The Design and Performance Evaluation of a Proactive Multipath Routing Protocol for Mobile Ad Hoc Networks

Ali Abdalla Etorban

Submitted in fulfilment of the requirements
of the degree of Doctor of Philosophy
at Heriot-Watt University
in the School of Mathematical and Computer Sciences
May 2012

Abstract

Due to unpredictable network topology changes, routing in Mobile Ad Hoc Networks (MANET) is an important and challenging research area. The routing protocol should detect and maintain a good route(s) between source and destination nodes in these dynamic networks. Many routing protocols have been proposed for mobile ad hoc networks, and none can be considered as the best under all conditions.

This thesis presents the design and implementation of a new proactive multipath MANET routing protocol. The protocol, named Multipath Destination Sequenced Distance Vector (MDSDV), is based on the well known single path Destination Sequenced Distance Vector (DSDV). We show that the protocol finds node-disjoint paths, i.e., paths which do not have any nodes in common, except for the source and the destination.

The thesis presents a systematic evaluation of MDSDV in comparison with three well known protocols: one proactive (DSDV), and two reactive (AODV) and (DSR). MDSDV behaves very well in terms of its packet delivery fraction and data dropped in both static and dynamic networks. It delivers nearly 100% of data in dense networks (networks with more than 20 nodes). The speed of the nodes and the number of sources have a low impact on its performance.

Acknowledgements

All thanks to Almighty Allah, most Gracious, most Merciful for giving me the guidance, good health, and the strength to achieve my goal of obtaining my PhD. I owe greatest gratitude to my supervisors Dr. Peter King and Professor Phil Trinder, for their guidance and continuous support during my research. Their devotion to research and serious attitude toward science have given me great encouragement and inspiration to accomplish this research. I will never forget their patience and their valuable comments during the period of my study. I feel very lucky that I finished my doctoral studies under their supervision.

I would like to express my sincere appreciation to the Computer Systems Support Group, especially Iain A. McCrone and Steve Mowbray, for providing me the simulating atmosphere in which the research is performed.

I would like to convey my special thanks to the Ministry of Education in Libya, and the Cultural Affairs in London, for the financial support during my study.

I would like to take this opportunity to express my thanks to all my friends and colleagues. I always remember the time that we have spent together, and wish them all success in their careers and happiness in their special lives.

Very special thanks to my wife Salma and my kids Mosab, Malek, and Mona for their patience during my research. I wish to express my deepest gratitude to my brothers and my sister, Abdalla, Belgasem, and Aisha.

Finally, and most importantly, I wish to express my deepest gratitude to my mother in Libya, also to my father (Allah bless his soul).

Declaration

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

List of Acronyms

ABR	Available Bit Rate
ACK	ACKnowledgement
AM	Available Message
AODV	Ad hoc On- demand Distance Vector routing protocol
AODV-BR	AODV with Backup Routes
AODVM	Ad hoc On-demand Distance Vector Multipath
AOMDV	Ad hoc On-demand Multipath Distance Vector Routing
ARAN	Authenticated Routing for Ad hoc Networks
ARM	Adapting to Route-demand and Mobility
ATM	Asynchronous Transfer Mode
CBR	Constant Bit Rate
CE	Consumed-Energy
CGSR	Clusterhead Gateway Switch Routing
CLR	clear
CMMBCR	Conditional Max-Min Battery Capacity Routing
CMU	Carnegie Mellon University
CoIn	Confidence Interval
CSMA	Carrier Sense Multiple Access

CTS	Clear To Send
DAG	Directional Acyclic Graph
DCF	Distributed Coordination Function
DDR	Distributed Dynamic Routing
DMPSR	Disjoint Multi-Path Source Routing
DREAM	Distance Routing Effect Algorithm for Mobility
Ds	Destination
DSDV	Destination Sequence Distance Vector
DSDV-MC	DSDV for Multiple Channels
DSR	Dynamic Source Routing Protocol
DST	Distributed Spanning Trees based routing protocol
DVR	Distance Vector Routing
EAODV	Energy-aware Ad hoc On-demand Distance Vector
Eff-DSDV	Efficient DSDV
EP	Error Packet
FAMA	Floor Acquisition Multiple Access
FD	Full Dump
Fh	First hop
FP	Failure Packet
FSR	Fisheye State Routing

