be discussed

[8] *On-demand* schemes require a route to be first established (through a route discovery mechanism), before routing on the packet can be initiated. Since this requires the source and destination pairs to communicate with each other before a route can be found, *table-driven* protocols are able route packets quicker than *on-demand* schemes. However, since the NS implementation of the DSDV protocol queues all packets (for which the next-hop is unknown), the overall delay performance of the DSDV protocol is not necessarily better than the other *on-demand* methodologies. But, as will be shown later, LAMP employs a localized flooding technique to handle all packets for which the next-hop destination(s) is unknown, rendering LAMP free from such delay drawbacks.

in section 5.2. Nevertheless, the incorporation of a multicast next-hop forward strategy allows the required data to be treated as a whole, instead of a separate set of unicast transmissions.

However, in the presence of mobility, next-hop links may become broken. Hence, to detect when the next-hop table scheme fails (and hence initiate the localized flooding algorithm), a multicast acknowledgement mechanism is required. But, since the IEEE 802.11(b) MAC protocol is unable to handle such procedures (see 2.3.5.1 and 3.5.1.2), this process is required to be handled by underlying routing protocol itself, as is done by LAMP.

The reminder of this chapter proceeds as follows: Section 5.2 begins by giving a detailed description of LAMP and its implementation specific structures. Sections 5.3 and 5.4 then describe the four simulation case studies that were used to illustrate and evaluate LAMP. Finally, section 5.5 concludes by highlighting LAMP's main advantages and disadvantages.

5.2 The LAMP Protocol

LAMP makes use of an underlying destination-sequenced next-hop routing table to identify forwarding neighbours. These neighbours are then appended to the packet, which is broadcast to all adjacent hosts. Each of these hosts is then responsible for checking if its address appears as one of the entries in the received next-hop list. If found, it is expected to send a unicasted acknowledgement packet back to the previous hop and then re-generate the next-hop list for all the destinations to which it has been assigned; else the broadcast is simply ignored. This way, the number of destinations that a node is required to handle becomes less and less, as the packet propagates on towards its intended recipients. This can be seen in Figure 63.

Figure 63(a) begins by indicating the topology state of the network, as well as, the destination and next-hop lists that the source node (*S*) appended to its' raw data packet. Note that *S* obtained the next-hop information from the underlying unicast protocol. This packet structure is then broadcasted to all adjacent nodes of *S* in Figure 63 (b), causing nodes 1, 4 and 6 to receive the packet. However, as the next-hop list of *S* only addressed nodes 1 and 4, just these hosts are required to send an acknowledgement back to *S* (the darkened blocks) and re-construct the next-hop lists for the destinations they have been assigned; while node 6 simply discards the packet. Since node 1 was only allocated *D1*, it discards entries pertaining to *D2* and *D3*. Similarly, node 4 discards *D1*, causing a reduction in the required header information as the packet progresses towards the three intended destinations. This process continues in Figure 63(c) and (d), until the data of *S* is delivered to *D1*, *D2* and *D3*.

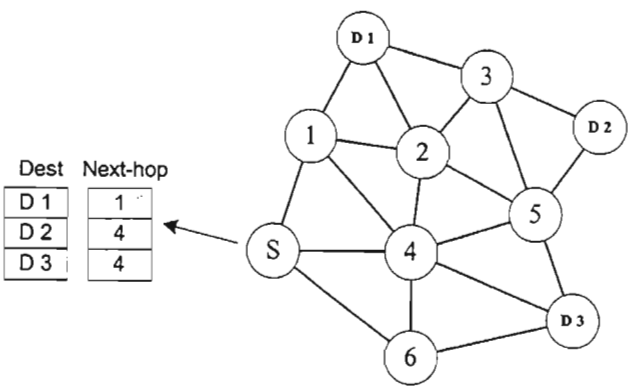


Figure 63(a): Showing how node S transmits a packet simultaneously, to three destination nodes D1, D2 & D3.

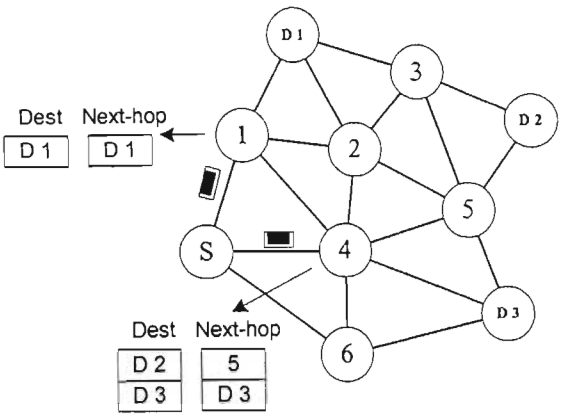


Figure 63(b): Showing nodes 1 & 4 acknowledging node S and re-constructing their next-hop lists to forward the data on towards the required destinations they have been assigned.

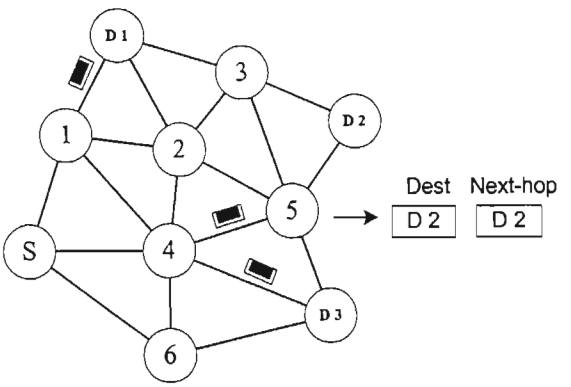


Figure 63(c): Showing nodes D1, D3 & 5 replying with acknowledgement packets, while node 5 re-constructs the next-hop entry it was assigned, from its locally established routing table.

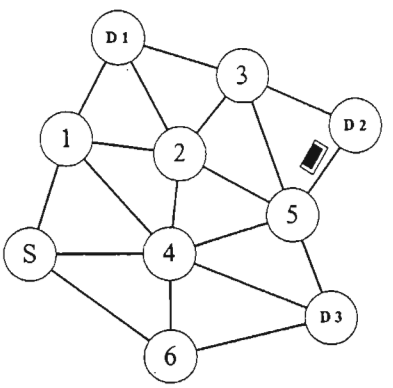


Figure 63(d): Showing node D2 replying with an acknowledgement packet back to node 5[9], thus completing the sequence of events that took place in order for D1, D2 & D3 to receive S's data.

But, as explained in section 2.2.2.1, *table-driven* protocols rely on the use of a periodic routing update mechanism to re-establish next-hop links in the presence of mobility. Since the protocol update mechanism is only triggered every few seconds (or, in some schemes, as soon as a neighbouring link has become broken, or both), there will be a period where the next-hop forwarding scheme will fail. During this period, the DSDV protocol will cache all packets intended for broken next-hop links, until the update mechanism has re-established a valid

[9] Note that if an acknowledgement is not received by an addressed neighboring node, that node will assume that a collision occurred and thus will re-send that packet to the unacknowledged next-hop destination again. However, since this section aims to give only a brief overview of the LAMP scheme, such details will be discussed later in this chapter.

neighbour for a particular packet. However, since it was the intent that time-critical information will be transmitted by each host, it was decided that caching (of a packet) for extended periods of time would be undesirable. In addition, the memory requirements needed to cache multiple packets for each broken next-hop link would be intolerable, due to the limited memory constraints of the handheld device purchased. Hence, instead of caching, a localized location aided flooding scheme was employed to route broken packets immediately, as this ensured acceptable real-time latency bounds. For more information on the drawbacks of caching, please refer to the comments of Figure 91 in Section 5.4.4.

The localized location aided flooding scheme chosen for LAMP is the same algorithm that was adopted by the LOTAR implementation of LAR, which includes *time* as one of its' routing metrics to overcome local maxima (Section 2.2.1.2.2). Note, however, that in [26], LAR was used to find a path more effectively from the source node to the destination node, during LOTAR's route discovery phase, since it provided a better alternative to the *blind flooding* technique employed in AODV. Thus, once a path was found, LOTAR no longer required the services of LAR, since it could now route a packet in a *table-driven* manner. But, since LAR is employed only when LAMP's *table-driven* scheme fails, it is used in LAMP to perform the actual routing of a packet. Nevertheless, the principle is still the same.

The modified LAR algorithm works as follows: Prior to transmitting a packet, a host will append two metrics – the distance it has to the *expected zone* (see section 2.2.1.2.1) and the *time*[10] at which the location co-ordinates were taken for that destination. This information is then broadcast to all neighbouring nodes. If a neighbouring node has not processed this packet before, it checks to see if it contains a better *time* metric for that destination and, if so, it replaces the two metrics with updated values and re-broadcasts the packet; else a host determines if it is deemed closer (or equal) to the *expected zone* than the value found in the received *distance* metric. If this is found to be the case, the packet is re-broadcasted with updated metric values (thus implementing a *greedy* forwarding strategy), or else it is discarded. Hence, this process is an attempt to bring the packet closer to the destination node after each broadcast, as depicted in Figure 64.

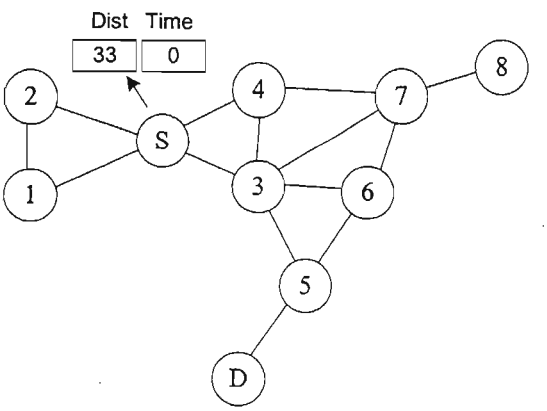


Figure 64(a): Showing the distance and time entries S has for the destination D, in order to demonstrate the flooding capability of LAMP.

[10] Note that *time* is a value that is transmitted during LAMP's periodic update mechanism. But, because each node only transmits its' update information after a certain time period, *time* functions similar to hop count, as illustrated in Figure 64.

Figure 64(a) starts by indicating the topology state of the network. Here node *S* wishes to send data to *D*, but, due to mobility, its' next-hop link to *D* has become broken, preventing it from employing the *table-driven* strategy discussed previously. Thus, instead of caching the packet, *S* reverts to a localized location aided flood to forward data to *D*. *S* begins the flood by appending the *time* and *distance* metrics for *D* to the required data packet. Note that this *distance* is the worst possible line-of-sight GPS reading (in meters) that was obtained from *S* to the *expected zone* of *D* (more about this later), while *time* represents the time that node *D* sent its' location information to *S* (assumed to be zero, for this demonstration). This structure is then broadcast to all adjacent nodes of Figure 64(b), which shows the *time* and *distance* values each neighbouring node contained for node *D*. Applying the modified LAR algorithm to these neighbours, causes nodes 1 and 3 to re-broadcast the packet, since they both contain a better *distance* to D, while node 3 additionally had a better *time*. However, although node 4 contained an equivalent *time* to D, it had a worse *distance*, and thus discarded the packet. Similarly, node 5 in Figure 64(c) re-broadcasts the packet, since all other adjacent nodes had either handled the packet previously or were discarded by the LAR algorithm. Thus, in this scenario, nodes 1, *S*, 3 & 5 were used to forward data on towards *D*. Note that had the *blind-flooding* methodology been used, all nine nodes (i.e. all nodes except *D*) would have re-transmitted the packet, causing bandwidth and power resources to be unnecessarily consumed.

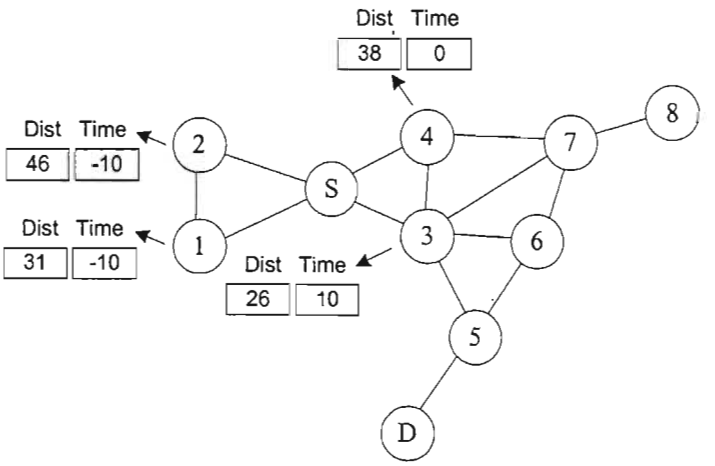


Figure 64(b): Showing the neighbouring *time* and *distance* metrics of S, causing nodes 1 & 3 to re-broadcast the packet.

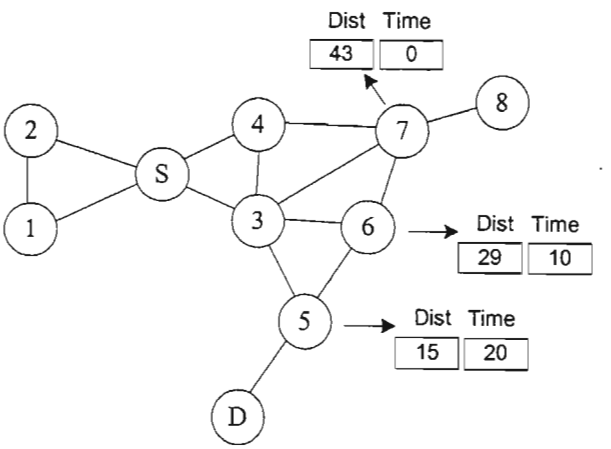


Figure 64(c): Showing the neighbouring *time* and *distance* metrics of node 3, causing node 5 to re-broadcast the packet.

As multiple destinations are required to be processed by LAMP simultaneously, some destinations will contain valid next-hop links, while others will be broken. Hence, some entries in a packet will be handled by the *table-driven* strategy, while others will be handled through the LAR technique. However, this point will be discussed further in Section 5.2.3. Note that since the modified LAR algorithm makes use of extra transmissions (such as the re-transmission of node 1 in Figure 64), it is only initiated whilst a particular route is broken. Once the periodic routing update mechanism has managed to repair the broken route, LAMP reverts back to the more optimal *table-driven* scheme. This way, congestion is kept to a minimum.

5.2.1 The Routing and Packet Tables

Since LAMP consists of two forwarding strategies, its' routing table retains information for both schemes, as illustrated in Table 1, below.

Table 1: Showing the constituent fields of a routing table entry in LAMP.

Unsigned int	Unsigned int	Unsigned char	Boolean	Double	Double	Double[11]
Destination IP	Next-hop ID	Hop count to destination	Advertise this entry	X position of destination[12]	Y position of destination	Time of last update

Where:

Unsigned int	= 4 bytes
Unsigned char	= 1 byte
Boolean	= 1 bit
Double	= 8 bytes

Comparing Table 1 to the table entry of the DSDV protocol, one will notice a similarity, especially in the first four fields. However, one will also notice that the sequence number of the destination is not included in Table 1, but yet LAMP is still termed a *destination-sequenced* forwarding scheme. The reason for this is because the sequence number of the destination is not stored in this table, but rather in another table, termed the packet table.

The packet table is responsible for ensuring that no transmitted data is received by a node more than once, which is necessary when an adaptive flooding algorithm is being utilized (see section 2.3 of [77]). This table structure is given in Table 2, over the page.

[11] Note that the NS simulator makes use of a double (8 bytes) to represent time, while the ARM Linux architecture uses a signed integer (4 bytes), where each bit represents the number of seconds that has elapsed since mid-night 1 January 1970. Hence, the size of this field is architecturally dependent.

[12] The NS simulator makes use of x- and y- positions for nodes, while (in ARM Linux) this information is replaced with latitudinal and longitudinal co-ordinates that are obtained from an externally connected OEM GPS module. In addition, it was found that no further accuracy was gained by using *double's* in Linux and thus the latitudinal and longitudinal co-ordinates were implemented with *floating point* values (4 bytes).

Table 2: Showing the constituent field of a packet table entry in LAMP.

Type:	Unsigned int	Unsigned short	Unsigned int	Double
	Source IP	Destination port ID	Packet sequence number	Time of last reception
Obtained from:	IP header	UDP header	LAMP header	Local host time

Where:

Unsigned int	= 4 bytes
Unsigned short	= 2 bytes
Double	= 8 bytes

Hence, through Table 2, a new packet table entry will be created for every *<source IP><destination port id>* pair, which uniquely identifies a packet through the *<source IP><destination port id><packet seq. no.>* triplet[13]. However, when using NS, this triplet is not required, since NS automatically attaches a global UID to all packets through its *common header* (see section 4.5). But, since this facility is not available in the ARM Linux architecture, a new packet identification scheme was necessary. Because the two main sources of traffic are the routing daemon and the user application, it is expected that the maximum number of entries contained in this table will be 16 (nodes) x 2 (ports) = 32. Hence, through the use of a maximum of only 32 entries, duplication of packets is completely avoided. In addition, note that a *<time of last reception>* field is also included. This field allows one to reset the sequence number of a packet, if a reception was not heard from a host after PKT_EXPIRE_TIME seconds. Thus, if a host becomes disconnected from a network for a period of time (through partitioning), that host will still be able to communicate immediately after re-connection, regardless of its' packet sequence number.

Hence, the combination of the two above tables permits non-duplicate data to be routed. Note that the *<advertise this entry>* field allows one to keep track of what entries are required to be re-transmitted on the next protocol update procedure, since this field indicates that new or updated information has been placed into the routing table, and therefore provides a means for knowing what entries are relevant for neighbouring nodes.

The remaining three fields of LAMP's routing table are used to support the location aided flooding algorithm. Since this implementation makes use of LOTAR's modified LAR scheme, the location information is provided with a *<time of last update>* field, to indicate its age.

Hence, the fields that will appear in a protocol update entry are:

- *<Destination IP>*,
- *<Hop count to destination>*,
- *<X position of destination>*,
- *<Y position of destination>* and
- *<Time of last update>*.

[13] Note that it may make more sense to use a *<source port id>* field, but since forwarding through the *netfilter* architecture (described in section 3.7) causes the UDP source port address to be altered at each forwarding host, it was decided that the destination port address would be more desirable, since it remains unchanged throughout the packets traversal. Similarly, the *<source IP>* is also altered by *netfilter* and thus the *<source IP>* field used here is in fact the original source IP address, which is obtained from the LAMP header (see section 5.2.2 for more information in this regard).

However, please refer to section 5.2.4 for the details pertaining to the periodic mechanism.

5.2.2 The Multicast Next-hop Forwarding Scheme

When an application wishes to transmit data to a set of destinations, it is expected to pass a packet through a UDP socket[14] with a structure that resembles Figure 65.

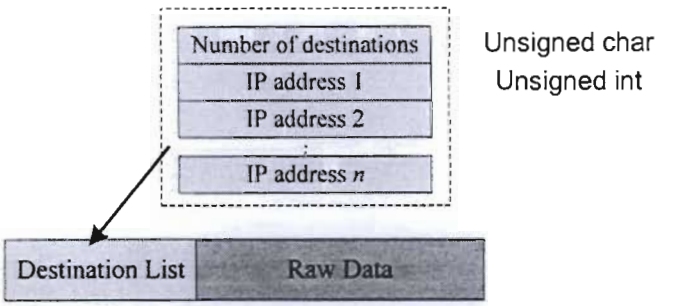


Figure 65: Showing the packet structure given to the protocol stack from the application layer.

This structure contains a list of IP addresses, which is pre-pended to the beginning of the user data. Once this structure arrives at the routing daemon (or agent), it is required to look up the next-hop destination (from its locally established routing table) for each IP address given. Assuming (for now) that a valid next-hop destination was indeed found for each address, the routing daemon appends a second list to the original packet, which contains the next-hop addresses, as shown in Figure 66.

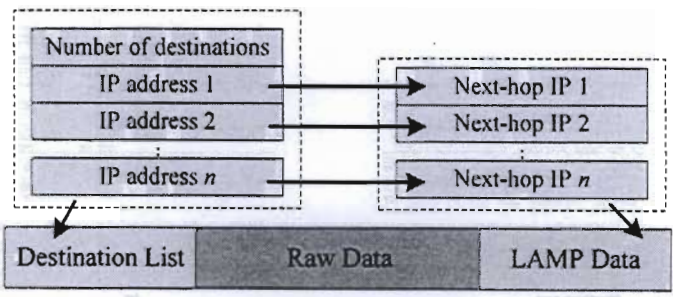


Figure 66: Showing the packet structure developed by the routing daemon.

In addition to the next-hop list, the routing daemon also adds a LAMP header, which contains the fields given in Table 3.

Table 3: Showing the constituent fields of the LAMP header.

Type:	Unsigned int	Unsigned int	Unsigned int	Unsigned int
	Header Marker	Packet sequence no.	Original source IP	Size of raw user data + 8

Where:
Unsigned int = 4 bytes

The *<header marker>* field is used to identify the start of the LAMP header. This is required to ensure that a protocol header is not placed twice on a packet, since packets may arrive (from the *netfilter* architecture) first through hook 4 and then through hook 5 (see Section 3.7). Hence, if

[14] Note that since data is required to be broadcast to all neighboring nodes, the SO_BROADCAST feature of the `setsockopt` function is required to be enabled by the ARM Linux routing daemon, prior to transmission or else the UDP socket will reject the packet.

the fourth-to-last integer value read by a daemon equals the defined header marker, the daemon will realize that the header is indeed in place. But, if the header marker is not read, then the packet was locally generated and thus requires the amendment of its header. However, during the initial testing phase of LAMP on the iPAQ, it was found that callout packets (explained later in this section) would be rejected by the packet table, since these packets would already have been handled by the routing daemon. Hence, the <*header marker*> field was also used to indicate to the daemon that the packet was part of the callout mechanism and thus had to be ignored by the packet table. Therefore, two header markers existed; one that signified a valid LAMP packet and another which identified callout packets.

The next field defines the sequence number of the packet. This field uniquely identifies a locally generated packet and is written by reading the value of the packet counter, which is incremented each time a new protocol header packet is added to a packet.

However, at first glance, the remaining two fields may seem a bit odd, since they contain information that should be accessible from the attached IP and UDP headers. But, since we are dealing with a unique multicast protocol, the forwarding mechanism of IP is unable to use its' locally established IP routing table[15] to determine the next-hop IP address of a packet (as there could be many next-hop addresses required). Hence, to forward a packet and yet avoid the forwarding structures of IP, the routing daemon is required to DROP the original packet (through the *netfilter* architecture) and create a new packet, which it transmits through its own routing socket. However, since sockets automatically set the source and port addresses of a packet, they overwrite the original source information. Hence, to correct this at all destination hosts, the original source IP address is included as part of the LAMP protocol header. Nevertheless, some people will argue that retaining this information is irrelevant, since both the IP and UDP headers are removed from a packet when it arrives at the receiving application. But, since the packet table of LAMP needs the <*original source IP*> field to prevent duplication, it was vital that the original source IP address be maintained throughout the packets traversal.

In addition, note that the full packet size (from the IP header to LAMP header) is required to be a multiple of four bytes (see section 3.5.2), so that the StrongARM CPU can interpret the packet data correctly. Thus, to force this size, a few null characters were inserted, in order to round-up the packet size to the nearest four byte boundary. But, to allow the destination routing daemon to remove these appended null characters before passing the original data to the receiving application, the number of null characters added needed to be stored or re-calculated. Since we wished to locate the <*header marker*> field at a specific location in the final data stream, it was decided that the null characters would be inserted prior to the LAMP header (as shown below), which meant that the LAMP header itself had to be a multiple of four bytes. However, since extra data was being attached to the original user data (and hence altering the UDP <*length*> field), the original size of UDP application data[16] was stored instead of the number of null characters. This way, the number of null characters could be re-calculated, as the routing daemon would be aware of the original data size. It also meant that the routing daemon could be

[15] The local host IP table is established through user-defined subnets, which determine where a packet should be routed next. However, in most cases, IP forwarding is disabled in Linux, since host PCs generally do not handle the routing of data, as these procedures are done through hubs, routers and gateways.