FT	Full Topology
GloMoSim	Global Mobile Information System Simulator
GPS	Global Positioning System
GSR	The Global State Routing
HARP	Hybrid Ad hoc Routing Protocol
HTF	Hybrid Tree Flooding
HM	Hello Message
HP	Hello Packet
IARP	Intra-zone Routing protocol
IEEE	Institute of Electrical and Electronics Engineers
IERP	Inter-zone Routing protocol
JiST	Java in Simulation Time
LAM	Lightweight Adaptive Multicast algorithm
LDSDV	Light Destination Sequenced Distance Vector routing protocol
Ln	Link number
LSP	link state packets
LSR	Link State Routing
MAC	Media Access Control
MACA	Multiple Access Collision Avoidance
MACAW	Multiple Access Collision Avoidance for Wireless

MANET	Mobile Ad Hoc Networks
MDSDV	Multipath Destination Sequenced Distance Vector routing protocol
MDSDV0	Preliminary Design of MDSDV
MPR	MultiPoint Relay
MRL	Message Retransmission List
MSSN	MPR Selector Sequence Number
M-Zone	Multiple Zones-based routing protocol
NAM	Network Animator
NDMR	Node-Disjoint Multipath Routing
Nh	Number of hops
NPDU	Network Protocol Data Unit
NRL	Normalized Routing Load
NS2	Network Simulator version 2
NRT-VBR	Non Real Time-Variable Bit Rate
NT	Neighbours Table
OLSR	Optimized Link State Routing
OPR	Optimal Path Routing protocol
PCM	Power Control MAC protocol
PCR	Peak Cell Rate

PDA Personal Digital Assistant

PDF Packet Delivery Fraction

PT Partial Topology

QoS Quality of Service

QRY query

QT Queue Table

RDER Route Discovery ERror

RE Residual-Energy

RERR Route Error

RRCM Route Reply Confirmation

RREP Route Reply

RREQ Route Request

RT Routing Table

RT-VBR Real Time-Variable Bit Rate

RTS Request to Send

RWP Random WayPoint

SAODV Secure Ad hoc On Demand Distance Vector routing protocol

SDSDV Secure Destination Sequenced Distance Vector routing protocol

S-DSDV Securing the Destination Sequenced Distance Vector routing protocol

SEAD	Secure Efficient Ad hoc Distance vector routing protocol
Sh	Second hop
SHARP	Sharp Hybrid Adaptive Routing Protocol
SMORT	Scalable Multipath On-demand Routing protocol
SMR	Split Multipath Routing protocol
Sn	Sequence number
SRP	Secure Routing Protocol
StD	Standard Deviation
SWANS	Scalable Wireless Ad hoc Network Simulator
TBRPF	Topology Broadcast based on Reverse Path Forwarding
TC	Topology Control
TORA	Temporally Ordered Routing Algorithm
TTL	Time To Live
UM	Update Message
UP	Update Packet
UPD	update
UBR	Unspecified Bit Rate
VBR	Variable Bit Rate
VoIP	Voice Over IP
WRP	Wireless Routing Protocol

ZHLS Zone-based Hierarchical Link State

ZRP Zone Routing Protocol

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, Mobile Ad Hoc Networks (MANETs) have gained an increasing significance. Ad Hoc networking is needed in many applications such as military and battlefield operations, virtual classrooms or conference rooms, and rescue operations in natural disasters. These kind of applications require a network regardless of any infrastructure, and this is the idea behind MANETs which can be considered as flexible networks and suitable for such applications. MANETs are typically characterized by high mobility and frequent link failures that result in low throughput and high end-to-end delay. The increasing use of MANETs for transferring multimedia applications such as voice, video and data, leads to the need to provide QoS support.

1.1 Problem Statement

MANET routing protocols are based on different design philosophies and proposed to meet certain requirements. Thus, the performance of a mobile ad hoc routing protocol may vary dramatically with the variations of network status. According to how

routing information is gathered and maintained by mobile nodes, routing protocols can be divided into proactive routing, reactive routing and hybrid routing.