[16] Note that the <*length*> field found in the UDP header includes eight bytes for itself (see 3.5.3).

doubly sure that only the raw user data size was passed to the end receiving application(s). Hence, the final packet structure that was given to the NIC for transmission is depicted below.

Table 4: Showing the packet structure given to the Network Interface Card for transmission.

IP header	UDP header	Destination List	Raw user data	LAMP data	# of null characters	LAMP header

Once Table 4 was given to the NIC, it is transmitted over the air medium using the wireless IEEE 802.11(b) MAC protocol. However, as mentioned previously, this protocol is unable to handle broadcast data reliably, since it is unaware of which destinations the packet is intended for, and hence is able to determine whether all acknowledgement packets were received correctly. Thus, instead of receiving multiple meaningless acknowledgment packets, broadcasting with IEEE 802.11(b) MAC simply ignores the acknowledgment sequence of its protocol engine. Similarly, the reservation (or virtual carrier sensing) mechanism is also ignored, and hence permits *the broadcast storm problem* to be more prominent when broadcasting is employed. Therefore, to reliably broadcast a packet using the IEEE 802.11(b) MAC protocol, the acknowledgement sequence is required to be handled by the routing layer itself. However, to facilitate such a sequence, a callout queue[17] was required, so that the original packet may be re-transmitted using the modified LAR algorithm (for all unacknowledged next-hop hosts). But to achieve such a system, a copy of the packet needs to be placed in the callout queue, prior to the original packet being broadcast.

Once the packet has been copied and transmitted, all neighbouring hosts are expected to search through the given next-hop list to determine whether its own IP address is represented. If, clearly, a host is unable to find its address in the list, that host may simply discard the packet, as no further processing is required[18]. But, if its address was indeed found, that host is required to forward the data structure, handling (through its routing table) only the destination IP addresses it has been assigned. Hence, all other destination and next-hop entries that did not correspond to their IP address will be pruned off, causing the destination and next-hop list to become shorter the as the packet travels towards its' intended recipients. However, prior to forwarding the modified packet, a host is required to send a unicast acknowledgment packet back to the preceding host, with the structure given in Table 5.

Table 5: Showing the constituent fields of an acknowledgement packet in LAMP.

IP header	UDP header	LAMP header

Note that the only data required to acknowledge a packet is the LAMP header, since it uniquely identifies a packet transmission. This, along with the IP and UDP headers, allows the previous-hop to remove a particular host from its callout queue. Once all the acknowledgements have been received for a particular packet, the duplicated packet can be deleted from the callout queue. However, if some of these acknowledgements were not received within the CALLOUT_SCHEDULE time interval, the copied packet will be removed from the callout

[17] A callout queue is a buffer that returns a packet after a certain amount of time, defined in LAMP as the CALLOUT_SCHEDULE time.

[18] The assumption is that there are still no broken next-hop links, as this topic will be dealt with in the following sub-section.

queue and re-transmitted to all host. Since this packet will contain just the un-acknowledged next-hop host addresses, only these will be required to re-handle the packet again. Generally, protocols re-transmit a packet three[19] times, before assuming that the next-hop link is broken. When, this happens the following fields of LAMP's routing table are altered:

<hop count to destination>	= Set to infinity[20].
<next-hop id>	= Set to invalid[21].
<time of last update>	= Increased by one second.

The *<time of last update>* field is incremented by one, to indicate that this update supersedes any previous updates, while the *<hop count to destination><next-hop id>* pair is altered to mark that the next-hop forwarding scheme is no longer valid for this particular destination. Once this occurs, the remaining next-hop entries in the callout queue are sent via the localized location aided flooding scheme.

5.2.3 The Location Aided Forwarding Scheme

As mentioned previously, this scheme is only initiated when the multicast table scheme has failed to find a valid next-hop neighbour for a particular destination in a packet. When this occurs, the following two metrics are used to assist surrounding hosts with forwarding decisions:

- The worst possible distance to the *expected zone* of the intended destination, and
- The time that the corresponding position was last recorded in LAMP's routing table.

However, if a host does not contain a table entry for a destination, then the above metrics are recorded with their absolute[22] worst possible values. These are then inserted in place of the next-hop information, which is broadcast to all neighbouring nodes. Thus, some destination entries will be intended for specific next-hop host, while others will be required to be handled by all. Hence, to accommodate this flexibility, the packet structure (of the LAMP data section) required modification, as depicted in Figure 67.

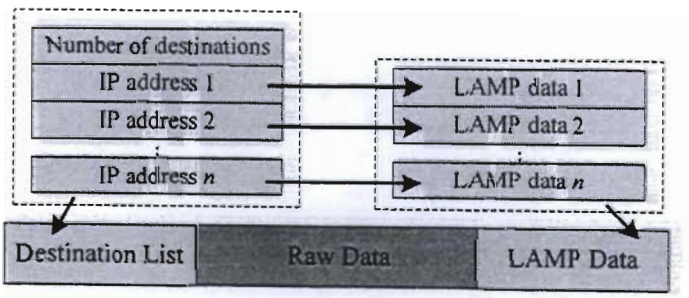


Figure 67: Showing the modified packet structure developed by the routing daemon.

[19] LAMP tries REQUEUE_LIMIT times, before initiating a location aided flood to handle all un-acknowledged next-hop host addresses.

[20] Infinity, by definition, can never be obtained. Thus, it is represented instead by the largest value that is able to be stored by the field in question. As this field is of type *unsigned char*, infinity is symbolized by the value 255.

[21] Since this project only makes use of 16 nodes, an invalid next-hop destination may be any value other than the address of one of the other 16 nodes. However, for convenience, it is represented by the reserved broadcast address, FF:FF:FF:FF.

[22] In other words, the value of the *distance* metric is set to its maximum, while *time* is set to zero. This is done so that neighboring hosts may modify these fields with more accurate information, as the packet progresses towards the intended destination(s).

Where a LAMP data entry consists of one of the following (depending on the scheme employed):

The next-hop forwarding scheme

Unsigned char (1 byte)	Type = 0
Unsigned int (4 bytes)	Next-hop IP address
(4 bytes)	Not used

The location aided forwarding scheme

Unsigned char (1 byte)	Type = 1
Unsigned int[23] (4 bytes)	Time
Float (4 bytes)	Distance

Therefore, when a host receives a packet which contains a *type 1* data entry, it will initiate the modified LAR algorithm (described in section 5.2) to determine whether any benefit will be gained by forwarding this entry. This way, the number of re-broadcasts, and thus the transmission overhead, are both kept to a minimum. Note, however, that since this scheme is a last resort, best effort methodology, acknowledgement packets cannot be used to guarantee delivery.

5.2.4 The Protocol Update Mechanism of LAMP

Like the DSDV algorithm, LAMP makes use of a protocol update mechanism to maintain its routing table. LAMP performs an update every UPDATE_INTERVAL seconds, by counting the number of <*advertise this entry*> fields that are set and placing each one, in turn, into a packet with a structure given in Figure 68.

Unsigned char (1 byte)	Number of table update entries
Unsigned int (4 bytes)	Destination IP address 1
Unsigned int (4 bytes)	Time of update 1
Float (4 bytes)	x co-ordinate 1
Float (4 bytes)	y co-ordinate 1
Unsigned char (1 byte)	Hop count 1
	Destination IP address 2
	Time of update 2
	x co-ordinate 2
	y co-ordinate 2
	Hop count 2
	:
	Destination IP address *n*
	Time of update *n*
	x co-ordinate *n*
	y co-ordinate *n*
	Hop count *n*

Figure 68: Showing the protocol update packet structure of LAMP.

[23] *Time* is represented differently in *NS* and hence changes in the size (and type) of this field need to be made, accordingly.

Once an entry is copied into the packet, the *<advertise this entry>* field is cleared and the final packet structure is broadcast to all neighbouring nodes with the headers indicated in Table 6 below.

Table 6: Showing the structure of an update entry packet in LAMP.

IP header	UDP header	LAMP update data

On reception of Table 6, a neighbouring node examines each entry in turn, to determine whether that entry is better (in time and hop count[24]) than the entry already contained within the routing table. If this is found to be the case, that table entry is replaced and the *<advertise this entry>* field set; else the entry is ignored. Thus, on the next protocol update interval, all altered entries will be re-broadcast to that hosts neighbouring nodes, resulting in a propagation of table update information. Note that a node's local position information is obtained from GPS modules, which are updated according to the NMEA standard. However, since NMEA operates via the RS-232 serial port, it was decided that this process be handled by a separate application program. But, this meant that this program would need to have access to the routing table of the routing daemon. Therefore, to permit such a process, the routing table was placed into a shared memory area, allowing both applications to coherently exploit the same memory addresses (see Section 6.2).

5.3 The Simulation Setup

When dealing with time-critical data, the two most important evaluation metrics are the average delivery ratio and maximum end-to-end latency (explained below). But, like all engineering problems, one generally gains something at the expense of something else. In ad-hoc routing, this "expense" is derived from a complex combination of computational, memory and transmission overhead. Thus, to put the above mentioned metrics into perspective, one needs to include (at least) a transmission overhead parameter, as identified by MANET (Mobile Ad-hoc NETworking) Working Group of the IETF (Internet Engineering Task Force) [1].

Hence, to correctly access the performance of LAMP, the following quantitative metrics were employed:

- The average end-to-end packet delivery ratio, which is defined as the percentage of application data packets that successfully reach their destination,
- The average end-to-end network delay (latency), described as the average amount of time that a packet undergoes to reach its' destination,
- The maximum end-to-end network delay (latency), identified as the greatest amount of time a packet took to reach its destination[25],
- The total number of bytes transmitted, excluding those that constituted user data,

[24] To allow routes with smaller hop-counts to supersede others, all *<hop count>* fields are increased by one before a comparison is made. This way, the shortest known path is always used to an intended host.

[25] Maximum delay is relevant in real-time audio systems because the human ear is able to sense the delay of a packet when end-to-end latencies exceed 250ms. Hence, any packet delayed for periods longer than this will automatically be dropped by the receiving application, causing a reduction in the effective packet delivery ratio of a protocol. Thus, this metric allows one to determine how effective a protocol is in keeping its' maximum latency below 250ms, thereby evaluating that protocols applicability to real-time data systems.

- The total number of bytes transmitted from protocol update and acknowledgement packets, and
- The total number of bytes transmitted from routed packets, excluding those that constituted user data[26].

However, to obtain the above metrics, certain choices needed to be made with regards to the configuration parameters used in the NS simulation environment (see section 4.4.1). A list of the settings adopted is given in Table 7, below.

Table 7: Showing the simulation parameters used to evaluate LAMP's performance.

Simulation Package	NS version 2.1 build 9
Network Area Size	705m x 372m
Duration of Simulation	1000 seconds
User Data Size Per Packet	125 bytes
Traffic Type Employed	CBR
Interface Queue Length	50
Node Transmission Range	250m
Number of Nodes constituting the Ad-hoc Network	16
Underlying MAC protocol	IEEE 802.11(b)

The networking area and node transmission range specifications were selected to provide a reasonable region for nodes to travel, but yet prevented a single node from becoming isolated (although node isolation was catered for in the LAMP scheme through the incorporation the *<time of last reception>* field found in Section 5.2.1). This was done to give each protocol the opportunity of delivering a 100% of its' application data, since this allowed the routing ability of a protocol to be accessed. If this had not been done, a protocol may only show a packet delivery ratio of 70%, when in fact the remaining 30% of the packets were dropped, not due to collisions, contentions or the inability of the protocol to find an alternate route during link breakages, but due to isolated nodes that resulted from host mobility patterns. Thus, to eliminate such scenarios, only mobility patterns with non-isolated nodes were selected for this evaluation. It was found (through trial-and-error) that an 705m x 372m area allowed destinations to be upto four hops away.

Since the PCS required the use of a real-time communication system, data traffic was arranged to emulate an 8 kHz audio channel. In the present application (see Section 6.3), this meant a data size of 125 bytes, which was transmitted every 0.1 seconds, using the CBR traffic generator of NS (see Section 4.4.2). The generator was configured to randomly select the number of source and destination hosts, based on the required packet transmission rate, which was averaged over 30 simulations to negate the outcome of a particular network topology.

In this regard, each node was directed through the influence of the *random waypoint* model, described in Section 4.4.2. But, to ensure that the effects of mobility would be inherent in the results, this model was modified so that the speed of a node could be set between ± 10% of the value allocated to it. Thus, a node passed with a value of 4 meters per second could travel between 3.6 and 4.4 meters per second, and not from 0 to 4 meters per second, as defined previously. In addition, a *pause time* of 1 second was used, since this ensured that a node would not remain stationary for any meaningful amount of time, before selecting a new location point (especially when high mobility rates were being investigated).

[26] Note that the last two parameters are not specified by the IEFT, but were included to determine the contribution that each protocol mechanism incurred on the *total amount of bytes transmitted.*

However, since NS based its' physical interface on AT&T's Lucent Wavelan PCMCIA card and not Cisco's Aironet 350 PCMCIA NIC (as used herein), the following parameters[27] of the physical interface model (known as `Phy/WirelessPhy` in NS) required modification:

```
CPThresh_  = 10.0        ; Capture threshold (dB)
CSThresh_  = 1.1973e-10  ; Carrier sense threshold (W)
RXThresh_  = 1.296e-10   ; Receive power threshold (W)
Rb_        = 11*1e6      ; Bit rate of the interface card (bits/s)
Pt_        = 0.100       ; Transmitted signal power (W)
freq_      = 2.4e+9      ; Frequency of transmission (Hz)
L_         = 1.0         ; System loss factor
```

Once these settings were entered into an OTcl configuration file (see 4.4.1.2), four case studies were performed to investigate the routing performance of LAMP. The first study that was conducted made use of a randomly scattered, stationary set of nodes, which were subjected to a packet transmission rate that varied from 1 packet per second to 50 packets per second[28]. The reason a stationary network was employed was to eliminate the effects of node mobility, so that LAMP's ability to handle congestion could be monitored independently. The protocols compared in this study included LAMP, the modified LAR algorithm, and a multicast version of the DSDV protocol (known throughout the reminder of this dissertation as MDSDV). Note that due to the novel nature of this protocol, previous multicast schemes were unable to be compared[29]. Also note that the modified LAR algorithm is an improved flooding technique, which limits the scope of a flood to a localized area. Therefore, LAR will perform better than the *blind flooding* scheme, as detailed in [25] and illustrated in Figure 64. However, since one wished to determine whether any performance was gained through the use of LAMP, it was imperative that LAMP be subjected to the most stringent of conditions. Hence, the modified LAR algorithm was selected over *blind flooding*.

In the second case study, the effects of node mobility were examined exclusively. This study followed the same procedure as the previous study, but fixed the packet transmission rate to 26.986 packets per second[30], while the node speed was varied from 1 meter per second to 8 meters per second. Note that eight meters per second approaches that of the Olympic world record for the 100m sprint and hence this range covered all possible speeds that *Homo sapiens* could reach (by foot).

However, since the combined effects of mobility and traffic transmission rates were more likely to occur in reality, a third case study was conducted to determine whether any additional effects could be observed. But, since it would be tedious to explore all possible combinations of node speed vs. packet transmission rates, only 5 trial points were selected, as indicated in Table 8, over the page.

---

[27] Note that these settings were obtained from both Cisco's website (http://www.cisco.com/warp/public/cc/pd/witc/ao350ap/prodlit/index.shtml) and NS's independent propagation utilities (found in the `/indep-utils/propagation` directory of the NS source tree).

[28] 50 packets per second were found to be the maximum rate that each handheld device could transmit, once the routing daemon had been implemented on the ARM Linux operating system. Note that 50 packets per second can also translate to five simultaneous 8 kHz audio channels.

[29] Previous schemes required either more than 20 nodes or based themselves on the concept of *group* addresses, both of which were inapplicable to the project at hand.

[30] This rate was selected since it was the turning point at which congestion began to play a role (see Figure 69).

Table 8: Showing selected trial points that were used to evaluate protocols during case study 3.

Trial	1	2	3	4	5
Node Speed (m/s)	1	2	4	6	8
Packet Tx Rate (pkts/s)	0.760	13.948	26.986	40.649	48.440

Note that, in Table 8, the severity of the network is increased, with each successive trial. This way, both the least and most severe network conditions from the previous studies can be examined, respectively.

Finally, the last study investigated LAMP's ability to forward unicast traffic. Here, LAMP was compared to four leading unicast protocols; namely DSDV, AODV, DSR and TORA (all of which were provided as part of the NS distribution). However, since LAMP was not developed for its' unicast ability, only the node mobility effects of this study are presented in this dissertation.

5.4 Results and Discussion

In addition to the simulation parameters adopted in Section 5.3, the following protocol parameters were used during each simulation (unless otherwise stated):

UPDATE_INTERVAL = 15 seconds
CALLOUT_SCHEDULE = 0.025 seconds
REQUEUE_LIMIT = 3

In the original unicast DSDV protocol, an UPDATE_INTERVAL of 15 seconds was chosen, since this value gave a suitable tradeoff between routing overhead and table accuracy. Thus, for comparative purposes, this value was also selected here. But, as will be explained later, LAMP exhibits a better performance, when this value is set to 5 seconds.

However, unlike the UPDATE_INTERVAL, the CALLOUT_SCHEDULE was determined through trial-and-error. It was found (on average) that when this time was reduced below 0.0025 seconds, a node would be required to re-transmit its' data packet more than once, thus incurring an unnecessary transmission overhead. But, since it takes CALLOUT_SCHEDULE * REQUEUE_LIMIT seconds to detect a broken link, the larger CALLOUT_SCHEDULE was made, the longer it takes LAMP to route a packet with a broken next-hop link. Thus, to ensure that the maximum packet delay was kept as small as possible, the CALLOUT_SCHEDULE value was set to 0.025 seconds. Note that the back-off time of the IEEE 802.11(b) MAC protocol may be randomly selected anywhere from 0 to 255 *aSlotTime*'s (provided no carrier signal is detected), where *aSlotTime* is set to 20μs (see 3.5.1.2). Hence, the minimum value that CALLOUT_SCHEDULE could be set to is 0.005 seconds. But, since the original data packet (to the one in the callout queue) will always be transmitted (in addition to other data) during this time interval, it is advised that this time be increased well beyond 0.005 seconds.

As for the REQUEUE_LIMIT, this value was set to 3, as this value was found to be the norm among the routing protocols surveyed.

5.4.1 Case Study One – Traffic Congestion

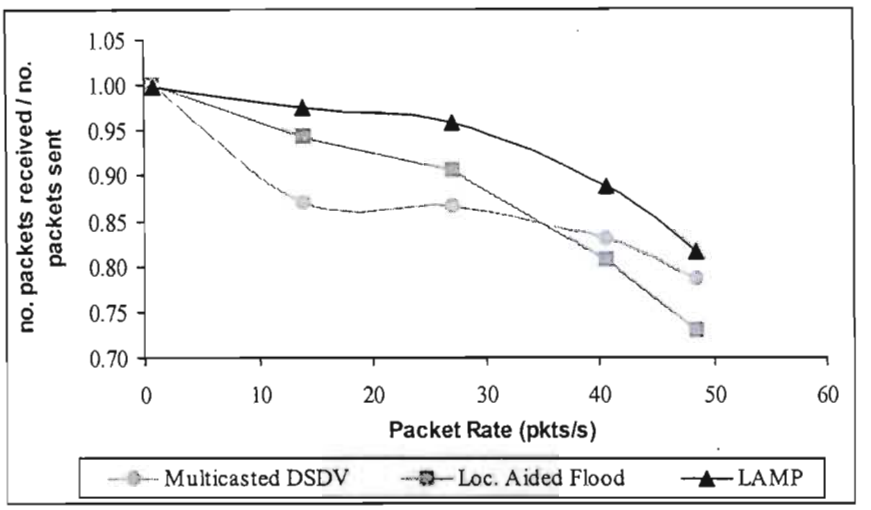


Figure 69: Showing the faction of application data packets that were successfully delivered (packet delivery ratio) as a function of the transmission rate.

The first result, given in Figure 69, shows how the combination of the two protocols can produce another, with a performance better than either alone. Since flooding causes congestion and multiple packet collisions, it performs well, while the packet transmission rate is kept low. But, as this is increased, congestion prevails, preventing flooding from reliably delivering packets. MDSDV, on the other hand, drops all packets intended for a broken next-hop neighbour. Since it takes 15 seconds for a route to become repaired (through the update mechanism), MDSDV may drop approximately *15 x the packet rate* packets, before fixing a link. Hence, some packets are dropped due to out-dated table information, while others from packet collisions. This, thus, causes a congestion cushioning effect to occur, allowing MDSDV to quickly fall to a particular delivery ratio (~ 86%) and then maintain this delivery rate, until it too becomes swamped by packet collisions. Since, LAMP uses MDSDV to reduce the contention that results from multiple forwarding nodes, it is able to route packets with a greater reliability than LAR, before it too is conquered by congestion. Note that LAMP only dips below the 90% delivery ratio at the point where, (theoretically) four simultaneous, audio channels were being transmitted. However, beyond this point, the network becomes too congested to handle the vast volume of data being sent, causing a rapid decline in the delivery ratio of each scheme.

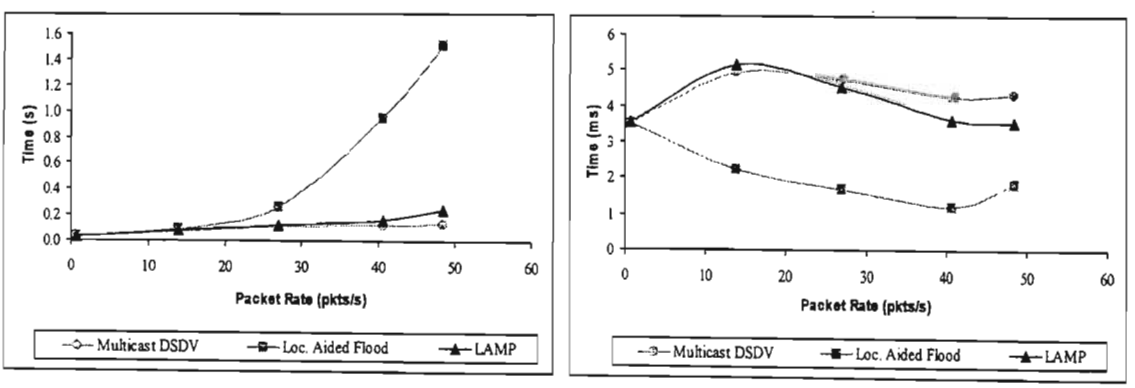


Figure 70: Showing (a) the maximum and (b) average delay that an application data packet experienced, when sent over a stationary set of nodes at varying transmission rates.

Figure 70(a) illustrates one of the downfalls of congestion. Flooding causes the back-off timers of the IEEE 802.11(b) MAC protocol to delay packets for a considerable amount of time, since the back-off algorithm exponentially increases its CW (contention window) in the presence of contention (see 3.5.1.1). However, since MDSDV drops all packets for which the next-hop link is broken, the maximum transmission time of a successful packet is kept low (below 0.14 seconds). In addition, LAMP uses MDSDV to route the majority of its packets, but reverts to a flood for the exceptional cases. Hence, its delay characteristics are very similar to MDSDV, making it suitable to time-critical audio applications[31].

This is also true for the average delay graph, illustrated in Figure 70(b). LAR is able to route packets faster than the MDSDV and LAMP schemes, since it contains a smaller computational complexity than the next-hop forwarding scheme of MDSDV[32]. As LAMP reverts to the LAR scheme only once its next-hop forward scheme fails, it experiences a greater average packet delay than MDSDV. However, as congestion increases, LAMP is forced to revert to LAR more and more (due to next-hop entries being set to 255 in its' routing table), causing the average packet delay of LAMP to dip below that of MDSDV. Note that LAR may seem to initially decrease the delay of an end-to-end packet, as the congestion rate is increased. However, this is not true, since Figure 69 shows that less and less packets are being able to reach their intended destination, causing a decrease in the latency curve, due to the average being calculated over fewer packets. In addition, since the maximum delay curve increases exponentially, more and more packets are being routed with a greater latency than the average, causing this curve to increase when 50 packets per second was being transmitted.

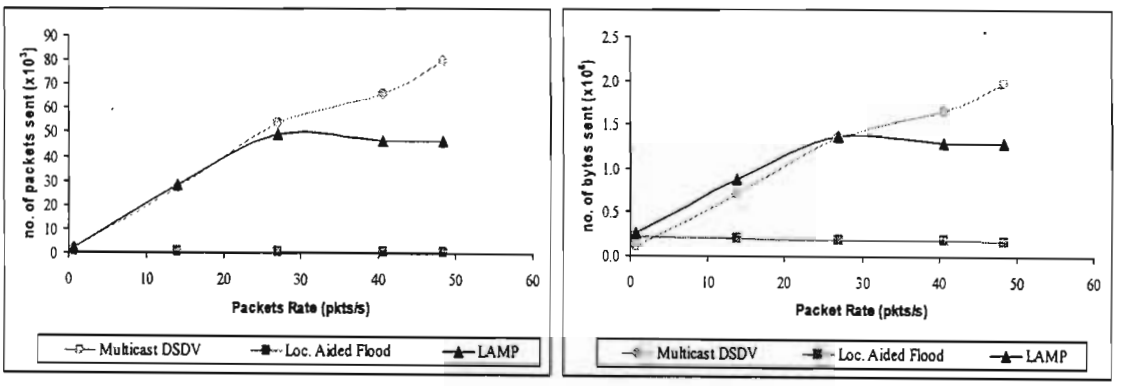


Figure 71: Showing (a) the total number of protocol specific (acknowledgement + routing update) packets sent as a function of the transmission rate and (b) its corresponding size (in bytes).

The number of protocol specific packets that are transmitted as a function of the transmission rate, is shown in Figure 71(a). Figure 71(a) also shows the point where the LAMP protocol deviates from MDSDV. This is because MDSDV and LAMP employ two different schemes to handle broken next-hop links. In LAMP, broken links are handled through the LAR algorithm, while in MDSDV, a broadcast is sent out to all neighbouring nodes informing them of the broken next-hop link. This feature is taken from the original DSDV protocol (known as a *triggered update* [27]) and is used to prevent surrounding nodes from forwarding data to this broken next-hop host. Thus, only when a collision deceives a stationary host into thinking that its' next-hop neighbour has moved out-of-range, do these two schemes begin to differ;

[31] LAMP's maximum end-to-end latency at a rate of 48.440 packets per second was 0.242 seconds.
[32] MDSDV needs to re-build the destination and next-hop lists at each forwarding node.

otherwise they exhibit the same amount of protocol specific overhead, since they use the same callout algorithm. Hence, according to Figure 71(a), this break away point occurs at about 20 packets per second, where LAMP begins to flatten, due to the influence of LAR. Note that LAR's independence to the transmission rate occurs because periodic updates are the only protocol specific data that it incurs, since LAR negates the use of acknowledgements.

However, when the protocol specific data is viewed in bytes (Figure 71(b)), one notices both LAR and LAMP producing more protocol overhead than MDSDV at low transmission rates. This makes sense, since LAMP is required to handle both next-hop and location table data, while MDSDV and LAR are only required to handle each type separately. On the other hand, LAR uses three[33] extra bytes for every packet entry (at every update interval), causing it to exhibit a much larger initial protocol overhead than MDSDV. But, since the number of acknowledgement packets required is directly proportional to the transmission rate, MDSDV's dominance is quickly superseded by LAR.

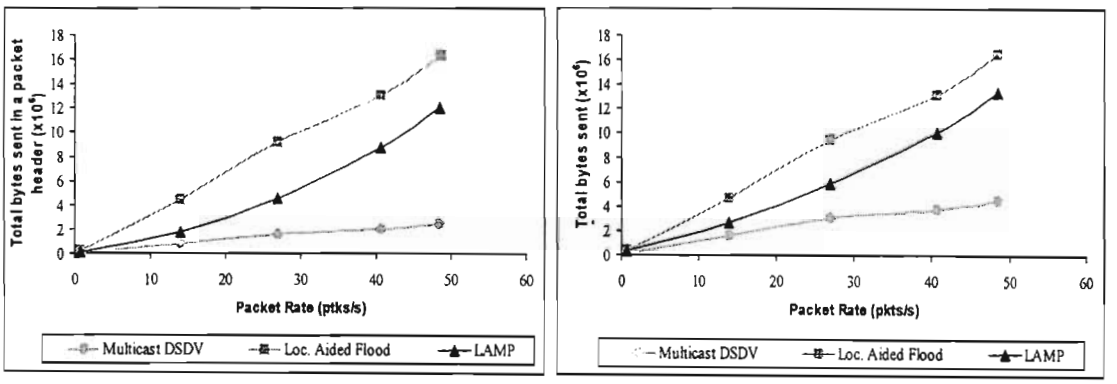


Figure 72: Showing (a) the total amount of bytes sent in a routed packet (excluding that which constituted user data) and (b) the resultant sum of the protocol specific and routing overhead graphs.

Figure 72(a) examines the amount of additional data that was sent (routing overhead), when trying to forward user data to a specific destination. Since flooding makes use of multiple re-transmissions, the same header information is transmitted several times over, causing a large amount of routing overhead to occur. On the other hand, MDSDV forwards data along the shortest path, resulting in fewer re-transmissions and less congestion. As LAMP is affected by both schemes in differing quantities[34], it lies somewhere midway between these two bounds.

Nevertheless, comparing Figure 72(b) with Figure 72(a), one notices that while both MDSDV and LAMP has increased slightly, LAR has remained largely unchanged. This can be understood by reviewing Figure 71(b). Whilst, LAR contributed an additional 200,000 bytes of protocol specific data to the total overhead, both MDSDV and LAMP produced a considerable amount more. Hence, when Figures 71(b) and 72(a) were summed together, the overhead difference between these three protocols becomes less pronounced. However, note how the total overhead graph of Figure 72(b) resembles Figure 72(a). This occurs because the routing overhead size is about 10 times greater than the protocol overhead size. Hence, what Figure 72(b) shows, is that there are three main factors that cause congestion:

[33] LAR uses 28 bytes for every entry in its protocol update packet, while MDSDV uses only 25.
[34] LAMP is influenced initially by MDSDV and then by LAR.

1. The packet transmission rate,
2. The number of nodes that re-forward routed data, and
3. The size of the header that is placed on an application packet.

Thus, if one is able to keep these criteria to a minimum, congestion will also be kept low.

5.4.2 Case Study Two – Mobility Effects

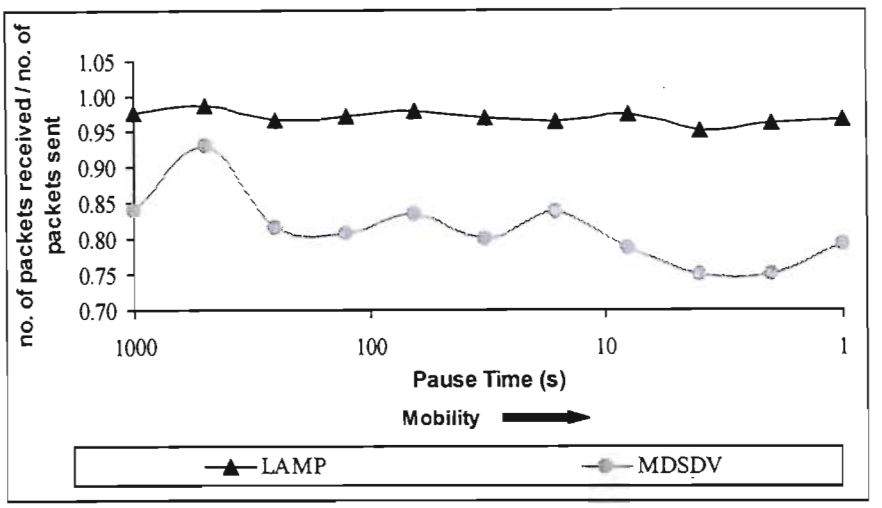


Figure 73: Showing the faction of application data packets successfully delivered as a function of *pause time*, where a *pause time* of zero represents constant mobility.

This second study shows the effect that node mobility has on the performance of a protocol. Figure 73 begins this study by showing how effective the pre-existing *random waypoint* model was in capturing mobility effects. Although, one is able to conclude that LAMP was able to deliver about 10% more of its packets than MDSDV, one is unable to observe the relationship of node mobility on the packet delivery rate. To understand why this is so, one needs to fully comprehend what the *pause time* represents. A *pause time* of 1000 seconds implies a stationary network, while a *pause time* of zero represents a continuously moving network. However, note that this description gives no indication to the speed that a node was moving. Hence, although the *random waypoint* model is able to capture node movement, it is unable to be used to analyze the effects of node speed. Thus, to correct this, the random waypoint model was modified in such a way that a definite speed could be assigned to a mobile node. When this was done, a completely different relationship resulted, as indicated in graphs of Figure 74 below.

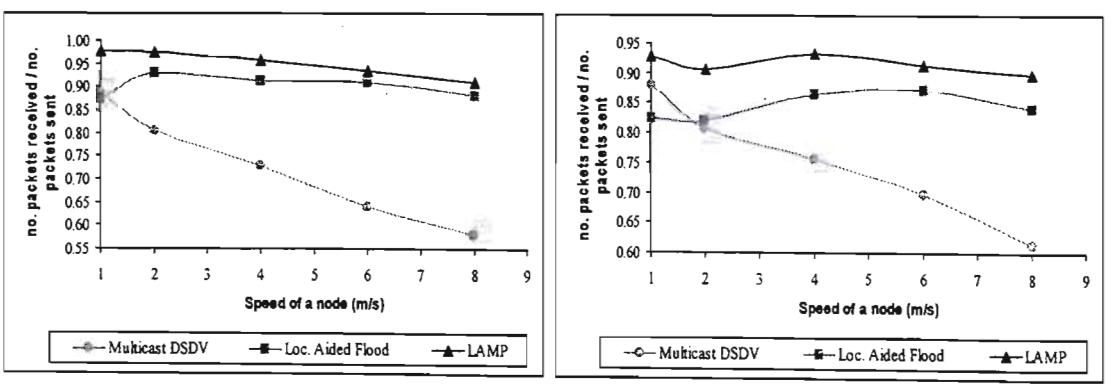


Figure 74: Showing packet delivery ratio vs. node mobility speed, at a transmission rate of (a) 26.986 packets per second and (b) 40.649 packets per second.

As can be seen from Figure 74, both the LAMP and flooding schemes are relatively immune to node mobility, while the next-hop tables of the MDSDV protocol are severely affected. This happens because neighbouring nodes are being altered too quickly for the 15 second periodic update mechanism to keep up. Hence, at high mobility rates, its' next-hop scheme fails altogether. On the other hand, however, flooding allows a packet to be routed with very little state information, since any neighbouring node will do, making it applicable in highly mobile environments [75]. Note that the difference between the two graphs of Figure 74 is the packet transmission rate. As one can see, the packet rate determines by how much LAMP initially deviates from LAR, confirming the results conducted in case study one (Figure 69). But, what is noted is that LAMP's delivery ratio begins to converge towards that of the location aided flood, as MDSDV performance decreases. This makes sense, since the only way LAMP can continue to deliver its packets reliably, is to revert more and more to its' flooding mechanism.

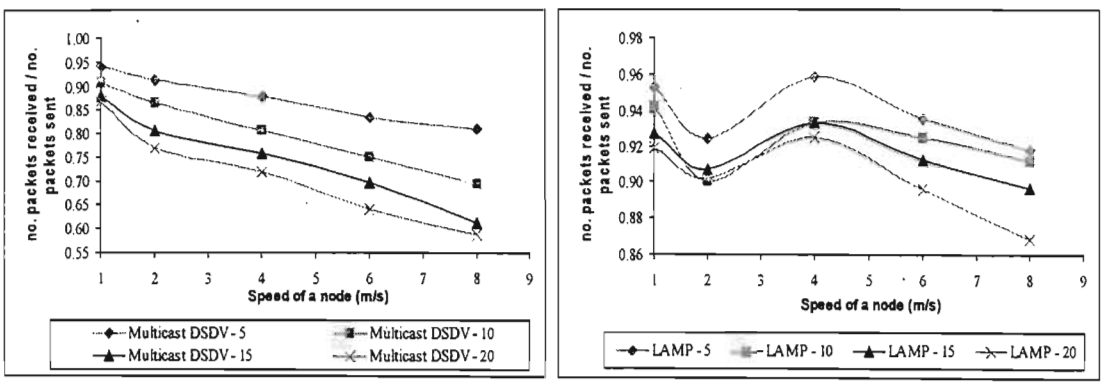


Figure 75: Showing the sensitivity of the delivery ratio to the update interval for (a) MDSDV and (b) LAMP (at a transmission rate of 40.649 packets per second).

Figure 75 illustrates how the protocol update interval can affect the packet delivery ratio of both MDSDV and LAMP. Note the plot in black is the same curve that was used in Figure 74(b) and the numbers indicated after each protocol signifies the number of seconds that was used for the UPDATE_INTERVAL. Figure 75(a) shows that MDSDV's packet delivery performance can be increased by an extra 22%, simply by changing the update interval from 20 seconds to 5. On the other hand, LAMP's performance increased only by an extra 5%, under the same conditions. Hence, although LAMP primarily uses MDSDV as its routing scheme of choice, it is largely unaffected by the value of the update interval. However, having said this, the larger the update interval is made, the more LAMP is forced to mimic LAR. Since it is the combination of both MDSDV and LAR that gives LAMP its superiority, the smaller the update interval is made, the more chance LAMP has to outperform LAR (as will be seen throughout the remaining graphs of this study).

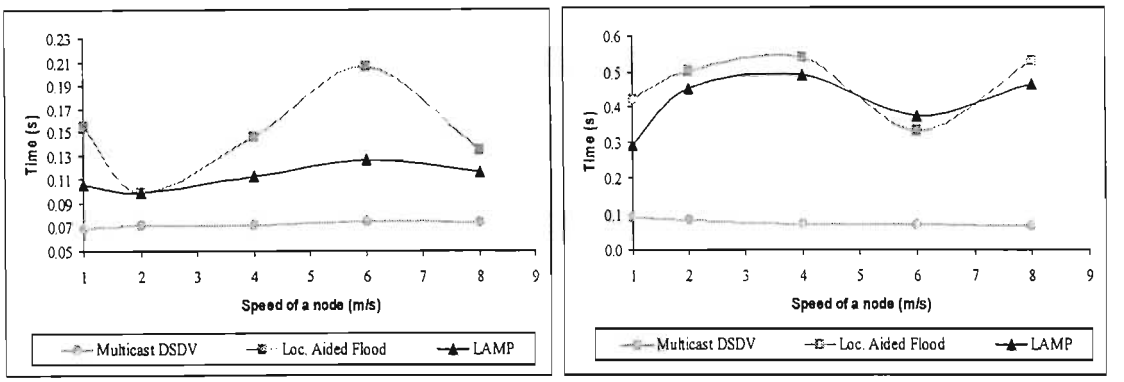


Figure 76: Showing the maximum delay vs. node speed of a packet, when sent at a transmission rate of (a) 26.986 packets per second and (b) 40.649 packets per second.

Similar to the previous study, the maximum delay experienced by a packet (Figure 76), was substantially larger when flooding was used. But, since the same packet rate was employed throughout, these curves vary about a particular time value, which was different from the situation found in Figure 70(a). Note that this variance occurs, due to the random, non-linear nature of the IEEE 802.11(b) MAC back-off algorithm. This point is particularly emphasized when viewing Figure 76(a). Here, contention through multiple re-broadcasts is affecting the back-off mechanism to varying amounts. MDSDV, which uses only a few nodes to forward data to an intended destination, is almost completely flat; LAR, which uses many forwarding nodes, exhibits a large, almost oscillatory action; and LAMP, which operates mid-way between these two bounds, only shows a small deviation.

Similarly, Figure 76(b) exhibits the same back-off trends. But here, LAMP seems to have utilized LAR more often than it did in Figure 76(a), causing it to mimic the same oscillatory characteristic of the LAR scheme. Also note that when the transmission rate was increased to 40.649 packets per second, both LAR and LAMP appear to have non-linearly increased their maximum delay bounds, which is not implied by Figure 70(a). Thus, it is quite feasible to assume that there must be a different overriding mechanism that comes into play, when the effects of transmission rate and mobility are combined. However, how this topic is dealt with in the third case study (Section 5.4.3).

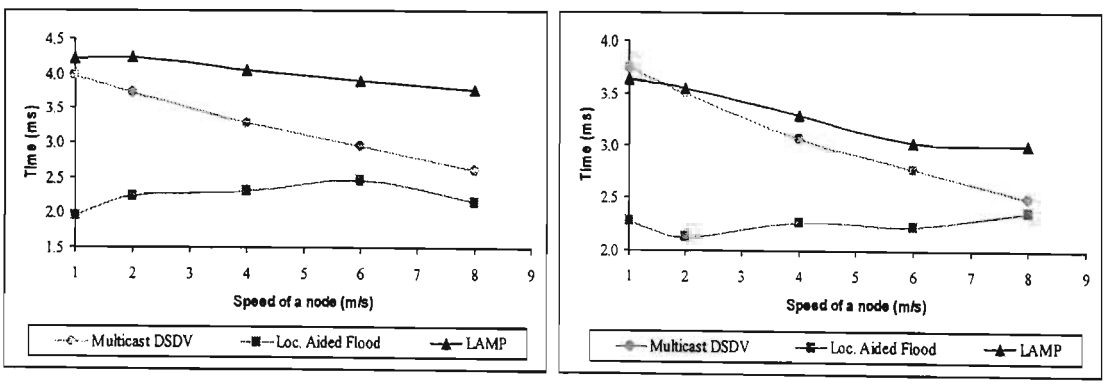


Figure 77: Showing the average delay vs. node speed of a packet, when sent at a transmission rate of (a) 26.986 packets per second and (b) 40.649 packets per second.

The average delay that a packet experienced is given in Figure 77. Looking at Figure 77(a), it seems that both LAR and LAMP's average delays are largely unaffected by node speed; while MDSDV appears to route packets more effectively, as the node speed is increased. But, bear in

mind that MDSDV average is decreasing not because its protocol is becoming more efficient, but rather the contrary. As the speed of a node is increased, MDSDV is routing less and less packets (as indicated in Figure 74), causing a reduction in its' average delay. This fact is verified by Figure 78, which shows the effect that the update interval had on the average packet delay of MDSDV. Since more links can be repaired with faster update intervals, additional packets can be routed (causing MDSDV to dip less), as the mobility speed of a node is increased.

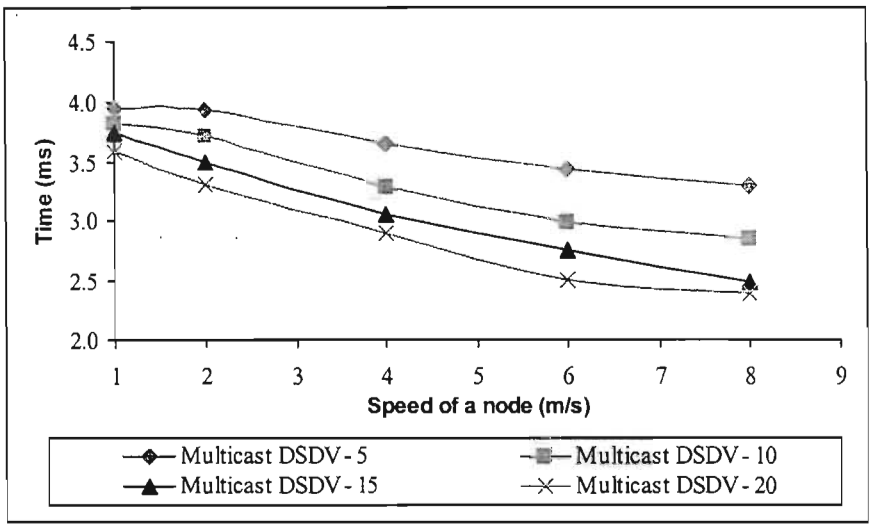


Figure 78: Showing the average packet delay vs. node speed of MDSDV, for various packet update intervals (all taken at a transmission rate of 40.649 packets per second).

Nevertheless, comparing Figure 77(a) and Figure 77(b), one notices that the average delay of LAMP has decreased, as the transmission rate has increased. This occurs because LAMP is reverting to LAR more and more, which is able to route packets faster than MDSDV. This point is further emphasized by Figure 79(b), which illustrates a progressive decrease in the average delay of a packet, as the update interval is increased.

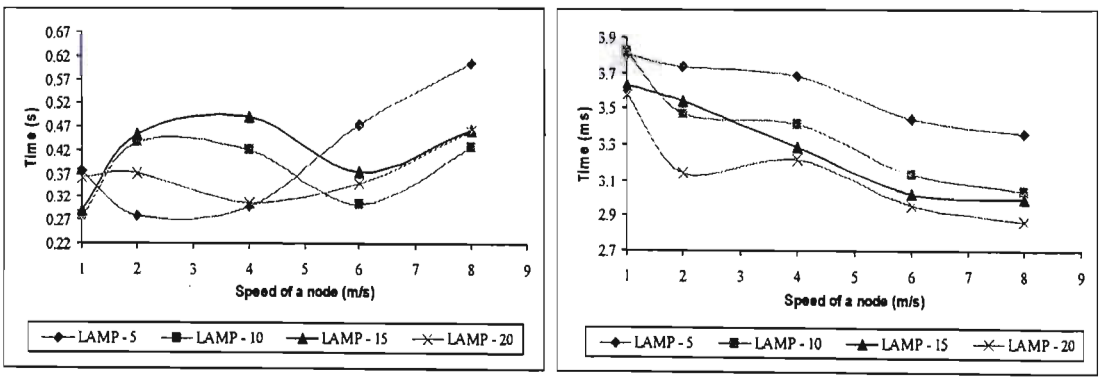


Figure 79: Showing (a) the maximum and (b) average delay that an application data packet experienced, when transmitted at various update intervals (and a transmission rate of 40.649 packets per second).

Figure 79(a) shows the effect that the IEEE 802.11(b) back-off algorithm had on the maximum delay of a packet. As can be seen, there is no noticeable correlation between the maximum delay and the update interval, due to the randomness of this algorithm. But, what can be deduced (from Figure 79(a)) is that the maximum delay of a packet lies between 0.27 and 0.62 seconds, when transmitted at 40.649 packets per second. However, from Figure 76, one concludes that

the upper latency bound (of LAMP) for all end-to-end data is well below 0.25 seconds, when transmitted at 26.986 packets per second. Hence, although LAMP is able to route 90% of its packets to their correct destinations (Figure 74), a few of these will be dropped when four audio channels are being sent simultaneously, due to exceeding the latency 250ms requirement. On the other hand, when only two audio channels are being transmitted, every packet that reaches its' destination will be handled by the receiving application, regardless of the mobility rate of a node.