Providing a convenient routing protocol for MANETs is a challenge because of its dynamic environment. Therefore, the suitability of each routing protocol depends on many parameters such as, network size, node mobility, and traffic load. All that, together with the limited resources in MANETs (e.g., bandwidth and energy) make the selection of an optimum routing protocol into a complicated task. The frequent topology changes and variable propagation conditions make a routing table obsolete very quickly, which results in enormous control overhead for route discovery and route maintenance. In some scenarios, route maintenance may consume so much in the way of resources that no bandwidth remains for the transmission of data packets. Even worse, the short lifetime of routing information means that a portion of the information may no longer be useful and thus the bandwidth used to distribute the routing update information could be wasted.

When a link failure has occurred, single path proactive routing protocols have to wait until receiving a new routing information to continue sending the data packets. This may lead to low performance and drop huge number of data packets. On the other hand, single path reactive routing protocols must invoke a route discovery process to obtain a route to the destination. This increases the control overhead and delay.

1.2 Motivation

Computer networks become more important due to the rapid growth of the Internet into our daily lives. We can observe the gradual deployment of new multimedia applications such as the world wide web, e-mail, video conferencing, video-on-demand, instant messaging, and Voice Over IP (VoIP). These applications generate traffic with characteristics that differ from traffic generated by data applications, and they are more sensitive to delay and loss [114]. As parts of the Internet become heavily loaded,

congestion may occur which may lead to buffer overflows and packet loss. It may also lead to packet delay as packets take longer to process. Latency may seem acceptable for some applications such as e-mail and file transfer. For real-time applications, data becomes obsolete if it does not arrive in time.

Internet networks have to carry many different types of data. Internet traffic also needs to provide Quality of Service (QoS) for each type of data. Internet traffic can be classified into compressed traffic and uncompressed traffic [80] as shown in Figure 1.1.

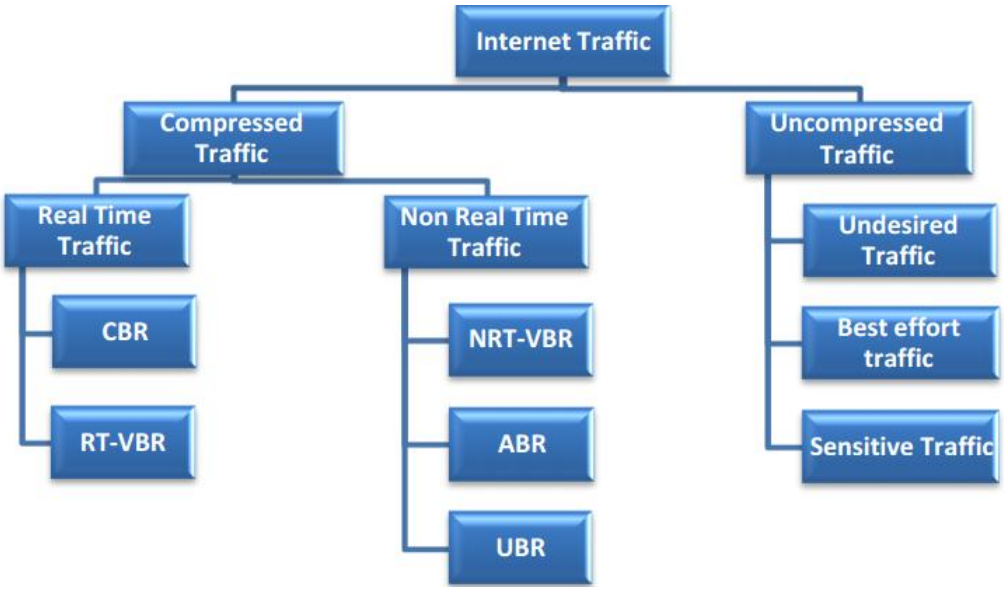

Figure 1.1: The Internet traffic classification [80]