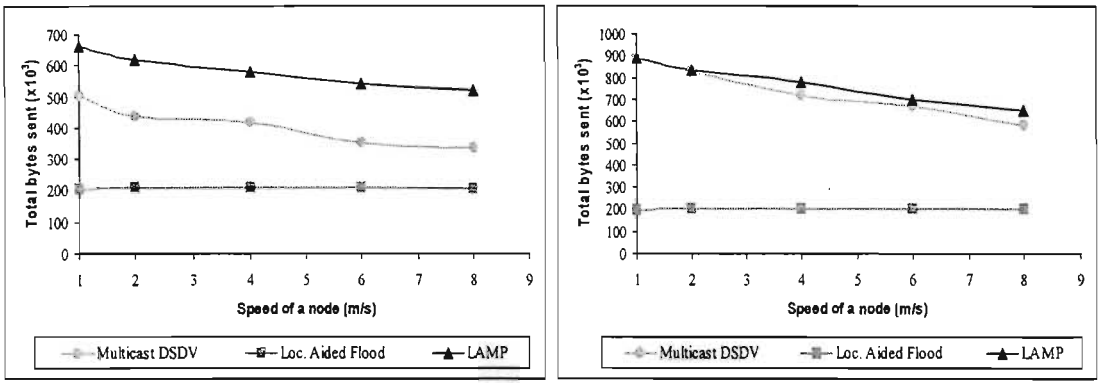


Figure 80: Showing the total number of protocol specific bytes sent as a function node speed, for a transmission rate of (a) 26.986 packets per second and (b) 40.649 packets per second.

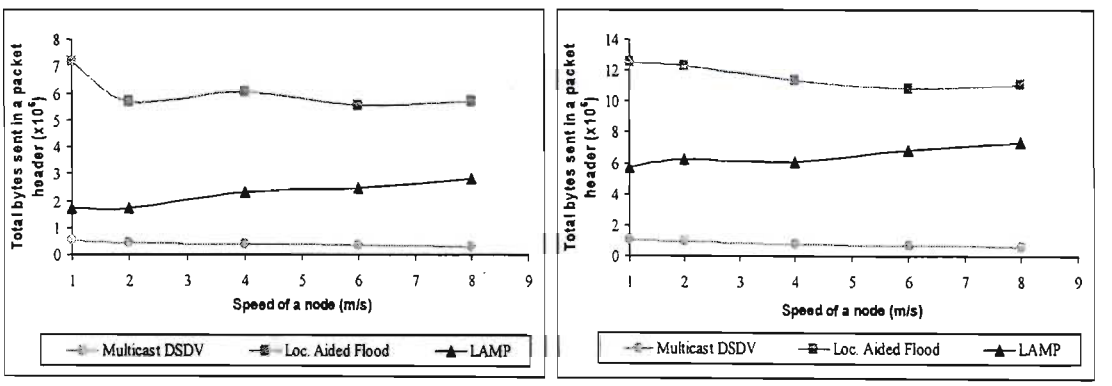


Figure 81: Showing routing overhead vs. node speed at a transmission rate of (a) 26.986 packets per second and (b) 40.649 packets per second.

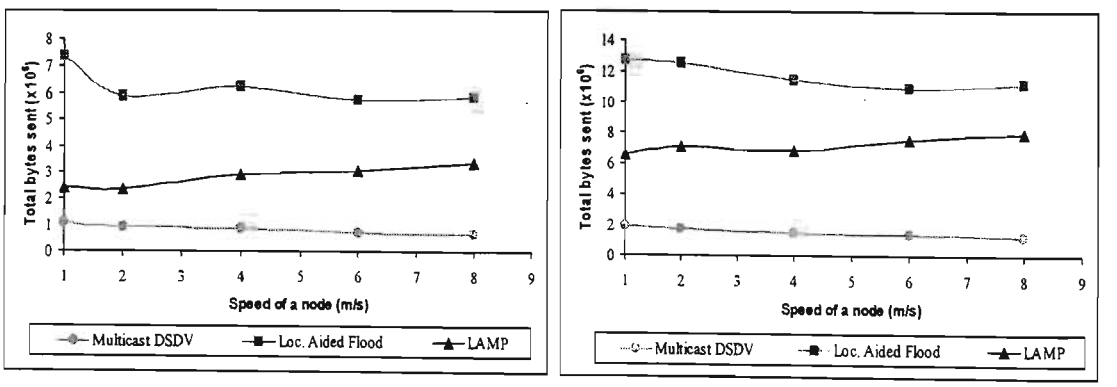


Figure 82: Showing the total induced overhead (in bytes) as a function of node speed, at a transmission rate of (a) 26.986 packets per second and (b) 40.649 packets per second.

The graphs of Figures 80, 81 and 82 reveal similar conclusions to those previously discovered in case study one. In Figure 80, LAMP produces the most amount protocol specific data (as explained in Figure 71(b)), but, comparing Figure 81 and Figure 82 together, this overhead is largely unnoticeable, due to the vast amount of data that is forwarded in Figure 81. However,

what is interesting about these figures is that it clearly shows the extent that LAR has influenced LAMP, especially when part (b) of these figures is compared to part (a). In Figure 80(b), LAMP is able reduce its overhead, to the point that both MDSDV and LAMP are shown to produce equal amounts of protocol specific data; while in Figure 81(b) and Figure 82(b), LAMP is forced into using more and more forwarding nodes (due to the LAR scheme), causing it to produce a larger overhead as both the packet rate and node speeds are increased.

Nevertheless, what is also apparent from Figure 81 and Figure 82, is that both LAR and MDSDV are slowly decreasing their overhead, as the speed of a node is increased. The reason for this is that packets are being dropped (see Figure 74) due to increased congestion, causing less data to be forwarded. Since it is the forwarded data that dominates Figure 82, a gradual decrease in the total overhead graph is noticed.

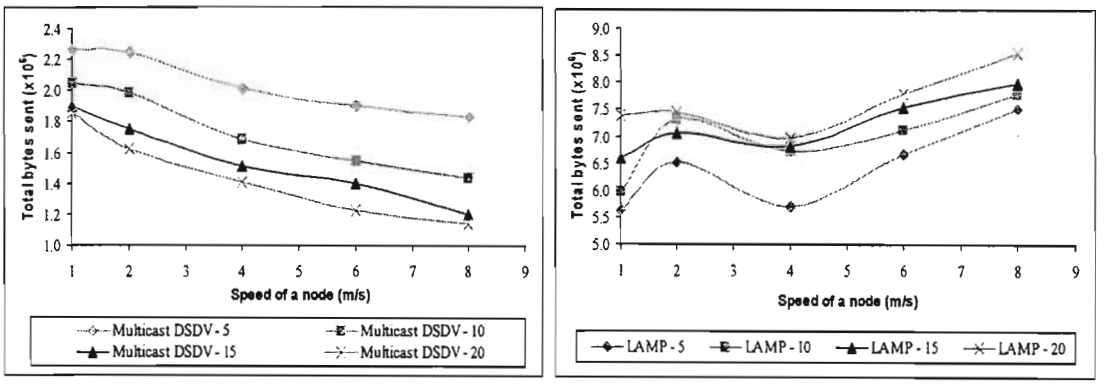


Figure 83: Showing the total overhead of (a) MDSDV and (b) LAMP, at various update intervals (for a transmission rate of 40.649 packets per second).

The last section of this study examines the effect the update interval had on the total overhead produced by MDSDV and LAMP. As can be noted by Figure 83, smaller update intervals cause a larger MDSDV overhead and a smaller LAMP overhead. The reason for this is two fold:

1. Smaller update intervals allow MDSDV to route more packets successfully, and hence incur higher protocol and forwarding overheads (Figure 84).
2. Smaller update intervals allow MDSDV to route more packets successfully, and, thus, LAMP has the opportunity to use MDSDV to route more of its packets. Since it is the forwarding overhead that dominates the total overhead metric, LAMP is able to employ MDSDV to route data over less neighbouring nodes and hence reduce the total overhead produced (Figure 85).

Hence, from the studies conducted herein[35], LAMP was able to route data in the most efficient manner, when the five second update interval was employed.

[35] Note that while routing is performed sporadically, protocol update packets occur periodically. Hence, even during the periods where no data is being sent, update packets will still continue to be broadcast. Thus, to maintain the efficiency of a protocol over an extended period of time, a higher update interval may be found to be favorable.

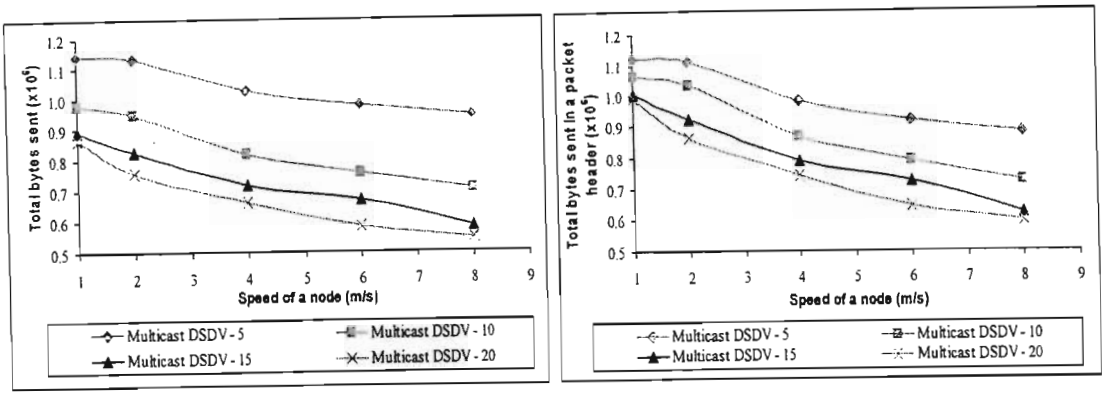


Figure 84: Showing the contribution that (a) the protocol update and (b) forwarding mechanisms (of MDSDV) had on the total induced overhead (for a transmission rate of 40.649 packets per second).

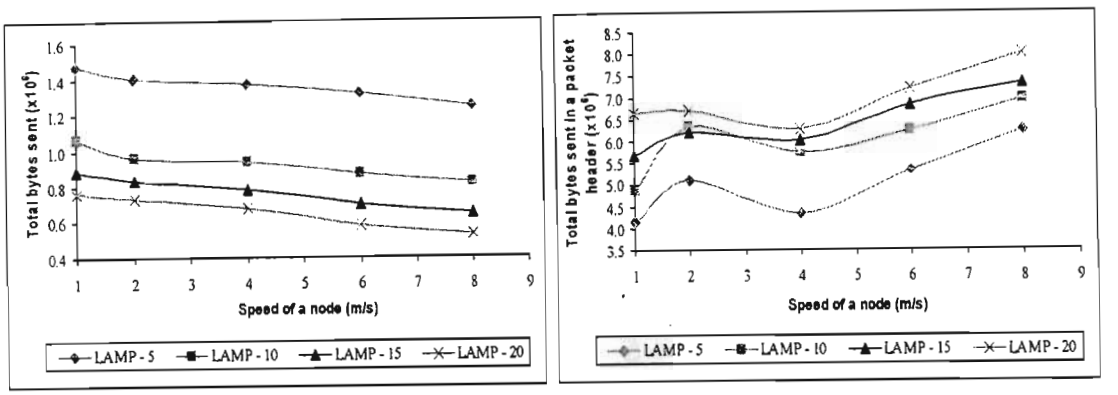


Figure 85: Showing the contribution that (a) the protocol update and (b) forwarding mechanisms (of LAMP) had on the total induced overhead (for a transmission rate of 40.649 packets per second).

5.4.3 Case Study Three – Combined Trials

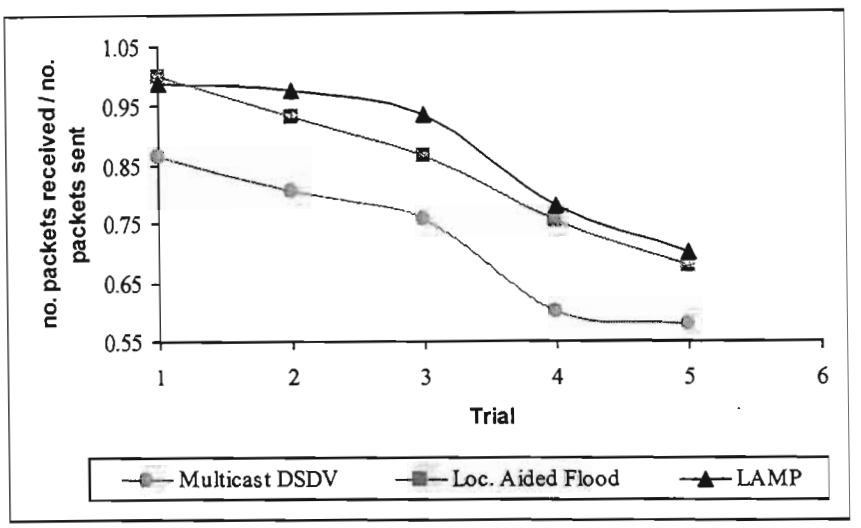


Figure 86: Showing the packet delivery ratio of various multicast algorithms, under varying trials.

Figure 86 starts this case study, by looking at the packet delivery ability of each protocol at the given trial points. Note, however, that an update interval of 15 seconds is still used throughout this case study, so that consistency can be maintained between this study and the previous two. Thus, although Figure 86 shows very little performance gain between LAMP and LAR, bare in mind that this can be altered through the manipulation of the update interval. Nevertheless, what

is apparent from Figure 86 is that LAMP's performance (above LAR) depends on the performance of MDSDV. While MDSDV was able to route 75% of its packets successfully, LAMP was able to out perform LAR by 6.8%. However, as MDSDV dropped off (due to congestion), LAMP could only manage a 2% increase over LAR. Hence, from the analysis of this graph alone, it is obvious that LAMP reverted to LAR more dramatically after trial 3.

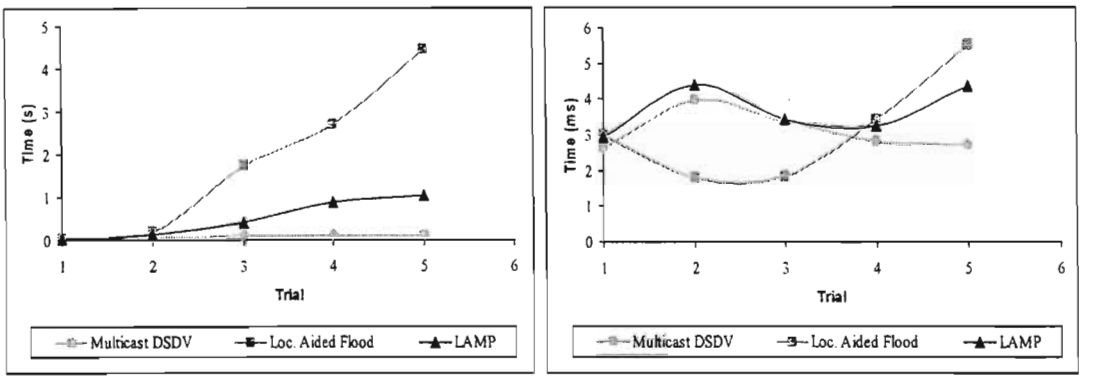


Figure 87: Showing (a) the maximum and (b) average delay that an application data packet experienced, as the severity of the network environment is increased.

The maximum delay taken by all packets during the simulation time is illustrated in Figure 87(a). Comparing this graph to Figure 70(a), one notices similar results. However, here, LAR did not increase its' delay as rapidly as it did in Figure 70(a). This reason for this lies in Figure 76. There one observed an oscillatory motion, which resulted from the randomness of the IEEE 802.11(b) back-off timer. Hence, the absolute maximum latency of LAR will vary, depending on the surrounding congestion level. Since it is expected that the congestion will increase with each trial point, it is logical to conclude that the variance will also increase with each trail point. Therefore, if error bars were placed at each trail point (with increasing magnitude), then the trend of Figure 70(a) would still be obtainable. This point may be verified by looking at how the size of the oscillations grew from Figure 76(a) to Figure 76(b). Nevertheless, what is also apparent from Figure 87(a), is that LAMP"s maximum delay doubled from trial 3 to trial 4, confirming LAR's influence after trial 3.

Figure 87(b) shows the average delay of all packets, for each trial point. Comparing this graph to Figure 70(b), one again notices a resemblance. However, due to the influence of Figure 77, MDSDV begins to slowly improve its delay after trail point 2, since fewer packets are being routed. Other the other hand, both LAMP and LAR begin to increase their average delay just prior to trial point 3. This happens due to the influence that the maximum value has on the average delay. Previously in Figure 70(b), contention was the only factor that affected the delay of a packet. Here, however, the constant change in neighbouring nodes was also contributing to further delay (as noted by the gradual increase in LAR's tread in Figure 77). Hence, instead of LAR remaining below MDSDV throughout simulation period (as observed in Figure 70(b)), LAR is forced to mimic its' maximum delay curve. Since LAMP is relying on LAR to route its packets after trial 3, LAMP begins to inherit the same qualities of LAR, preventing it from dipping below MDSDV (as seen in Figure 70(b)).

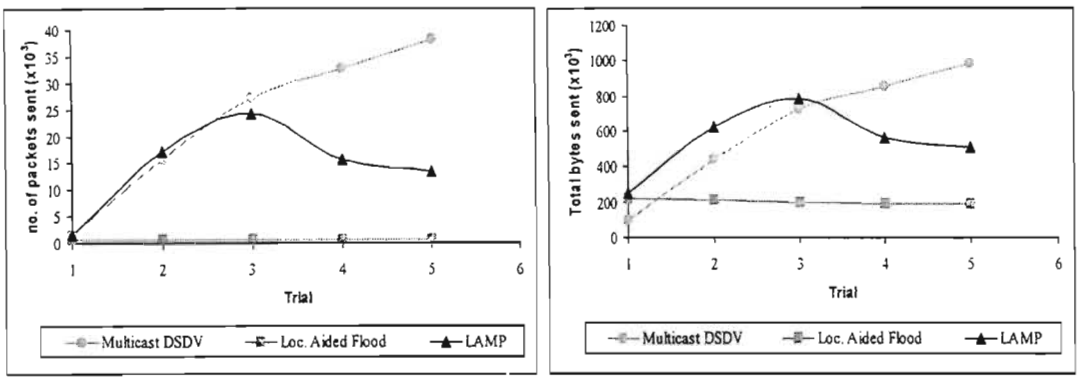


Figure 88: Showing (a) the total number of protocol specific packets sent as a function of various network conditions and (b) its corresponding size (in bytes).

Figure 88(a) shows the number of protocol specific (acknowledgement + routing update) packets that were transmitted during the simulation time period. By comparing Figure 88(a) and Figure 71(a) together, many parallels can observed. In both cases, LAMP begins by tracing MDSDV and then reverts to LAR, causing it to flatten; whilst MDSDV continues to rise indefinitely. However, what is interesting about this graph is its' initial circular shape, which differs from the linear trend found in Figure 71. The reason for this can be explained from Figure 80. There, one observed that the number of protocol packets decreased, with increasing node speed, but increased as the transmission rate was increased. Thus, since one variable increases, while the other decreases, a circular action was noticed.

When Figure 88(a) is viewed in bytes (Figure 88(b)), one observed an increased offset in both LAMP and LAR. As described in Figure 71(b), this comes about due to the differing sizes of the periodic update entries employed by each protocol. LAMP uses 29 bytes for each of its entries, LAR 28 bytes and MDSDV 25. Since an update packet is transmitted every UPDATE_INTERVAL seconds, these differing bytes add up throughout the simulation period, causing each protocol to experience an offset (which is proportional to the original entry size).

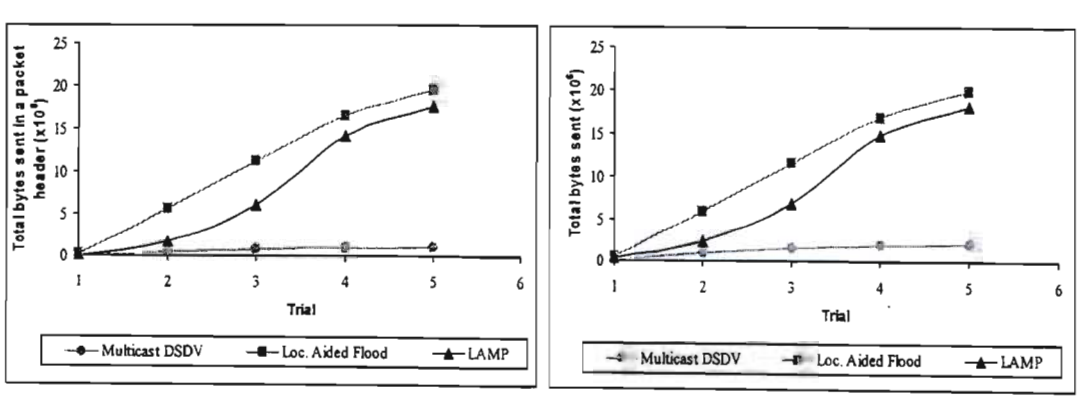


Figure 89: Showing (a) the total routing overhead induced by each scheme and (b) the resultant graph when the protocol specific and routing overhead metrics were summed.

The number of bytes that were forwarded by each protocol is given in Figure 89(a). Again one notices the dominance that LAR had on LAMP just prior to trial point 3, which caused LAMP to "s-bend" towards LAR. However, comparing Figure 72(a) and Figure 82 together, one notices two different trends concerning both MDSDV and LAR. Whilst Figure 72(a) shows these two schemes increase with increasing transmission rate, Figure 82 indicates that they both decrease with increased node speed. Since the increase of the packet rate is greater than the

decrease in the node speed, Figure 89(a) displays a subdued increase with increasing trial points. Nevertheless, what is re-emphasized by Figure 89(a) and (b) is that the forwarding mechanism of each of these protocols contributes the most overhead, as indicated in Figure 89(b).

5.4.4 Case Study Four – LAMP's Unicast Ability

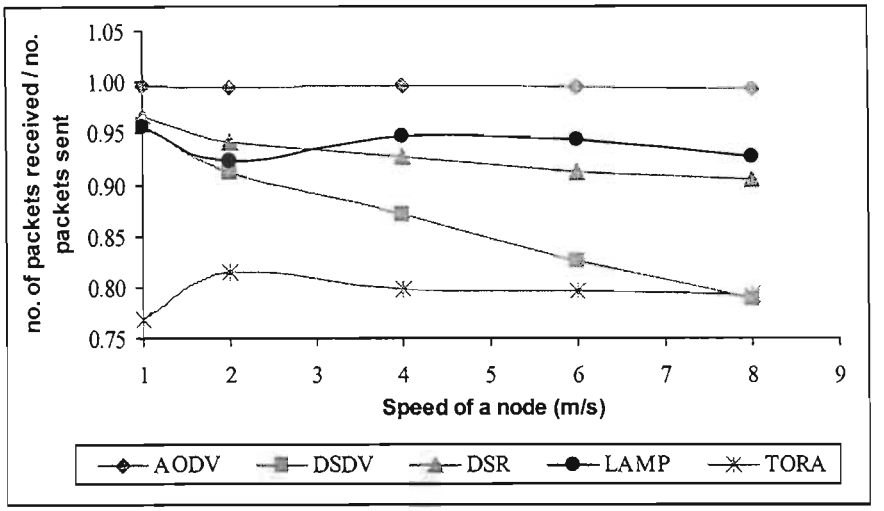


Figure 90: Showing the packet delivery ratio vs. node mobility speed for various unicast algorithms, at a transmission rate of 40.649 packets per second.

The final study takes a look at LAMP[36] as a unicast protocol. Although LAMP was not designed for its' unicast ability, such a study allows one to assess whether a separate scheme is required to send unicast transmissions or whether this same protocol can be used for both multicast and unicast purposes. Figure 90 begins this study by showing a comparison of the packet delivery performance of various leading unicast protocols, at a rate of 40.649 packets per second. As can be seen from this graph, the DSDV algorithm decreases in a similar manner to the MDSDV scheme (indicated in Figure 73), confirming its' correctness. In addition, Figure 90 indicates that LAMP is comparable to DSR (in its' packet delivery ability) and that it was able to increase its' packet delivery performance from that of Figure 74(b) (under the same conditions). The reason for this is two fold; Firstly, a smaller update interval was used, and, secondly, application data is only being sent to a single destination, which meant that fewer forwarding nodes were being employed to route data, resulting in less congestion (and hence fewer collisions).

Note that DSR was unable to route all its packets to the required destination, due to the way it selects a route. DSR chooses a route based on hop-count and not *freshness* (as is done by LAMP, DSDV and AODV), which may cause DSR to select an obsolete route for a packet (from its' route cache) [41]. Thus, DSR's packet delivery performance will deteriorate as the speed of a node is increased. On the other hand, TORA is shown to deliver only 80% of its' packets. This occurs because the *link reversal* scheme of TORA can introduce short-lived routing loops, which, according to the NS implementation of TORA, causes a packet to be automatically dropped [45]. Nevertheless, Figure 90 shows that the unicast protocol with the best performance is AODV, since it was able to deliver 99% of all packets it routed. This happens because AODV selects a route in an on-demand fashion and bases it on *freshness* (destination sequence-numbers). Therefore, AODV is able to out perform all four protocols,

[36] Note that throughout this case study a 5 second update interval was used for LAMP.

since it does not suffer from the same UPDATE_INTERVAL dependence that is inherent in the DSDV protocol.

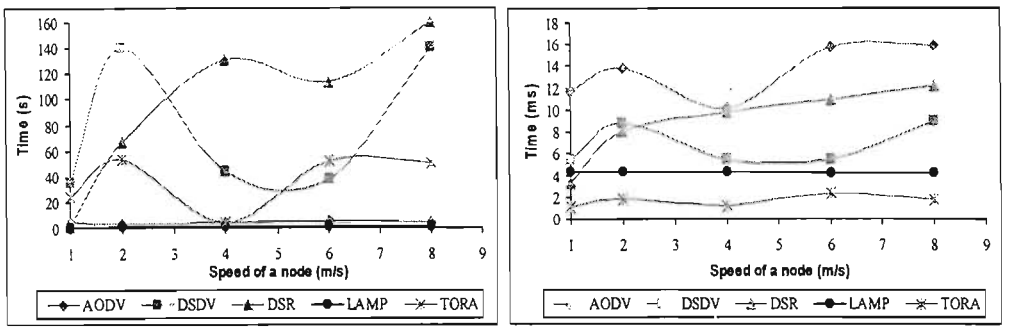


Figure 91: Showing (a) the maximum and (b) average delay of various unicast algorithms, as the speed of node was increased from one to eight meters per second.

Figure 91(a) shows the maximum delay of each protocol. Due to DSR selecting obsolete routes, it exhibits the greatest delay, which rises indefinitely with increased node mobility. However, DSDV showed peculiar results when compared to MDSDV in Figure 76. The reason for this is that DSDV queues all broken next-hop packets, until a valid neighbouring node has been found to forward each stored packet. Since this storage process may happen at each forwarding host, the time taken to route a packet becomes unpredictable, causing it to exhibit a large oscillatory pattern. This drawback is what forced the author of LAMP to implement a LAR flood when the next-hop neighbour becomes invalid. As shown in Figure 91(a), this strategy allowed LAMP to keep its' maximum delay bound down to a minimum, making it suitable for audio communications. In fact, a packet in LAMP experienced anything from 4 to 8 times less delay than its nearest competitor (AODV). This is because AODV is required to initiate a new route request every time a next-hop link becomes invalid. Since this process requires a network wide flood (or blind flood) followed by a unicast reply, LAMP is able to route data faster than AODV as illustrated in Figure 91(b). On the other hand, the *link reversal* technique of TORA allows packets to be routed with half the average delay of LAMP. This is because once the DAG has been established (Section 2.2.2.2.4), multiple unicast paths are known to a destination. Thus, when one path becomes invalid in the presence of mobility, an alternate path is simply used. This is the benefit of using trees. However, when these alternate paths become invalid, a new DAG is required to be re-established for the whole network, causing TORA to produce a higher maximum delay than LAMP.

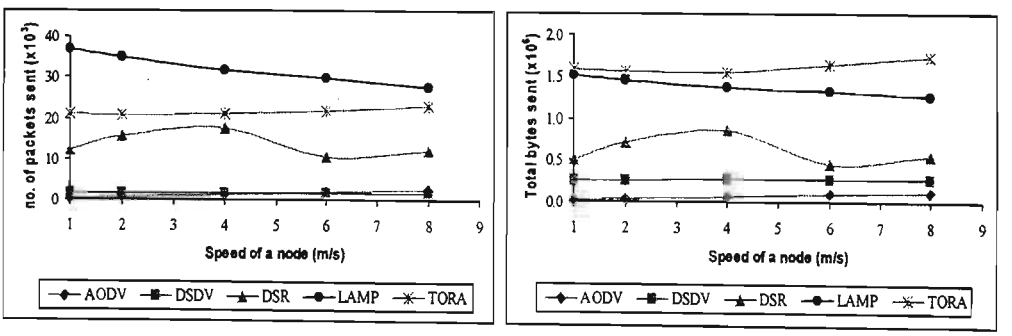


Figure 92: Showing (a) the total number of protocol specific packets transmitted as a function of node mobility and (b) its corresponding size (in bytes), for various unicast protocols.

Figure 92(a) shows the number of protocol specific packets that were transmitted, as a function of node speed. As shown, LAMP transmits the most number of packets. This occurs because

LAMP employs broadcasting to communicate to neighbouring nodes and hence assumes an unreliable next-hop link, due to the nature of the IEEE 802.11(b) MAC architecture. Thus, unlike the other protocols (which send unicast packets), LAMP additionally uses acknowledgement packets to detect broken links. On the other hand, the other schemes use the unicast link-layer breakage detection of the IEEE 802.11(b) MAC to detect these anomalies and thus do not require the use of acknowledgement packets (in the routing layer) [111]. Having said this, however, TORA is shown to deliver more protocol specific data that LAMP (Figure 92(b)). This is due to TORA's DAG establishment, which is used to assign a *height* to each node. Since TORA uses very little header information to forward data (Figure 93(a)), it uses this protocol specific stage to establish each path and, thus, dominates the overhead produced in Figure 92(b). Nevertheless, comparing DSR, AODV and DSDV together, one notices DSR exhibiting the most protocol overhead. This is because DSR transmits the full routing path to each forwarding node, so these nodes may update their local cache memory; whereas both AODV and DSDV forward only the destination address [8]. But, the mechanism used to determine where to forward a packet next is different for DSDV than AODV. DSDV makes use of protocol updates, while AODV uses route requests. Since route requests are only created as needed, AODV is able to produce less protocol specific data than DSDV (Figure 92(b)).

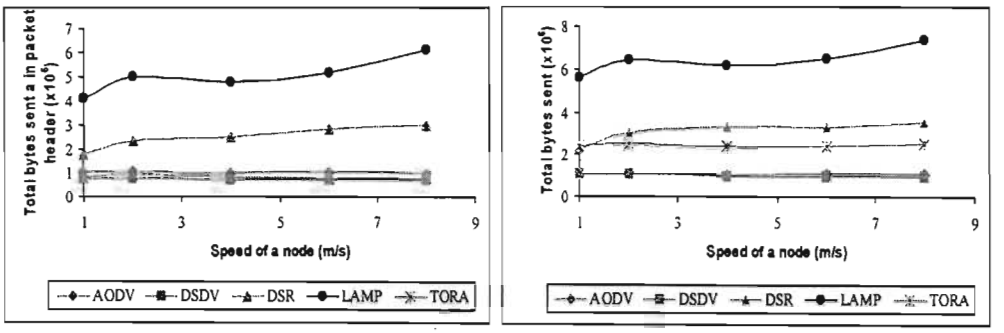


Figure 93: Showing (a) the amount of bytes sent in a routed packet and (b) the total overhead produced by each unicast scheme.

The total header size of all routed packets is illustrated in Figure 93(a). As indicated here, DSDV, AODV and TORA, are able to route data with a ¼ of the overhead of LAMP; while DSR was only capable of achieving a ½. The reason DSR was so expensive, compared to other unicast schemes, is that, like LAMP, it embeds additional information in the header of a routed packet. DSR does this to eliminate the need of a routing table, since each packet contains the full routing path, which is discovered via a locally established cache. But, as shown in Figure 93(b), this methodology causes DSR to incur a large amount of overhead. However, since LAMP additionally floods broken next-hop packets to their required destination, it was able to exceed DSR in terms of overhead. Nevertheless, what Figure 93(a) also shows is that AODV, DSDV and TORA exhibit almost equal amounts of routed overhead. This happens because each of these seems have already established a path for a packet in Figure 92. Thus, very little data is required to perform routing.

However, unlike the multicast case studies, being efficient in Figure 92(a) does not imply a low total overhead, since, in unicasting, the protocol specific graphs are comparable to those found during routing. Hence, to TORA's detriment, both AODV and DSDV become superior to TORA in Figure 93(b). This occurs because of the dominance TORA had in Figure 92(b). But, what is interesting here is that both AODV and DSDV are shown to incur the same amount of total overhead. By comparing Figure 92(b) to Figure 93(a), one notices a reversal in the

supremacy of these two schemes. Thus, when summed together, comparable results were witnessed.

Therefore, although LAMP was able route data with a minimal latency and an acceptable delivery ratio, it incurred a large total overhead (compared to other dedicated unicast schemes) due to the following reasons:

1. Deficiencies in the IEEE 802.11(b) MAC architecture. Since acknowledgement packets are ignored by this protocol during broadcasted traffic, these structures are required to be handled in the routing layer, causing greater latencies in the detection of broken links and the transmission of unnecessary data bits (compare the number of bytes contained in a LAMP ACK (Table 5), with the number of bytes contained in a MAC ACK (see Figure 48)). In addition, current MAC protocols can only handle the transmission of one simultaneous next-hop address. Hence, to accommodate for extra addresses, a list needed to be appended as part of the LAMP header, causing additional overhead.
2. Deficiencies that resulted from IP. IP can only support one destination address. Hence, if more is required, these need to be additionally supplied by the protocol concerned.

Thus, instead of using the same structures to route multicast data, LAMP should simply employ a unicast transmission whenever a single destination is required, since this will negate the need for acknowledgements. In addition, the two lists used to address multiple destinations should be dropped, whenever unicast routing is needed. This way, overhead will be reduced by more than half (compare LAMP to the other unicast schemes of Figure 93(b)), causing LAMP to exhibit improved unicast characteristics.

5.5 Conclusion

LAMP is a novel routing algorithm that combines a *table-driven* unicast protocol with a location aided flood. The *table-driven* scheme is used to identify neighbouring nodes that will aid in the forward progression of a packet to all required hosts, while the location aided flood is employed to find alternate paths to destinations when some peer-to-peer link is no longer valid. Results have shown that such a combination negates the dependency of the update interval on the packet delivering ability of a scheme (especially during high mobility), but does so at an added congestion cost that is well below that of *bind-flooding*. Since Williams et al [77] demonstrated that *blind-flooding* was suitable for sparsely connected networks, LAMP offers an alternate solution that displays better delay, packet delivery and congestion characteristics than any other flood-limiting technique Williams researched.

In addition, LAMP was able to route multicast data directly to all intended recipients and thus negated the need of *group* addresses. Since *groups* require a *spanning tree*, LAMP is not constraint to the same storage requirements needed by these schemes, making it appropriate to high-end embedded handheld applications such as defined by the PCS.

Results have shown LAMP to be comparable to other unicast routing schemes in terms of its latency and packet delivering ability. However, due to the amount of overhead it produced, suggestions have been made to use unicast techniques during the forwarding of single destination data. Hence, if these procedures are adhered to, LAMP will be able to provide both unicast and multicast capabilities to sparsely connected ad-hoc environments, and thus will be deemed suitable for the delivery of delay sensitive data, such as that found in real-time audio communication subsystems.

Implementation Status

6.1 Introduction

This chapter describes the current implementation status of LAMP within the ARM Linux kernel. In this regard, details are given on the task of each developed process, and how these processes interacted to forward (and display) multicast data. Once this has been achieved, a brief overview is given of the elementary tests that were conducted to validate LAMP's multi-hop functionality.

6.2 The Interaction of the Developed Modules

As mentioned in Chapter 5, a *table-driven* forwarding strategy was selected for two reasons:

- The PCS required up-to-date position information. This was done so that the GUI locations (Figure 5) of all hosts contained within a particular cell (i.e. the hosts contained within the dashed circle of Figure 4) could be continuously refresh.
- The location aided flooding algorithm required the use of an up-to-date location *service*. This was done so that the flood could be restricted to a localized area.

Since "up-to-date" information can only be obtained from a periodic update mechanism, the DSDV protocol was chosen over other unicast strategies. However, the bullets above indicate that the same location information was required by two separate processes; one that refreshed GUI information and one that refreshed table information. In addition, since location information was updated by the location *service* (a mechanism that resided within LAMP), a strategy was required to allow the GUI application to gain read-only access to the routing table of LAMP.

One method of achieving this is through the use of a shared memory area, which reserves a portion of memory that starts from a specific address. However, since this memory area will be common among different processes, a technique is needed to ensure that no two processes can gain access to the same memory area simultaneously. This is because variables in this area may be altered by one process, during the read/write cycle of another. Since this can lead to unpredictable process behavior, a key is used to lock the memory area prior to a process gaining access. Once this process has finished reading/writing data to the shared area, it is unlocked, allowing another process to gain access to it. This way, only one process may gain access at a time, ensuring variable consistency among each process. An illustration of this process interaction is given in Figure 94, below.

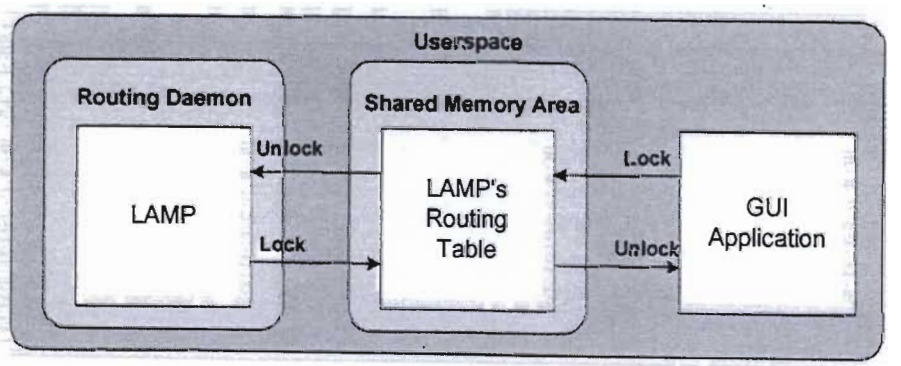


Figure 94: Showing the shared memory interaction of the GUI application and the LAMP routing protocol.

Since the routing table of LAMP could now be accessed from any process, it was decided that an additional module should be developed to decode the required GPS NMEA string, which would update a nodes' local position directly within the routing table (Figure 95). This way, a modular structure was introduced into the project, allowing each process to be coded and debugged independently.

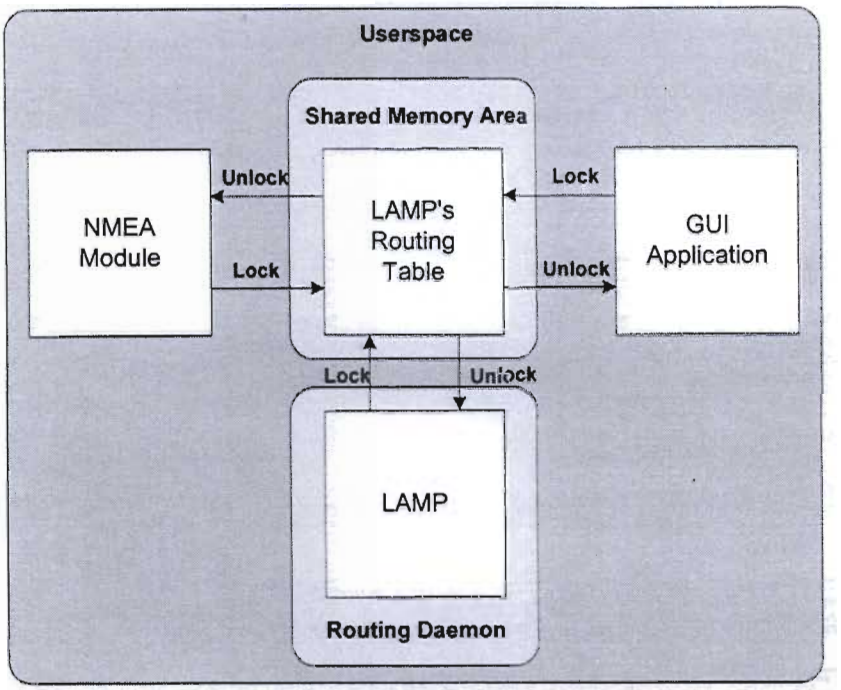


Figure 95: Showing the shared memory interaction of the all three application processes.

The NMEA standard defines both the transmission medium and the format of each GPS string. The output of NMEA is specified to be EIA-422A [122], but for most purposes one can consider this be RS-232 compatible. Hence, the GPS module was simply connected to the on-board serial port of the iPAQ (see Figure 41). Code was then written in the NMEA module to open the serial port driver[37], which allowed the GPS module to stream ASCII strings to the NMEA module.

The NMEA-1083 standard states that each string must begin with a "dollar" ($) and end with "carriage return" (CR) and "linefeed" (LF) characters [122]. In addition, fields within the string are "comma" delimited, followed by an "asterisk" (*) and a 16-bit XOR (exclusive-or) checksum of the full ASCII string. However, because many NMEA strings exist, the first field indicates the type of string being read. For purposes of this dissertation, only the GPGGA string will be discussed. Its' format is indicated in Table 9.

[37] The serial port was configured to operate at 4800 bps, 8 data bits, no parity and one stop bit (known as NMEA-1083).

Table 9: Showing the format of the GPGGA NMEA string [122].

Field #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
String	$GPGGA,hhmmss.ss,llll.ll,a,yyyyy.yy,a,x,xx,x.x,x.x,M,x.x,M,x.x,xxxx*hh															

Where:

1 = Global Positioning System (Fix Data)
2 = UTC of Position
3 = Latitude
4 = N or S
5 = Longitude
6 = E or W
7 = GPS quality indicator (0=invalid; 1=GPS fix; 2=Differential GPS fix)
8 = Number of satellites in use [not those in view]
9 = Horizontal dilution of position
10 = Antenna altitude above/below mean sea level
11 = Meters (Antenna height unit)
12 = Geoidal separation (Difference between WGS-84 earth ellipsoid and mean sea level)
13 = Meters (Units of geoidal separation)
14 = Age in seconds since last update from differential reference station
15 = Differential reference station ID number
16 = Checksum

Hence, the task of the NMEA module was to ensure that the seventh field value of the serial stream was non-zero and that the <checksum> was valid, before passing fields 3, 4, 5 & 6 to the shared memory table. With this in mind, Figure 95 can be extended to reflect both the *netfilter* framework (described in Section 3.7) and the serial modules required to obtain the GPGGA NMEA string, as given in Figure 96.

In addition, the following should be noted with reference to Figure 96:

- The "kernel packet filter" represents the module that was used to apply the *netfilter* rules to each packet, causing a response of either NF_DROP, NF_ACCEPT, NF_STOLEN or NF_QUEUE to be returned back to the *netfiler* framework (see Section 3.7).
- The "IP packet queue" was used to send NF_QUEUE'ed packets to the background routing daemon for modification, after which the original packet was either accepted, dropped or replaced (Section 3.7).
- The routing daemon required the use of a datagram socket to transmit data via the NIC, so that data, not intended for this local host, may be re-forwarded (Section 5.2.2).
- A "multiplexer module" was required to connect four serial streams (or pipes, as they are known in Linux) to one physical port, as illustrated in Figure 41.
- A "PSCP (Positional Communication System Protocol) module" was inserted to convert GUI interactions to peripheral interactions.
- The "TRDCS (Tactical Radio Data Communications Standard) module" was used to interact with the protocol stack of the tactical radio network. However, note that only text messages are currently able to be transmitted over the tactical radio network, through the use of two iPAQ's.
- The "image module" conveyed CMOS images to the GUI.
- A "voice application" was developed to record an 8 kHz real-time audio channel received from the headset (Figure 41), which was transmitted and played-back over the established ad-hoc network (more about this in Section 6.3).

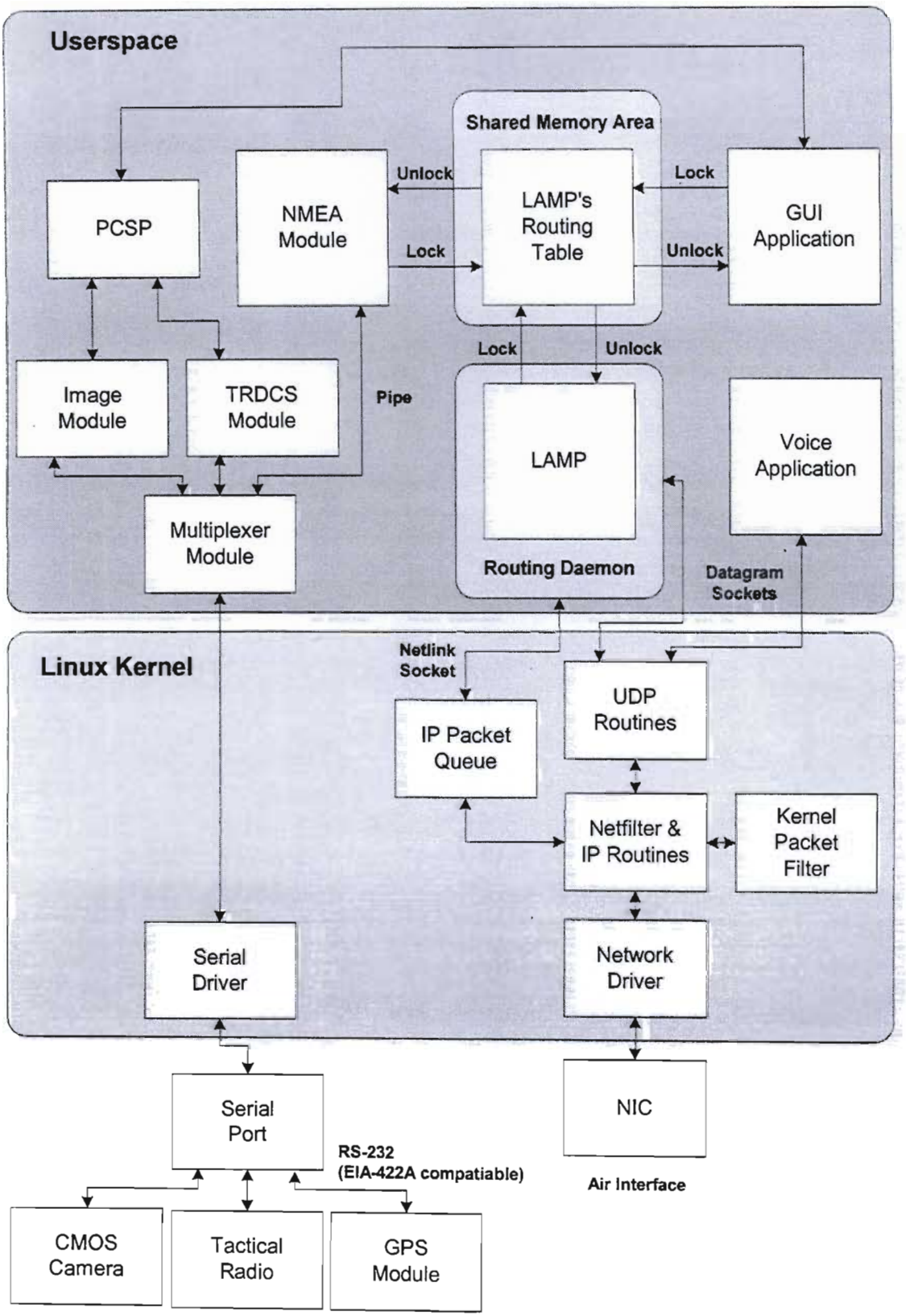


Figure 96: Showing the interaction of all participating modules.

6.3 The Elementary Routing Tests

In order to test the multi-hop functionality of LAMP, a few elementary tests were necessary to ensure that LAMP conformed to the requirements of the PCS. However, to conduct these tests, two applications were needed.

The first application that was developed (APP1) simply pre-pended a hard-coded destination list to a 125 byte data packet, which was transmitted at a rate specified by the user through the use of the Linux prompt. In addition, a signal handler was constructed to print the number of packets that was transmitted, once the user terminated the application. This way, an end-to-end delivery ratio could be calculated at each source/destination pair.

The second application (APP2) performed the sample and play-back of the real-time 8 kHz voice stream. Here a sampling thread[38] was used to quantize a voice signal (obtained from the microphone of the headset) into frames, which was then transmitted over the NIC through the use of the *netfilter* architecture. On the receiving end, another thread was used to queue each received packet, which was buffered for playback (on the headset). In each case, the encoding and decoding phase of the raw audio data was accomplished through the use of an on-board DSP, which could be configured to handle different sample rates. The sample rate opted for in this project was that of an 8 kHz mono (single channel) data stream, which (when coded by the Pulse Code Modulation (PCM) scheme) translated to packet size of 125 bytes, transmitted at a rate of 10 packets per second. An illustration of the voice application (with its' two threads) is given in Figure 97, below.

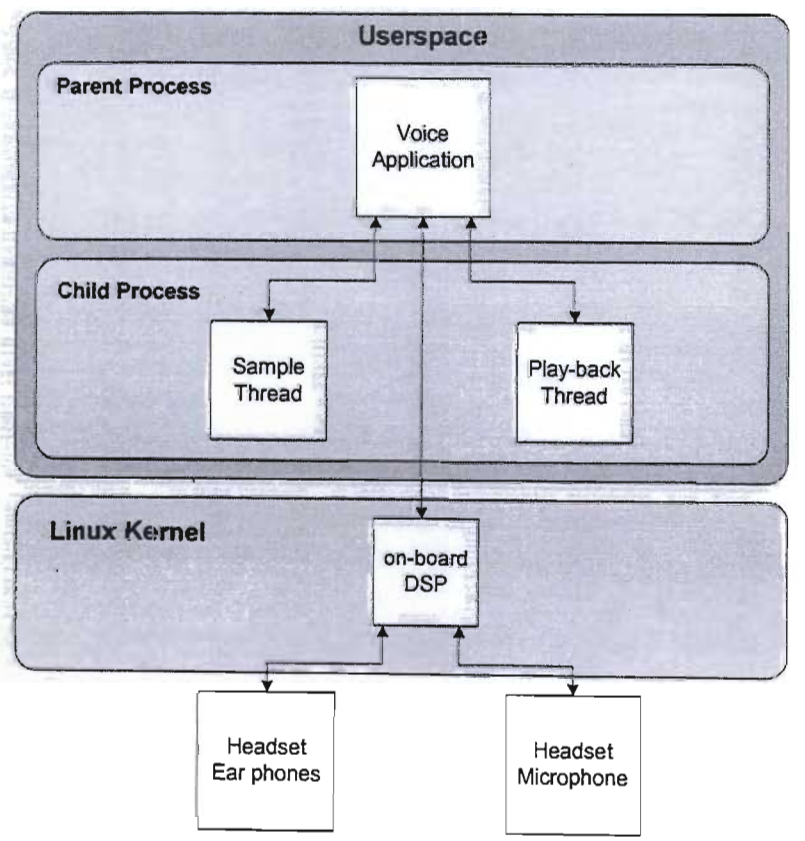


Figure 97: Showing the components on the voice application.

[38] A thread is a special type of process in Linux.

6.3.1 The Outdoor Tests

The first test that was performed examined the transmission range of the NIC. Here, two iPAQ's were loaded with APP1 and were taken out into the University of Natal (Durban) campus to establish the line-of-sight distance that could be achieved between these two devices. It was found that when a distance of ~200m was established, very few packets (below 5%) were received correctly, with occasional "dead spots" (deriving from noise and the multi-path interference of buildings, cars, trees and people) occurring at distances of 120m and more.

This test was then modified to include a third iPAQ, which was employed as a router to forward data between the two previous devices, since such a strategy would allow one to assess LAMP's routing ability. However, in order to ensure that data was in fact being forwarded, APP1's destination list was adjusted to include the router. This way, end-to-end delivery ratios could be calculated at both the router and the intended destination node, allowing the multicast ability of LAMP to be assessed. It was found that when both the router and destination nodes were placed near each other (but 50m away the source) a delivery ratio of 95% (and above) was received by each, even when a transmission rate of 50 packets per second was being sent. However, when the destination node was moved to distance of 150m, the router's delivery ratio dropped to 80% and the destination node received only 75% of the packets that were received from the router (i.e. an end-to-end delivery ratio of 60%).

One explanation for this has been the following: While the two destination nodes were together, any packet that was not acknowledged was re-broadcast via the LAR algorithm, causing both destination nodes to forward the data to one another. Since this increased the likelihood of each node receiving the packet, a ratio of 95% was achieved. However, when the two destination nodes were separated, very few packets were able to reach the distant node directly (from test 1), causing that node to be almost completely reliant on the router. Hence, when a packet was not acknowledged and the source reverted to a LAR flood, a re-broadcast was only sent through the router, causing a reduction in both ratios.

Nevertheless, the following can be noted from these two tests:

- Nodes 250m (or less) away did not always receive a transmission due to interference and noise, which was not evident by the NS distribution. This is because NS assumes that each transmission is perfectly circular (see Section 4.5). Hence, other than a collision, a node in NS will always receive a packet if it is within transmission range of sending node. However, in reality, this was not found to be the case. In fact, the probability of a packet being received correctly was dependent on distance, with occasional "dead spots" occurring. Thus, if there are three nodes, where one is in the middle of the other two and each node is within 200m of one another, then the nodes on either end may experience an oscillatory hop count to each other that varies from 1 to 2. This is because the probability of the routing update packet being received correctly by these two nodes is less than 5% (from the first test). Hence, it is more likely that the routing table will show these nodes to be two hops away, when if fact they can be reached in one. This was confirmed by an application (APP3), which simply viewed the routing table of LAMP from the shared memory area, described previously.
- LAMP was able to forward multicast traffic, but the results obtained were poorer than those observed during simulation. In Chapter 5, LAMP was shown to deliver 83% of its packets to all intended destinations, when transmission rate of 50 packets per second was used (Figure

69). However, in reality, this value fell to 60%. In addition, when LAMP was replaced with a *blind-flooding* scheme, the end-to-end delivery ratio fell to 50% (under the same conditions), confirming LAMP's superiority over such schemes. Hence, although simulation predicted that four audio transmissions could be routed simultaneously, it is suggested that this value should be reduced to 2 or 3 transmission instead. However, as will be explained in the next section, work is still needed to confirm this.

6.3.2 The Indoor Test

To prevent the hop count from oscillating, an indoor test was conducted. This is because it was found that when a node was moved behind the lift of the School of Electrical, Electronic and Computer Engineering building that node became completely isolated from other nodes within the postgraduate lab, as shown in Figure 98.

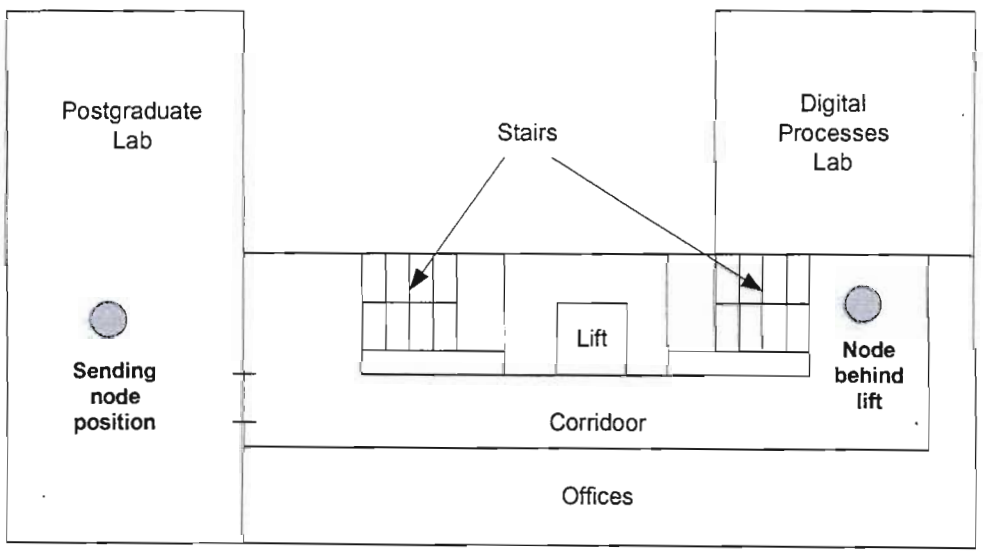


Figure 98: Showing the position of the isolated node, within the School of Electrical, Electronic and Computer Engineering building.

However, being indoors meant that GPS information would no longer be available to each node, since the GPS antenna was unable to "view" any satellites. Hence, LAMP was modified to perform *blind-flooding* under these conditions, allowing the more optimal LAR algorithm to be initiated when location data was available. This way, the PCS will still be able to function when GPS information was unattainable.

Nevertheless, to ensure that the hop count within the routing table did not oscillate, APP3 was loaded onto three nodes; with one placed in the postgraduate lab (the sending node), one in the corridor, and another behind the lift (see Figure 98). Hence, in this configuration, the nodes on either end were unable to communicate, without the intervention of the node in the corridor. When this was done, it was found that the hop count was indeed stable, and that each end contained one hop to the router and two hops to each other. Also, when the node in the corridor was moved closer to either end, the other became isolated, as expected. With this new location investigated, the audio application (APP2) was loaded onto the iPAQ's for testing.

The audio that was received (from the end-to-end nodes) was clear and when the router moved closer to either end, the audio signal terminated (at the point where the other node became isolated), but was re-established when moved back towards the centralized position. However,

note that this scenario can only validate the reception of one audio channel. This is because the reception from multiple voice sources requires mixing.

Mixing is the process of adding multiple audio streams together, in order to produce a single voice stream that can be played-back on the headset. Work was conducted to place each received stream into a separate buffer, which was then de-queued and digitally added for output on the headset. However, when this was done, it was found that the resulting stream was verbally indistinguishable. Suggestions (to solve this dilemma) have been to use a 2's complement adding scheme (instead of the ordinary arithmetic approach), but this still needs to be confirmed.

Hence, although the above test was able to prove that a single voice channel was able to be routed over the ad-hoc network, work is still required to determine the number of audio channels that can be handled by each iPAQ simultaneously.

6.4 Conclusion

Although Chapter 5 was able to demonstrate LAMP's supremacy over other flooding schemes, this chapter has motivated the need for protocol verification through a physical implementation. This disagreement arises from assumptions that were made during simulation, such as the transmission of perfectly circular emissions. Since assumptions such as this negates the likelihood of noise and multi-path interference, discrepancies occur in the theoretical performance of a protocol.

Nevertheless, this chapter was able to illustrate the interaction (and modularization) of the developed modules and was able to demonstrate the progress that has been made to ensure that LAMP is able to comply with the requirements of PCS. In this regard, LAMP was shown to route an 8 kHz mono voice channel, as well as, operate both indoors and outdoors, through a modification made to its' flooding algorithm.

Conclusion

7.1 Dissertation Summery

An ad-hoc network is a network that consists of an autonomous set of wireless nodes. Since these nodes are liberated to roam about in a non-deterministic manner, routing protocols are required to be adaptive, distributed and employ a multi-hop scheme, so that each node may act as a router for each of its neighbours, causing a large degree of network connectivity.

However, the ad-hoc routing problem is diverse in nature and contains many contradictions. For instance, delivering packets with minimal delay and maximum reliability, usually results in large protocol overheads, which consume both energy and bandwidth. Therefore, a tradeoff (or compromise) is generally made, whereby one property is sacrificed for another. The task of the protocol designer is thus to find the optimal balance of all these constraints, so that the pitfalls of the resulting scheme may be negligible for the particular environment (and/or application) under consideration. Hence, many routing schemes currently exist today, with each offering a unique balance of such properties.

Proactive unicast protocols make use of a next-hop table to determine an implicit path to an intended destination node. Reactive schemes, on the other hand, first search the network space to establish a forwarding path and then perform routing. Thus, proactive schemes are able to routing data immediately, while reactive schemes are required to delay their routing sequence, until an appropriate path is found. Hence, proactive schemes are capable of routing data faster than their reactive counterparts, but are obligated to do so through the use of an "up-to-date" table, which contains knowledge of every node within the network. Since these tables become large for large networks, reactive schemes are favored over those of proactive (due to storing next-hop information when needed), making them both scalable and efficient.

The same balance is true for multicast protocols. While *source-based* algorithms construct a *spanning tree* for every multicast source present, *core-based* schemes only establish one *spanning tree* per multicast *group*. Thus, although *core-based* schemes consume less protocol overhead to maintain its' trees, they suffer from suboptimal forwarding paths and a single point of failure. Nevertheless, both these strategies were suited to slow moving networks, as each *spanning tree* required continual repair. Therefore, to improve the mobility constraint of multicasted networks, *mesh-based* algorithms were developed, which kept knowledge of multiple routes to each *group* member. Thus, when one route became broken due to mobility, other alternate routes were available. Although such schemes proved to be more resilient to mobility, they induced greater protocol overheads than previous schemes and hence were still deemed unacceptable in highly mobile environments. Therefore, new methodologies were required.

One such strategy was *blind-flooding*, which distributed a single copy of a packet to all nodes within the network, regardless of the indented recipients. But, this scheme re-transmitted a packet n times, where n was equal to the number of nodes present in the network. Since this lead to the *broadcast storm problem*, alternate schemes were constructed to limit forwarding nodes, so that all nodes could be reached in m transmissions, where m was less than n. However, such strategies were only feasible in large networks, since little performance could be gained over *blind-flooding* when employed in sparsely connected networks. Hence, for such

environments, another strategy had to be developed. This scheme became known as LAMP, a Location Aided Multicasting Protocol.

LAMP uses an underlying unicast protocol to identify specific next-hop neighbours that are suitable to forward a packet on towards the intended recipients. However, instead of sending a unicast transmission to each identified neighbour, LAMP broadcasts its' packets to all neighbouring nodes, so that a single transmission can be used instead. But, to achieve this, a list had to be appended to each packet, so that neighbouring nodes may determine if they are responsible for handling data at each successive hop. Thus, through this strategy, a shortest-hop *spanning tree* could be implicitly defined, allowing data to be routed over a minimum number of re-transmissions.

Had the underlying protocol been able to maintain an "up-to-date" state of all links within the network, then this strategy would suffice in forwarding multicast data to all intended recipients. However, in reality, proactive unicast schemes require a period of time (known as the update interval) to refresh links and thus a potential exists that some links have since become broken, causing invalid routing paths to occur. During such situations, LAMP employs a location limited flood, in order to discover alternate paths to a particular destination without exploiting unnecessary bandwidth. But, since flooding cannot guarantee that a path will be found, it is only suited to situations where the reception of a packet is delay critical, as used in time-bound audio applications.

Nevertheless, simulation results have shown LAMP to perform well as both a unicast and a multicast strategy, offering a suitable, alternate routing methodology to other pre-existing multicast and flood-limiting algorithms, which were deemed unacceptable to sparsely connected handheld terminals. But, since simulation only necessitates the formulation, testing and comparison of a protocol under various repeatable scenarios, it, alone, is insufficient for protocol verification. In addition, simulation can never represent the real world, no matter how accurate its' underlying protocol models are. Thus, to confirm a protocols' correct operation, physical implementation is imperative.

However, due to protocol simplifications made during simulation and the lack of dedicated hardware, difficulties can arise during the migration process. One such "difficulty" is the use of an operating system, which often requires routing protocols to conform to pre-defined interfaces. Thus, without prior knowledge of what these interfaces are, many simulated protocols never reach implementation or have to be modified, often resulting in the re-development of the whole protocol. Therefore, prior to developing LAMP, an appropriate operating system for high-end mobile devices was selected. This system, known as ARM Linux, facilitated the incorporation of kernel manipulation modules, which permitted calling functions to be altered during the transmission and reception of packets.

One such module was *netfilter*. *Netfilter* allows hooks to be placed throughout the kernel, so that additional rules can be applied to a packet, in order to determine whether it should be queued to a user-defined, background application for modification or sent via the usual IP routines. Hence, through this architecture, protocols such as LAMP can be implemented in user-space, giving the protocol developer all the necessary resources needed to readily incorporate new algorithms into ARM Linux.

7.2 Future Work

7.2.1 Scalability

Previous work on LAMP has been geared towards the development of a *table-driven* strategy for the identification of neighbouring nodes. However, *table-driven* strategies suffer from scalability issues, preventing such schemes from being deployed in large networks. Hence, to extend LAMP's application to bigger networks, this scheme needs to be replaced by some reactive forwarding methodology. But, to do so, a *spanning tree* would be necessary, since these structures have the ability to use one address, the multicast *group* address, to represent all intended recipients, thus negating the need of a next-hop forwarding list. Since this defeats the very incentive of LAMP, it is not perceived that LAMP will ever be used outside sparsely connected ad-hoc environments. Also, reactive strategies require a certain amount of time to search for the required recipients, in order to build the desired *spanning tree*. Since this time will depend on the number of nodes present in the network and the current congestion level, care will need to be taken to ensure the maximum latency bound of a packet does not exceed the delay that is acceptable for playback in real-time audio applications.

Therefore, future work should not focus on making LAMP scalable, but rather look at congestion issues, since congestion is related to both bandwidth and energy consumption.

7.2.2 Congestion

The main disadvantage of LAMP is that data is flooded until the periodic update interval of the underlying unicast protocol is able to fix broken next-hop links. Although this strategy permits the delivery of a packet during route failure, it induces high bandwidth consumption. Thus, to reduce the time between updates, a technique (borrowed from reactive protocols) can be adopted, whereby a reverse path is established in conjunction with the flooding process. This way, the flooding process will not only act as a temporary routing strategy, but also a repairing mechanism. Since repairs will now be made on an *on-demand* basis, flooding will be required less often, causing a better utilization of the available bandwidth.

Another technique than can be employed is the *distance effect* (See section 2.2.1.2.1), which allows the update interval to adapt to the current mobility conditions of the network. Since this will cause the underlying unicast protocol to respond quicker to link breakages, unnecessary flooding can be prohibited.

7.2.3 Implementation

In Chapter 3, an architecture, known as *netfilter*, was given to incorporate new routing schemes into the existing IP stack. However, what was not mentioned is that this method of protocol implementation is not the only scheme that can be used. An alternate solution is to implement protocols directly within the kernel [123]. However, implementing routing protocols in this manner relies on intricate knowledge of the Linux IP stack, making this strategy very operating system dependent. With constant developments still being made to both the ARM Linux kernel and the routing algorithm, it was predicted that portability issues could result, which would have detrimental effects to the progress of the project. Thus, to avoid this dependency, *netfilter* was selected instead.

Nevertheless, full kernel implementations do offer two advantages. Since packets can be manipulated directly, transportation to user-space is no longer required, permitting code to be both efficient and fast [124]. In addition, the use of an IP queue will no longer be necessary, causing savings in both the memory and processing requirements of the routing daemon.

Appendix A - Tracing Through the Linux IP Stack

This chapter gives a detailed description of how packets are sent/received to/from the 2.4.18 Linux kernel. Due to the rapid development of the Linux operating system, many documents describing the journey of packets through the IP stack were found to be obsolete, with the most recent detailing the 2.0.34 kernel [4]. However, with the aid of the Internet, many documents were found pertaining to either a particular aspect of the 2.4.x kernel or the changes made since kernel 2.2.x. Therefore, with reference to these documents, a thorough understanding of the responsibilities of each *netfilter* hook was gained, which aided in the incorporation of LAMP into the existing Linux framework. However, since details of the Linux 2.4.18 IP stack do not form the focus of the dissertation, it is expected that this endeavor will be placed in the public domain, so other research groups need not undergo this same work again.

Please note, however, that all the call directories indicated, are referenced with regards to the main Linux source code directory, usually found in `/usr/src/linux-2.4`.

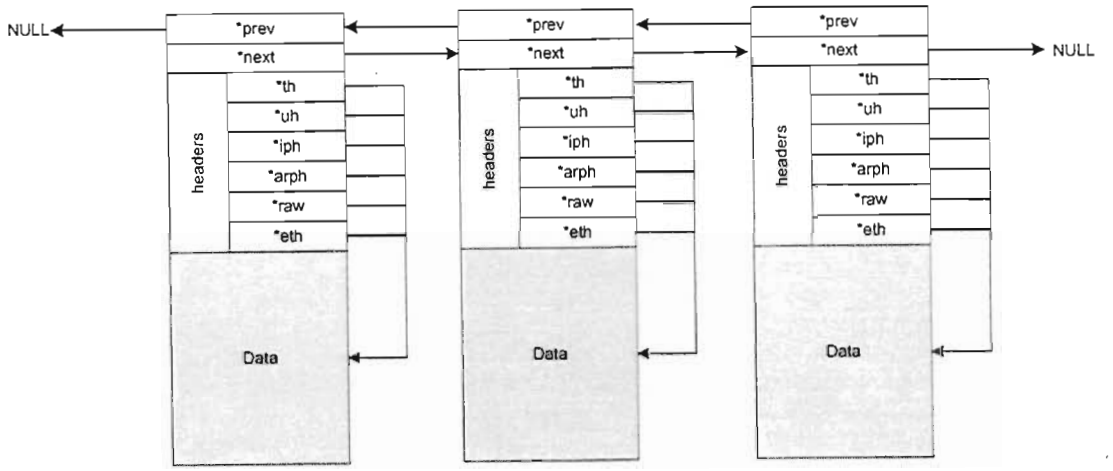
I.I Reception of a Packet

We begin this journey by first looking at the calls that are made within the kernel, which involve the reception of a packet. This sequence starts when the NIC senses the transmission of data over the air interface.

I.I.I Reception from the NIC to the IP Packet Handler

Since this project deals with wireless communication, all packets sensed within the NIC's radio range are received and transferred into memory. Once fully received, an interrupt request is generated by the NIC to get the attention of the interrupt service routine, which resides as part of the kernel network driver [125]. When the interrupt service routine is invoked, a test is performed to determine whether the packet is intended for this host. This is done by comparing the MAC address of host with the destination MAC address of the packet. Provided the packet was not broadcasted or the NIC was not put into promiscuous mode, only packets with matching MAC addresses are accepted, while others are ignored. Those accepted are allocated a `sk_buff` structure (through `dev_alloc_skb()`) and then invoked with `netif_rx()`. The `dev_alloc_skb()` function is responsible for copying the packet from the memory of the NIC to the newly allocated `skb` (a packet contained within the `sk_buff` structure). Note that there are a number of ways to achieve this, such as using DMA transfers, but details of this is not given here.

The `sk_buff` structure (defined in `/include/linux/skbuff.h`) is a double-linked buffer [107] that resides in the kernels' memory to store (and manage [126]) network packets to/from the socket interface. Essentially, the `sk_buff` structure is an efficient control structure that has a block of memory attached, with pointers to each header within its payload, as shown in Figure 99.



***th** (TCP header pointer)
***uh** (UDP header pointer)
***iph** (IP header pointer)
***arph** (ARP header pointer)
***raw** (raw IP header pointer)
***eth** (Ethernet header pointer)

Figure 99: Showing three double-linked socket buffer structures, each containing header pointers to the corresponding data section of the packet.

The `netif_rx()` (given in `/net/core/dev.c`) is a generic receive handler [125] that first timestamps the received packet [127] and then disables its interrupts. A test is then done to determine whether there is space to place the `skb` into a receive CPU FIFO queue, and, if so, the `skb` is queued and a software interrupt request (`softirq`) is scheduled by setting the NET_RX_SOFTIRQ flag (through `__cpu_raise_softirq(this_cpu, NET_RX_SOFTIRQ)` [128]); else the packet is dropped and the `skb` freed. Once a `softirq` has been raised, its interrupts are re-enabled and congestion level returned. Since this code resides within the interrupt service routine, it is required to be fast [125], so that awaiting packets of the NIC are not dropped. Thus the computational processing of the packet is delayed until the CPU(s) become idle. This is the purpose of the `softirq`, which are only triggered when the CPU is not busy handling any interrupts.

`Softirq`'s provide a means for multiple CPU (SMP) machines to process received network packets [127] and is handled through the `do_softirq()` function (located in `/kernel/softirq.c`). This routine basically determines the appropriate handling routine required (if any), through various defines in the kernel source code [128]. The routine that corresponds to the NET_RX_SOFTIRQ flag is `net_rx_action()`. Note that since `softirq`'s are only flagged, `do_softirq()` is required to be called regularly within the kernel, to ensure `softirq`'s are continually processed. This is achieved by placing the `do_softirq()` call in the following three positions [125]:

- In `do_IRQ()` (found in `/arch/i386/kernel/irq.c`), which is the generic interrupt handler,
- In `schedule()` (given in `/kernel/sched.c`), which is the main process (application) scheduling function,
- And just before the terminations of system calls (described later).

The task of the net_rx_action() function (defined in /net/core/dev.c) is to dequeue the first packet obtained from the FIFO queue and pass it to the appropriate protocol handler function. However, many protocol handlers may be registered with the kernel [125, 128] and thus each is called in sequence, starting with the generic protocol handlers (ETH_P_ALL) first. Nevertheless, the protocol handler that deals specifically with IP packets (ETH_P_IP) is called ip_rcv().

I.I.II From the IP Packet Handler to the UDP Packet Handler

ip_rcv() (located in /net/ipv4/ip_input.c) is the protocol handler responsible for processing IPv4 packets. Once ip_rcv() obtains a packet, it first checks to see that is was received properly. It achieves this by ensuring the following [127]:

- The packet is at least 20 bytes long (the size of the IP header),
- The IP version field contained within the IP header is indeed 4,
- The checksum field of the IP header corresponds to that re-calculated by ip_rcv().

If any of these tests fails, the packet is dropped, or else the size of the IP packet is recalculated and the skb trimmed, but only if the network driver had padded the buffer out with zero's [127]. Once resized, the first of the *netfilter* hooks is called, through the use of the "NF_HOOK (PF_INET, NF_IP_PRE_ROUTING, ip_rcv_finish)" macro. As can be seen by this macro, the packet is sent through the NF_IP_PRE_ROUTING hook, after which the function ip_rcv_finish() is invoked, but only if the hook returned with ACCEPT [107].

ip_rcv_finish() (defined in /net/ipv4/ip_input.c) deals with the routing aspects of incoming IP packets [125]. However, ip_rcv_finish() only deals with where to send the packet next, which is achieved by examining the skb->dst structure. The skb->dst->input structure is a function pointer that contains the function that is to receive the packet next. If this structure is blank (NULL), the ip_route_input() function is invoked to fill-in this structure. This is because, from this point on, the packet may continue to one of four possible routines, namely [127]:

- ip_local_deliver() (given in /net/ipv4/ip_input.c) is called when the packet is destined locally and hence will result in the IP packet continuing to transverse up the IP stack.
- ip_forward() (located in /net/ipv4/ip_forward.c) is called when the packet is required to be forwarded and hence this host is only acting as a router for this IP packet. Tracing of the IP forwarding routines are not covered in this description, but may be followed by examining [127].
- ip_error() (shown in /net/ipv4/route.c) is called when ip_route_input() was unable to determine a route for the packet.
- ip_mc_input() (found in /net/ipv4/ipmr.c) is called when the IP packet is intended for a multicast route.

Once the ip_route_input() function returns, ip_rcv_finish() makes a call to the function pointed by skb->dst->input. However, in order to gain a better understanding as to the conditions that must exist for a particular choice of function, an examination into the ip_route_input() function is required.

The ip_route_input() function (defined in /net/ipv4/route.c) is the function that inspects the cache memory of the host, in order to determine where the packet is to be processed

next. If one is found, the `skb->dst` structure is set and the `ip_route_input()` function is returned; else a check is made to determine whether the packet is part a multicast route. If so, `ip_route_input_mc()` is called, or else `ip_route_input_slow()` is invoked [129].

Since this description does not deal with multicasting, only `ip_route_input_slow()` will be examined. The `ip_route_input_slow()` function (given in `/net/ipv4/route.c`) looks through the routing tables (the Forward Information Base (FIB)) of a host [130], in order to find a route for the packet. Before doing so, however, a number of error checking routines are performed. If all these checks are passed, then the FIB lookup commences through the use of the `fib_lookup()` function [129]; else the `skb->dst->input` pointer is set to the `ip_error()` function. The `fib_lookup()` function is used to search through the routing table of the host, in order to find a route. If a route is found, `fib_lookup()` returns with a `res` (result) structure that indicates the type of route found; else res.type is set to RTN_UNREACHABLE. If the returned `res.type` structure contains either RTN_BROADCAST or RTN_LOCAL, the `skb->dst->input()` pointer is set to `ip_local_deliver()` [130]. Once a route is found within the FIB, it is placed within the quicker routing cache, so that it may be found by `ip_route_input()` on the next packet arrival.

Assuming `res.type` was indeed set to either RTN_BROADCAST or RTN_LOCAL, `ip_local_deliver()` is invoked by `ip_rcv_finish()`. The `ip_local_deliver()` function (located in `/net/ipv4/ip_input.c`) is the function that deals with IP fragmentation reassembly [125] (through `ip_defrag()`). Once the packet has been fully reassembled, a call is made to another *netfilter* hook, namely the "`NF_HOOK (PF_INET, NF_IP_LOCAL_IN, ip_local_deliver_finish)`" macro.

The `ip_local_deliver_finish()` function (found in `/net/ipv4/ip_input.c`) is responsible for completing any outstanding tasks that are required by IP, before the packet is transported to the INET protocol handlers [125]. But, before doing so, the IP header is trimmed and a test is performed to determine whether the packet is intended for a raw IP socket, in which case the `raw_v4_input()` function is invoked (see [125] for more details on raw IP packets). In most cases, however, the packet will not be sent to raw sockets and hence the proper INET protocol will need to be determined. This is done by taking the protocol field of the IP header and using it as an index into the `inet_protos[]` array. The returned `inet_protocol` structure (defined in `/include/net/protocol.h`) contains the correct protocol handler, which is then invoked. These protocol handler functions include `tcp_v4_rcv()`, `udp_rcv()`, `icmp_rcv()` and `igmp_rcv()`, corresponding to the TCP, UDP, ICMP and IGMP protocols, respectively [125]. Since this project deals with UDP packets only, just the `udp_rcv()` function will be considered.

I.I.III From the UDP Packet Handler to Userland Reception

The `udp_rcv()` function (defined in `/net/ipv4/udp.c`) is the UDP INET protocol handler. This function starts by performing some integrity checks to see if the packet length and checksum fields are correct. If so, then a further check is performed to determine whether the packet is part of a broadcast or multicast, in which case, the `udp_v4_mcast_deliver()` function is called; else the packet is intended as part of a unicast transmission and thus

udp_v4_lookup() is invoked to establish the receiving socket [131] from the port address of the packet. If udp_v4_lookup() was unable to find the correct socket, then udp_rcv() received a packet on a port that did not contain a corresponding receiving application and hence the packet is discarded [131]. However, if a socket was found, then udp_queue_rcv_skb() is called upon.

The udp_v4_mcast_deliver() function (given in /net/ipv4/udp.c) basically sends the packet to every socket that contains listeners. For each socket found, udp_queue_rcv_skb() is invoked.

The udp_queue_rcv_skb() function (found in /net/ipv4/udp.c) applies UDP kernel packet filtering (if defined) and then calls socket_queue_rcv_skb() [129].

The socket_queue_rcv_skb() function (located in /include/net/sock.h) places the UDP packet onto the socket's receive queue [131]. Each protocol family contains its own received queue and when this queue becomes full, the packet is dropped. Once queued, the function pointed to by sk->data_ready is invoked [131]. However, before doing so, a user defined filter may be applied to the packet, through the sk_filter() call. This function is only invoked if a filter was attached to the socket, through the setsockopt() system call [131]. Depending on the rules set for this filter, the packet may be discarded, trimmed or accepted with change [131].

In order to discover which function is actually called by sk->data_ready, one needs to examine the sock_init_data() function [132] (defined in /net/core/sock.c), which configures the socket with error, read & write queues during its initialization stage. On line 1194 of the source code, the assignment "sk->data_ready = sock_def_readable;" will be seen. Hence, the call will continue with the sock_def_readable() function.

The sock_def_readable() function (located in /net/core/sock.c) makes a call to the function wake_up_interruptible(), if the sockets' process has been put into sleep mode and the sleep queue for the socket is defined [132]. The wake_up_interruptible() function (given in /include/linux/sched.h) is actually an alias for the __wake_up() function. The __wake_up() function (found in /kernel/sched.c) simply calls the kernels' core scheduler wakeup function __wake_up_common() [132]. This function invokes try_to_wake_up() that wakes up process(s), by placing them on the run_queue and marking them as TASK_RUNNING. Once awoken, the packet is transferred to all processes (userland applications) waiting on the socket (via the sk_wake_async() function call [133, 134]). The reason application processes needed to be awoken is that receiving sockets are generally blocked until data becomes available for them. To see this occurrence, one needs to follow the calls that occur when the user executes the "read (socket, data, length)" statement.

I.I.IV Blocked Sockets

The "read (socket, data, length)" (defined in the libc.a C library) statement causes libc to issue a system call software interrupt, via the 0x80 assembly instruction [135]. The interrupt handler (defined in /arch/i386/kernel/entry.S) then identifies which system call is requested, by examining the EAX register of the CPU. The value of this register can be found in /include/asm/unistd.h, which defines the values for each system call. The system call value that involves the "read ()" statement is __NR_read. This value is then used

as an index into the sys_call_table [135] array, which calls the sys_read() system function.

The sys_read() function (found in /fs/read_write.c) is part of the kernels' Virtual File System [107]. It tests whether the given file descriptor has the FMODE_READ flag set, which is used to inform the kernel that the user has opened this file (socket) with read permissions enabled, and that the data buffer (contained within userland) is able to be accessed (using the locks_verify_area() function). If this flag is indeed set, the file->f_op->read function is invoked. This function can be determined by looking at sock_map_fd() function (given in /net/socket.c) that is invoked during the creation of the socket. There one will find the following code assignment: "file->f_op = sock->inode->i_fop = &socket_file_ops;". The socket_file_ops structure is defined (in /net/socket.c) as follows:

```
struct file_operations socket_file_ops {
        :
      read:  sock_read,
      write: sock_write,
        :
};
```

Hence, the sys_read() function will call the sock_read() function.

At this point, the user call has entered into the socket layer. The sock_read() function (located in /net/socket.c) is the read function written for BSD sockets [109]. It calls socki_lookup() to associate the sockets data structure with the file descriptors' inode [134]. Once associated, the "data" buffer allocated in userland is then placed in a msg structure (so that it may be transported into kernel space) and the function sock_recvmsg() invoked.

The sock_recvmsg() function (given in /net/socket.c) allocates a blank socket control message that accompanies the msg structure throughout its journey. Once allocated, the protocol specific sock->ops->recvmsg() function is called [136]. If this function returns without errors, then the accompanying control message is sent to the scm_recv() function for further processing. The sock->ops->recvmsg() function is defined during kernel startup, through the inet_stream_ops and inet_dgram_ops structures (found in /net/ipv4/af_inet.c). Both of these structures direct the sock->ops->recvmsg() function to the inet_recvmsg() function [134], hence this is the function that is actually invoked next.

The inet_recvmsg() function (located in /net/ipv4/af_inet.c) is the receiving function for the INET layer. It calls the sk->prot->recvmsg(), which points to the TCP, UDP or RAW message receive functions (registered by the inetsw_array[] array structure during kernel startup), depending on the socket type used [136]. Since this project deals just with UDP packets, only the udp_recvmsg() function will be considered.

The udp_recvmsg() function (defined in /net/ipvv4/udp.c) checks that a valid address was passed to it and then invokes skb_recv_datagram(). Once this function returns correctly, the received data is copied to userland through either skb_copy_datagram_iovec() or skb_copy_and_csum_datagram_iovec(), depending on whether a checksum is required [133]. Finally the sock_recv_timestamp() function is

called, the datagram freed (from the sockets' receive buffer), and the number of bytes retrieved returned.

However, one will be asking how the above trace shows the user process going to sleep. This is shown by examining the `skb_recv_datagram()` function (given in `/net/core/datagram.c`). It contains a code snippet that resembles the following:

```
{
     :
    do {
        :
        skb = skb_dequeue (&sk->receive_queue);
        if (skb)
            return skb;
        error = -EAGAIN;
        if (!timeo)
            goto no_packet;
    } while (wait_for_packet (sk, err, &timeo) == 0);
    return NULL;

no_packet:
    *err = error;
    return NULL;
}
```

Here `skb_dequeue()` is called to return the first packet stored in the sockets receive queue, and `wait_for_packet()` is invoked to place the process onto the `sk->sleep` wait queue, marking it as TASK_INTERRUPTIBLE [136].

I.II Transmission of a Packet

We now look at how the kernel deals with the transmission of a packet from userland. This sequence of calls starts when a running application issues a "`write`" system call to a socket.

I.II.I Transmission from Userland to the UDP Packet Handler

The "`write (socket, data, length)`" (defined in the `libc.a` C library) statement causes `libc` to issue the 0x80 assembly instruction [135], which invokes the interrupt handler (found in `/arch/i386/kernel/entry.S`) with an index of `__NR_write` (given in `/include/asm/unistd.h`). When this index is placed within the `sys_call_table` [135] array, the `sys_write()` system function is called [137].

The `sys_write()` function (located in `/fs/read_write.c`) is part of the kernels' Virtual File System [107]. It tests whether the given file descriptor has the FMODE_WRITE flag set, which is used to inform the kernel that the user has opened this file (socket) with write permissions enabled, and that the data buffer (contained within userland) is able to be accessed (using the `locks_verify_area()` function). If this flag is indeed set, the `file->f_op->write` function is invoked. As mentioned previously, this points to the `sock_write()` function.

At this point, the user's call has entered into the socket layer. The `sock_write()` function (defined in `/net/socket.c`) is the write function written for BSD sockets [109]. It checks that "length" is not zero and, if true, calls `socki_lookup()` to associate the sockets' data structure with the file descriptors' `inode`. Once associated, the "data" contained within the userland

memory is then placed in a `msg` structure (so that it may be transported into kernel space) and the function `sock_sendmsg()` invoked [136].

The `sock_sendmsg()` function (given in `/net/socket.c`) attaches a socket control message for the `msg` structure, through the `scm_send()` function. If allocated correctly, the protocol specific `sock->ops->sendmsg()` function is called [137] and socket control message destroyed. The `sock->ops->sendmsg()` function is defined during kernel startup, through the `inet_stream_ops` and `inet_dgram_ops` structures (located in `/net/ipv4/af_inet.c`). Both of these structures direct the `sock->ops->sendmsg()` function to the `inet_sendmsg()` function [107], hence this is the function that is actually invoked next.

The `inet_sendmsg()` function (found in `/net/ipv4/af_inet.c`) is the sending function for the INET layer. It calls the `sk->prot->sendmsg()`, which points the UDP message sending function `udp_sendmsg()`, through the `inetsw_array[]` array structure indicated earlier [109].

I.II.II From the UDP Packet handler to the IP Packet Handler

The `udp_sendmsg()` function (defined in `/net/ipvv4/udp.c`) performs numerous checks to ensure that valid address and data lengths were passed to it. If these tests return without error, another series of tests are initiated to determine whether `sk_dst_check()` or `ip_route_output()` is required to find a route to the destination host [130]. Once a route has been found, segments of the fake UDP header is constructed and `ip_build_xmit()` invoked [136]. The reason a fake header is constructed is that selected fields of the IP header is required for the UDP checksum calculation (see section 3.5.3).

The `sk_dst_check()` function (located in `/include/net/sock.h`) is used to search the sockets' cache memory for a route [130], while `ip_route_output()` is used to search the relevant routing tables of the host [137]. Since routing information is more likely to be found in the routing tables, `sk_dst_check()` is only invoked if `udp_sendmsg()` is sure that a route exists in the socket cache, and hence will find a route there relatively faster.

The `ip_route_output()` function (found in `/include/net/route.h`) is used to create a key [130], which is needed to index a particular chain within the routing cache (hash table) of a host. Once the key has been created, control is handed over to `ip_route_output_key()` to actually search for a route. The `ip_route_output_key()` function (shown in `/net/ipv4/route.c`) iterates through the selected bucket, until either a route was found or the chain ended [130]. If a route was found, the expiry time of this entry within the chain is updated and the function returned; else `ip_route_output_slow()` is invoked to find a route within the hosts' FIB.

The `ip_route_output_slow()` function (given in `/net/ipv4/route.c`) is responsible for searching the FIB and placing the resolved route into the routing cache, if found [130]. First tests are performed on the source to determine whether the address is part of a multicast, badclass or zero net, and if so, terminates the search and returns with the appropriate error. Next, the outgoing device is determined, followed by further testing to determine whether the destination is a broadcast address, part of a multicasting address, or the local loopback address (IP address 127.0.0.1) [130]. If this was the case, a search through the FIB is skipped, or else a search is initiated through either `fib_lookup()` or `main_table->tb_lookup()`. Once a

route is found, it is placed within the route cache through `rt_intern_hash()`, which also invokes ARP to resolve the MAC address of the next hop router [130]. Note that `fib_lookup()` (defined in `/include/net/ip_fib.h`) tries to determine the outgoing router by applying the more specific network masks first, before moving on towards the more general ones [129]. If, after applying all the network masks, the router is still not resolved, the default router is used instead (if defined) [130].

I.II.III From the IP Packet Handler to the Network Driver

The `ip_build_xmit()` function (found in `/net/ipv4/ip_output.c`) starts by issuing a test to determine whether the IP header has been attached. If it has not, the packet length is increased to include the IP header, after which a further tests is performed to determine whether IP fragmentation is required [130]. If so, the packet is sent to `ip_build_xmit_slow()` for further processing; else the hardware (Ethernet) header is allocated and the IP header built (if necessary). Once built, the "`NF_HOOK (PF_INET, NF_IP_LOCAL_OUT, output_maybe_reroute)`" macro is called [130].

The `ip_build_xmit_slow()` function (given in `/net/ipv4/ip_ouput.c`) builds and sends fragments, starting with the segment containing the highest offset (which helps for both the transmission and the reception of fragments, since the fake UDP header checksum can be calculated and sent on the last fragment, through `getfrag ()`). This routine starts by calculating the number of fragments required and then repeats the following steps for each fragment:

- The hardware header is allocated,
- The IP header is built,
- The offset is adjusted for the next fragment,
- The "`NF_HOOK (PF_INET, NF_IP_LOCAL_OUT, output_maybe_reroute)`" macro is called.

The `output_maybe_reroute()` function (located in `/net/ipv4/ip_output.c`) is defined as follows [130]:

```
static inline int output_maybe_reroute(struct sk_buff *skb) {
      return skb->dst->output(skb);
}
```

As one can see, this function simply calls `skb->dst->output` [136]. The reason `output_maybe_reroute()` was invoked within the *netfilter* hook and not `skb->dst->output` is simply for precaution, since the routing daemon called by *netfilter* may change the route and hence affect the function pointed by `skb->dst->output`.

To determine where the code proceeds next, one needs to look back at the `ip_route_output_slow()` function, since it was the function that set this pointer. There `skb->dst->output` is set to the following [136]:

- `ip_output()` for unicasts,
- `ip_mc_output()` for both broadcasts and multicasts,
- `ip_rt_bug()` for bugs that may have occurred somehow (not considered here).

Since this discussion does not include multicasting, only the broadcasting aspects of `ip_mc_output()` will be considered. The `ip_mc_output()` function (defined in `/net/ipv4/ip_output.c`) creates a duplicate copy of the broadcasting packet (a making a clone of the socket buffer structure) and invokes the copy through the "`NF_HOOK (PF_INET, NF_IP_POST_ROUTING, ip_dev_loopback_xmit)`" macro. Once the copy has been sent, the original packet is sent to `ip_finish_output()`. Note that the `ip_dev_loopback_xmit()` function (given in `/net/ipv4/ip_output.c`) simply sends the duplicate packet to the loopback device, so that local applications can receive a copy of the broadcasted data.

The `ip_output()` function (located in `/net/ipv4/ip_output.c`) performs NAT (Network Address Translation), provided the Linux operating system was configured with this option, and then makes a call to `ip_finish_output()`.

The `ip_finish_output()` function (found in `/net/ipv4/ip_output.c`) sets the MAC layer output device and protocol type (ETH_P_IP), and then invokes the "`NF_HOOK (PF_INET, NF_IP_POST_ROUTING, ip_finish_output2)`" macro.

The `ip_finish_output2()` function (given in `/net/ipv4/ip_output.c`) calls either `hh->hh_output` or `dst->neighbour->output`, depending on whether `hh` or `dst->neighbour` was defined, respectively [130]. To discover where these function point, one needs to look at the ARP constructor (`arp_constructor()`), which is invoked during the ARP initialization stage of the kernel startup mechanism [128]. The `arp_constructor()` function (located in `/net/ipv4/arp.c`) attaches different `neigh_ops` structures that define both of the above function calls [128]. Generally, these will point to `dev_queue_xmit()`.

I.II.IV From the Network Driver to Transmission

The `dev_queue_xmit()` function (defined in `/net/core/dev.c`) is responsible for queuing a given `skb` for transmission by the NIC. If the NIC does not support checksumming, the MAC layer checksum is first computed; else the checksum is left for the NIC. Next a test is done to determine whether the output device (set by `ip_finish_output()`) contains a queue, and if so, invokes its enqueue function (`q->enqueue`) and then makes a call to `qdisc_run()` [136], or else, if the output device is UP (IFF_UP), then `dev_queue_xmit()` will try to send the packet directly, by invoking `dev->hard_start_xmit()`; else the packet is dropped [129]. Depending on the NIC driver used, `dev->hard_start_xmit()` will be mapped to some driver specific transmission handler function [128]. The `qdisc_run()` function (given in `/include/net/pkt_sched.h`) attempts to flush the outgoing device queue, by repeatedly calling `qdsc_restart()`, which dequeues a `skb` from the queue and then calls `dev->hard_start_xmit()` [129].

Due to the `dev->hard_start_xmit()` function being driver specific, a generic description of its functionality is very difficult. However, the network driver will generally re-queue a `skb` and invoke `netif_schedule()` (located in `/include/linux/netdevice.h`) to raise a

NET_TX_SOFTIRQ, if the device is busy; else it will copy the packet to the memory of the NIC and transmit it [129]. The `do_softirq()` function (described previously) will then see the NET_TX_SOFTIRQ flag set and thus initiate `net_tx_action()`. The `net_tx_action()` function (given in `/net/core/dev.c`) will then test if it can get a lock on the device queue and, if so will call `qdisc_run()`; else it will re-schedule the `softirq` (through `netif_schedule()`), to try to send the packet again, until it is eventually sent.

References

[1] S. Corson and J. Macker, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations," Request For Comments 2501, January 1999.

[2] B. A. Chambers, "The Grid Roofnet: a Rooftop Ad Hoc Wireless Network," in *Electrical Engineering and Computer Science*: Massachusetts Institute of Technology (MIT), 2002.

[3] R. Morris, J. Jannotti, F. Kaashoek, J. Li, and D. De Couto, "CarNet: A Scalable Ad Hoc Wireless Network System," in *proc. Association for Computing Machinery (ACM) Special Interest Group on Communications (SIGCOMM)*, 2000.

[4] S. T. Satchell and H. B. J. Clifford, *Linux IP Stacks Commentary: Guide to Gaining Insider's Knowledge on the IP Stacks of the Linux Code*: Coriolis Open Press, 2000.

[5] S.-J. Lee, M. Gerla, and C.-K. Toh, "A Simulation Study of Table-Driven and On-Demand Routing Protocols for Mobile Ad Hoc Networks," *IEEE Network Magazine*, 1999.

[6] J. Walrand and P. Varaiya, *High-Performance Communication Networks*, Second ed: Morgan Kaufmann, 2000.

[7] M. Corson, S. Batsell, and J. Macker, "Architectural Considerations for Mobile Mesh Networking," 1996, available from http://tonnant.itd.nrl.navy.mil/mmnet/mmnetRFC.txt.

[8] E. M. Royer and C.-K. Toh, "A Review of Current Routing Protocols for Ad Hoc Wireless Mobile Networks," *IEEE Personal Communications*, pp. 46-55, 1999.

[9] S. R. Das, R. Castaneda, J. Yan, and R. Sengupta, "Comparative Performance Evaluation of Routing Protocols for Mobile, Ad hoc Networks," in *proc. IEEE International Conference on Computer Communications and Networks (ICCCN)*, 1998.

[10] M. Mauve, J. Widmer, and H. Hartenstein, "A Survey on Position-Based Routing in Mobile Ad-hoc Networks," *IEEE Network Magazine*, vol. 15, no. 6, pp. 30-39, 2001.

[11] E. D. Kaplan, *Understanding GPS: Principles and Applications*: Artech House Inc., 1996.

[12] S. Capkun, M. Hamdi, and J.-P. Hubaux, "GPS-free positioning in mobile Ad-Hoc networks," in *proc. 34th Hawaii International Conference on System Sciences*, 2001.

[13] J. Hightower and G. Borriello, "Location Systems for Ubiquitous Computing," *IEEE Computer Society Press*, vol. 34, no. 8, pp. 57-66, 2001.

[14] Z. J. Haas and B. Liang, "Ad Hoc Mobility Management With Uniform Quorum Systems," *IEEE ACM Transactions on Networking*, vol. 7, 1999.

[15] N. K. Guba and T. Camp, "GLS: a Location Service for an Ad Hoc Network," in *proc. Grace Hopper Celebration*, 2002.

[16] S. Basagni, I. Chlamtac, V. Syrotiuk, and B. Woodward, "A Distance Routing Effect Algorithm for Mobility (DREAM)," in *proc. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 1998.

[17] H. Takagi and L. Kleinrock, "Optimal Transmission Ranges for Randomly Distributed Packet Radio Terminals," *IEEE Transactions on Communications*, vol. COM-32, no. 3, pp. 246-257, 1984.

[18] T.-C. Hou and V. O. K. Li, "Transmission Range Control in Multihop Packet Radio Networks," *IEEE Transactions on Communications*, vol. COM-34, pp. 38-44, 1986.

[19] E. Kranakis, H. Singh, and J. Urrutia, "Compass Routing on Geometric Networks," in *proc. 11th Canadian Conference on Computational Geometry*, 1999.

[20] R. Nelson and L. Kleinrock, "The Spatial Capacity of a Slotted ALOHA Multihop Packet Radio Network with Capture," *IEEE Transactions on Communications*, vol. COM-32, no. 6, pp. 684-694, 1984.

[21] B. Karp and H. T. Kung, "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks," in *proc. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[22] G. Toussaint, "The relative neighborhood graph of a finite planar set," *Pattern Recognition*, vol. 12, pp. 261-268, 1980.

[23] K. Gabriel and R. Sokal, "A new statistical approach to geographic variation analysis," *Systematic Zoology*, vol. 18, pp. 259-278, 1969.

[24] T. Camp, J. Boleng, B. Williams, L. Wilcox, and W. Navidi, "Performance Comparison of Two Location Based Routing Protocols for Ad Hoc Networks," in *proc. IEEE INFOCOM - The Conference on Computer Communications*, 2002.

[25] Y.-B. Ko and N. H. Vaidya, "Location-Aided Routing (LAR) in Mobile Ad Hoc Networks," in *proc. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 1998.

[26] K. Wu and J. Harms, "Location Trace Aided Routing in Mobile Ad Hoc Networks," in *proc. International Conference on Computer Communications and Networks (ICCCN)*, 2000.

[27] C. E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," *Computer Communications Review*, pp. 234-244, 1994.

[28] C.-C. Chiang, H.-K. Wu, W. Liu, and M. Gerla, "Routing In Clustered Multihop, Mobile Wireless Networks With Fading Channel," in *proc. IEEE Singapore International Conference (SICON)*, 1997.

[29] A. Iwata, C.-C. Chiang, G. Pei, M. Gerla, and T.-W. Chen, "Scalable Routing Strategies for Ad hoc Wireless Networks," *IEEE Journal on Selected Areas in Communications*, pp. 1369-1379, 1999.

[30] A. Ephremides, J. E. Wieselthier, and D. J. Baker, "A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling," in *proc. Proceedings of the IEEE*, 1987.

[31] M. Gerla and J. T.-C. Tsai, "Multicluster, Mobile, Multimedia Radio Network," *ACM-Baltzer Journal of Wireless Networks*, vol. 1, no. 3, pp. 255-265, 1995.

[32] A. D. Amis, R. Prakash, D. Huynh, and T. Vuong, "Max-Min D-Cluster Formation in Wireless Ad Hoc Networks," in *proc. IEEE INFOCOM - The Conference on Computer Communications*, 2000.

[33] A. D. Amis and R. Prakash, "Load-Balancing Clusters in Wireless Ad Hoc Networks," in *proc. IEEE Symposium on Application-Specific Systems and Software Engineering (ASSET)*, 2000.

[34] J. Sucec and I. Marsic, "Clustering Overhead for Hierarchical Routing in Mobile Ad hoc Networks," in *proc. IEEE INFOCOM - The Conference on Computer Communications*, 2002.

[35] S. Murthy and J. J. Garcia-Luna-Aceves, "An Efficient Routing Protocol for Wireless Networks," *ACM Mobile Networks and Applications Journal*, pp. 183-197, 1996.

[36] T.-W. Chen and M. Gerla, "Global State Routing: A New Routing Scheme for Ad-hoc Wireless Networks," in *proc. IEEE International Conference on Communications (ICC)*, 1998.

[37] L. Kleinrock and K. Stevens, "Fisheye: A Lenslike Computer Display Transformation," Computer Science Department of UCLA, Technical Report 1971.

[38] J. Postel, "The Internet Protocol Specification," USC/Information Sciences Institute, Request For Comments 791, September 1981.

[39] C. E. Perkins and E. M. Royer, "Ad-hoc On-Demand Distance Vector Routing," in *proc. IEEE Workshop on Mobile Computer Systems and Applications*, 1999.

[40] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," *Mobile Computing*, pp. 153-181, 1996.

[41] S. R. Das, C. E. Perkins, and E. M. Royer, "Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks," in *proc. IEEE INFOCOM - The Conference on Computer Communications*, 2000.

[42] M. S. Corson and A. Ephremides, "A Distributed Routing Algorithm for Mobile Wireless Networks," *ACM/Baltzer Wireless Networks Journal*, vol. 1, no. 1, pp. 61-81, 1995.

[43] V. D. Park and M. S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," in *proc. IEEE INFOCOM - The Conference on Computer Communications*, 1997.

[44] E. Gafni and D. Bertsekas, "Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology," *IEEE Transactions on Communications*, 1981.

[45] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva, "A Performance Comparison of Multi-Hop Wireless Ad Hoc Networking Routing Protocols," in *proc. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 1998.

[46] C.-K. Toh, "A Novel Distributed Routing Protocol To Support Ad-Hoc Mobile Computing," in *proc. IEEE 15th Annual International Phoenix Conference on Computers and Communication*, 1996.

[47] R. Dube, C. D. Rais, K.-Y. Wang, and S. K. Thripathi, "Signal Stability based Adaptive Routing (SSA) for Ad-Hoc Mobile Networks," *IEEE Personal Communications*, pp. 36-45, 1997.

[48] Z. J. Haas and M. R. Pearlman, "The Zone Routing Protocol (ZRP) for Ad Hoc Networks," IETF, Internet Draft, November 1997.

[49] J. Moy, "OSPF Version 2," Ascend Communications Incorporated, Request For Comments 2328, April 1998.

[50] G. Malkin, "RIP Version 2," Bay Networks, Request For Comments 2453, November 1998.

[51] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "Internet Group Management Protocol, Version 3," Request For Comments 3376, October 2002.

[52] Internet Assigned Numbers Authority, "Internet Multicast Addresses," 2003, available from http://www.iana.org/assignments/multicast-addresses.

[53] D. Waitzman, C. Partridge, and S. Deering, "Distance Vector Multicast Routing Protocol," Request For Comments 1075, November 1988.

[54] A. Adams, J. Nicholas, and W. Siadak, "Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised)," IETF, Internet Draft, February 2003.

[55] S. Deering, "Multicast Routing in Internetworks and Extended LANs," in *proc. ACM Special Interest Group on Data Communications (SIGCOMM)*, 1988.

[56] Y. K. Dalal and R. M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM*, vol. 21, no. 12, pp. 1040-1048, 1978.

[57] J. Moy, "Multicast Extensions to OSPF," Request For Comments 1584, March 1994.

[58] C.-K. Toh, *Ad Hoc Mobile Wireless Networks: Protocols and Systems*: Prentice Hall, 2002.

[59] A. J. Ballardie, P. F. Tsuchiya, and J. Crowcroft, "Core Based Trees (CBT)," in *proc. ACM Special Interest Group on Data Communications (SIGCOMM)*, 1993.

[60] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas, "Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)," IETF, Internet Draft, March 2003.

[61] S.-J. Lee, W. Su, J. Hsu, M. Gerla, and R. Bagrodia, "A Performance Comparison Study of Ad Hoc Wireless Multicast Protocols," in *proc. Annual Joint Conference of the IEEE Computer and Communications Societies*, 2000.

[62] S.-J. Lee, W. Su, and M. Gerla, "On-Demand Multicast Routing Protocol (ODMRP) for Ad Hoc Networks," IETF, Internet Draft, June 1999.

[63] C.-C. Chiang, M. Gerla, and L. Zhang, "Forwarding Group Multicast Protocol (FGMP) for Multihop, Mobile Wireless Networks," *Baltzer Cluster Computing*, vol. 1, no. 2, pp. 187-196, 1998.

[64] Y.-B. Ko and N. H. Vaidya, "Location-Based Multicast in Mobile Ad Hoc Networks," Department of Computer Science at the Texas A&M University, Technical Report 98-018, September 1998.

[65] M. R. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*: W.H. Freeman, 1979.

[66] M. Gerla, C.-C. Chiang, and L. Zhang, "Tree Multicast Strategies in Mobile, Multihop Wireless Networks," *ACM/Balzter Mobile Networks and Applications Journal*, 1998.

[67] C.-K. Toh, G. Guichala, and S. Bunchua, "ABAM: On-Demand Associativity-Based Multicast Routing for Ad Hoc Mobile Networks," in *proc. IEEE International Conference on Vehicular Technology*, 2000.

[68] E. M. Royer and C. E. Perkins, "Multicast Operation of the Ad-hoc On-Demand Distance Vector Routing Protocol," in *proc. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

[69] T. Kunz and E. Cheng, "Multicasting in Ad-Hoc Networks: Comparing MAODV and ODMRP," in *proc. International Conference on Distributed Computing Systems*, 2002.

[70] P. M. Mohan, J. J. Johnson, K. Murugan, and V. Ramachandran, "A Comparative and Performance Study of On Demand Multicast Routing Protocols for Ad Hoc Networks," in *proc. International Conference on High Performance Computing*, 2002.

[71] M. Liu, R. R. Talpade, A. McAuley, and E. Bommaiah, "AMRoute: Adhoc Multicast Routing Protocol," University of Maryland and the Institute for Systems Research, Technical Report CSHCN T.R. 99-1(ISR T.R. 99-8), 1999.

[72] H. Eriksson, "MBONE: The Multicast Backbone," *Communications of the ACM*, vol. 37, no. 8, pp. 54-60, 1994.

[73] C. W. Wu and Y. C. Tay, "AMRIS: A Multicast Protocol for Ad hoc Wireless Networks," in *proc. IEEE Military Communications Conference (MILCOM)*, 1999.

[74] J. J. Garcia-Luna-Aceves and E. L. Madruga, "The Core-Assisted Mesh Protocol," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 8, pp. 1380-1394, 1999.

[75] K. Obraczka and G. Tsudik, "Multicast Routing Issues in Ad hoc Networks," in *proc. IEEE International Conference on Universal Personal Communications (ICUPC)*, 1998.

[76] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The Broadcast Storm Problem in a Mobile Ad Hoc Network," in *proc. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

[77] B. Williams and T. Camp, "Comparison of Broadcasting Techniques for Mobile Ad Hoc Networks," in *proc. ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC)*, 2002.

[78] F. Tobagi and L. Kleinrock, "Packet Switching in Radio Channels: Part II - The Hidden Terminal Problem in Carrier Sense Multiple-Access and the Busy-Tone Solution," *IEEE Transactions on Communications*, vol. COM-23, no. 12, pp. 1417-1433, 1975.

[79] H. Lim and C. Kim, "Multicast Tree Construction and Flooding in Wireless Ad Hoc Networks," in *proc. ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM)*, 2000.

[80] W. Peng and X. Lu, "On the Reduction of Broadcast Redundancy in Mobile Ad Hoc Networks," in *proc. ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC)*, 2000.

[81] A. Qayyum, L. Viennot, and A. Laouiti, "Multipoint Relaying: An Efficient Technique for Flooding in Mobile Wireless Networks," INRIA, Technical Report RR-3898, March 2000.

[82] W. Peng and X. Lu, "AHBP: An Efficient Broadcast Protocol for Mobile Ad Hoc Networks," *Journal of Science and Technology*, 2002.

[83] G. Dommety and R. Jain, "Potential Networking Applications of Global Positional Systems (GPS)," The Ohio State University Computer Science Dept., Technical Report TR-24, April 1996.

[84] Compaq Information Technologies Group, "QuickSpecs - Compaq iPAQ Pocket PC H3800 Series," 2002, available from http://h18000.www1.hp.com/products/quickspecs/10977_na/10977_na.HTML.

[85] J. Epplin, "Exploring Linux PDA Software Alternatives," 2001, available from http://www.linuxdevices.com/articles/AT3058975992.html.

[86] Cygnus Solutions, "eCos (Embedded Cygnus Operating System)," 2003, available from www.cygnus.com.

[87] Wind River, "VxWorks," 2003, available from http://www.windriver.com/products/vxworks5/index.html.

[88] QNX Software Systems LTD, "QNX Real-Time Operating Systems," 2003, available from http://www.qnx.com/.

[89] Lynx Works, "LynxOS® RTOS," 2003, available from http://www.lynuxworks.com/products/lynxos/lynxos.php3.

[90] C. Halsall, "Linux on an iPAQ," 2001, available from http://linux.oreillynet.com/pub/a/linux/2001/06/01/linux_ipaq.html.

[91] The Familiar Project, "The Familiar Linux Distribution," 2003, available from http://familiar.handhelds.org/.

[92] Century Software, "The Microwindows Project," 2002, available from http://embedded.centurysoftware.com/.

[93] Trolltech, "The Qt/Embedded Environment," 2003, available from http://www.trolltech.com/products/embedded/.

[94] Transvirtual Technologies Inc., "Pocket Linux," 2003, available from http://www.pocketlinux.org/.

[95] Transvirtual Technologies Inc., "The Kaffe Java Virtual Machine," 2002, available from http://www.kaffe.org/.

[96] M. Hauben, "History of ARPANET," 1994, available from http://www.dei.isep.ipp.pt/docs/arpa.html.

[97] D. M. Ritchie, "The Evolution of the Unix Time-sharing System," 1984, available from http://cm.bell-labs.com/cm/cs/who/dmr/hist.html.

[98] The Trustees of Indiana University, "History of UNIX," 1996, available from http://www.uwsg.iu.edu/usail/concepts/unixhx.html.

[99] Bell-Labs, "The Creation of the UNIX Operating System," 2002, available from http://www.bell-labs.com/history/unix/.

[100] D. Teare, "Designing Cisco Networks," 1999, available from http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/introint.htm.

[101] D. D. Clark, "Realizing the Information Future - A Recent National Research Council Report," 1994, available from http://web.mit.edu/comm-forum/www/forums/941110-S.htm.

[102] The IEEE Standard 802.11, "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications," The Institute of Electrical and Electronics Engineers, Inc 1999.

[103] The IEEE Standard 802.11(b), "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band," The Institute of Electrical and Electronics Engineers, Inc 1999.

[104] The IEEE Standard 802.2, "Logical Link Control," The Institute of Electrical and Electronics Engineers, Inc 1998.

[105] J. Postel, "The User Datagram Protocol," USC/Information Sciences Institute, Request For Comments 768, August 1980.

[106] G. Insolvibile, "Linux Socket Filter: Sniffing Bytes over the Network - Kernel Korner - Issue 86," 2001, available from http://www.linuxjournal.com.

[107] C. Rodrigues, "Netfilter Paper," 2000, available from http://www.gis.net/~craigr/netfilter_paper.pdf.

[108] B. Hall, "Beej's Guide to Network Programming - Using Internet Sockets," 2001, available from www.ecst.csuchico.edu/~beej/guide/net/.

[109] M. Beck, U. Kunitz, R. Magnus, M. Dziadzka, and D. Verworner, *Linux Kernel Internels*, Second ed: Addison-Wesley, 1997.

[110] P. Russell, "Linux 2.4 NAT HOWTO," 2001, available from http://www.netfilter.org/unreliable-guides/NAT-HOWTO/.

[111] N. J. Dearham and S. A. McDonald, "A Handheld Implementation of a Location Aided Multicasting Protocol (LAMP)," in *proc. Military Information and Communications Symposium of South Africa (MICSSA)*, 2003. <Accepted for Publication>

[112] S. McCanne and S. Floyd, "*NS* - Network Simulator," 2003, available from http://www-mash.cs.berkeley.edu/ns/.

[113] S. Keshav, "REAL: A Network Simulator," Computer Science Department of UC Berkeley, Technical Report 88/472, 1988.

[114] D. F. Bacon, A. Dupuy, J. Schwartz, and Y. Yemini, "NEST: A Network Simulation and Prototyping Testbed," in *proc. Winter USENIX Technical Conference*, 1988.

[115] K. Fall and K. Varadhan, "The *NS* Manual," 2003, available from http://www-mash.cs.berkeley.edu/ns/.

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

[117] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala, "Improving Simulation for Network Research," University of Southern California, Los Angeles, Technical Report 99-702, 1999.

[118] The CMU Monarch Project, "Wireless and Mobility Extensions to *NS*," Computer Science Department, Carnegie Mellon University, Technical Report 1999.

[119] J. Elson and D. Estrin, "Time Synchronization for Wireless Sensor Networks," in *proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[120] H. Baker, "Computing "Great Circle Distances" from Latitudes and Longitudes," 1995, available from http://home.pipeline.com/~hbaker1/FAQ-lat-long.txt.

[121] P. Johansson, T. Larsson, and N. Hedman, "Scenario-based Performance Analysis of Routing Protocols for Mobile Ad-hoc Networks," in *proc. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

[122] K.H. Hirschelmann, "NMEA 0183," 2002, available from http://www.kh-gps.de/nmea-faq.htm.

[123] D. A. Maltz, J. Broch, and D. B. Johnson, "Experiences Designing and Building a Multi-hop Wireless Ad hoc Testbed," CMU School of Computer Science, Technical Report CMU-CS-99-116, 1999.

[124] E. M. Royer and C. E. Perkins, "An Implementation Study of the AODV Routing Protocol," in *proc. IEEE Wireless Communications and Networking Conference*, 2000.

[125] G. Insolvibile, "Inside the Linux Packet Filter, Part I - Kernel Korner - Issue 94," 2002, available from http://www.linuxjournal.com.

[126] H. Welte, "skb - Linux Network Buffers," 2000, available from ftp://ftp.gnumonks.org/pub/doc/skb-doc.ps.gz.

[127] H. Welte, "The journey of a packet through the linux 2.4 network stack," 2000, available from ftp://ftp.gnumonks.org/pub/doc/packet-journey-2.4.ps.gz.

[128] J. Khoo, "Linux Kernel Programming (Network) - Layer 2 Processing," 2000, available from http://www.aist-nara.ac.jp/~jonath-k/linux/layer2.html.

[129] G. Herrin, "Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stack," Department of Computer Science - University of New Hampshire, Technical Report TR 00-04, May 2000.

[130] J. Dobbelaere, "Linux Kernel Internals - IP Network Layer," Computer Science Department of the College of William & Mary, Presentation Slides, Fall 2001.

[131] G. Insolvibile, "Inside the Linux Packet Filter, Part II - Kernel Korner - Issue 95," 2002, available from http://www.linuxjournal.com.

[132] J. Khoo, "Linux Kernel Programming (Network) - Layer 3 Processing," 2000, available from http://www.aist-nara.ac.jp/~jonath-k/linux/layer3.html.

[133] P. Dwerryhouse, "The Linux IPv4 UDP kernel code," 2001, available from http://leapster.org/linux/kernel/udp/.
[134] C. M. Homan, "Computer Science Notes - Receive," 2000, available from www.cs.rochester.edu/u/choman/cs573/receive.html.
[135] J. Khoo, "Linux Kernel Programming (Network) - Layer 4 Processing," 2000, available from http://www.aist-nara.ac.jp/~jonath-k/linux/layer4.html.
[136] Y. Zhang, "Lecture 17 of Linux Networking (III)," 2002, available from http://www.cs.utexas.edu/users/ygz/378-02S/lecture17.html.
[137] C. M. Homan, "Computer Science Notes - Send," 2000, available from www.cs.rochester.edu/u/choman/cs573/send.html.