Compressed traffic is proposed to reduce the size of transmission and storage. The traffic is compressed at the source node and travels through the network. Then it is decompressed at the destination node. Four main traffic types are proposed in Asynchronous Transfer Mode (ATM) service categories: Constant Bit Rate (CBR), Variable Bit Rate (VBR), Available Bit Rate (ABR), and Unspecified Bit Rate (UBR). *CBR* is real time traffic that has a constant sending rate. The traffic sending rate is

specified by the Peak Cell Rate (PCR) parameter. CBR is used for connections that need traffic on-time synchronization between the source and destination. *VBR* traffic is a type of loss and delay sensitive traffic. It is known as an ATM service category that is intended for both the real-time and non-real-time applications [101]. The Real Time-VBR (RT-VBR) service category is used for connections that carry traffic at variable rates, which depend on accurate timing between the source and destination. The Non Real Time-VBR (NRT-VBR) service category is used for connections that carry traffic at variable rates, which have no need for on-time synchronization between the source and destination. The *ABR* service category is intended for sources that have the ability to increase their data rate and use the available bandwidth. The ABR service category is designed for any type of traffic that lacks time sensitivity and expects no guarantees of service. The *UBR* service category is intended for best effort traffic and non-critical applications that are very tolerant of delay and packet loss.

Uncompressed traffic is normal traffic that travels through the network in the normal form (without compression). It can be divided into three different categories: sensitive traffic, best- effort traffic, and undesired traffic. *Sensitive traffic* is traffic transmitted over the network if there is online interaction between the source and the destination. It must be delivered with a minimum delay, less packets drops, and it offers a real-time network service. In real time traffic, information about the traffic cannot be obtained in advance, therefore it needs online processing. In *Best-effort traffic*, there is no online interaction. It is easy to extract the entire traffic information before it is transmitted. *Undesired traffic* is created by the delivery of the spam traffic worms, bonnets, and other malicious attacks.

The motivation for choosing a Multipath routing protocol for my research comes from the fact that routing is a challenging issue and a very interesting research area in MANETs. Multipath routing protocols are considered more reliable and robust. Furthermore, whenever a link failure is detected in a primary route, the source node can select the optimal route among multiple available routes. This mechanism enhances

route availability and consequently reduces control overhead, saves energy, enhances data transmission rate, and increases the network bandwidth [68]. For these reasons, multipath routing is useful for many applications in MANETs such as heavy multimedia and real-time traffic, routing fault tolerance, and load balancing.
To the best of our knowledge, most multipath routing protocols are reactive, where the route is established only when a source node needs to send data to the intended receiver. So, we desired to investigate and discover the strengths and weakness of using a proactive multipath routing protocol. Additionally, several routing algorithms have been proposed to improve the performance of DSDV, but none of them uses a multipath technique. DSDV suffers from low performance as it is a single path routing protocol. When a link failure is detected, the node has to wait until fresh routing information is received that may contain route information to the desired destination. This leads to low performance as a result of data packets being dropped.

This research proposes a proactive multipath routing protocol based on DSDV that aims to i) provide efficient fault tolerance in the sense of faster and efficient recovery from route failures in dynamic networks, ii) achieve high QoS in terms of packet delivery ratio and end-to-end delay to support multimedia applications over MANETs, and iii) minimize routing overhead to reduce the energy and bandwidth consumption. In the investigation of this issue, we pay specific attention to the scalability of algorithms in respect of network size, nodal mobility and traffic load.

I have chosen DSDV as a basis for my new routing protocol for the following reasons:

- DSDV is an easy routing protocol to understand and is easy to implement.
- Many routing protocols are based on DSDV such as AODV [93].
- Several routing algorithms have been proposed [3][13][20][52][57][60][70][71][72][118][119], to improve the performance of DSDV, but none of them uses a multipath technique. Section 3.3.2 briefly describes some of these proposed algorithms.

1.3 Thesis Contributions

The thesis makes the following research contributions:

1. We present the design of a novel multipath proactive routing protocol for MANETs. The protocol is novel in proactively maintaining multiple loop-free and node-disjoint paths. We use a simulation-based prototyping methodology. We designed and implemented an initial routing protocol for MANETs called Multipath Destination Sequenced Distance Vector (MDSDV0), extending the well-studied single path Destination Sequenced Distance Vector (DSDV) protocol. MDSDV0 computes multiple loop-free and node-disjoint paths. [Chapter 4].

   Performance measurements show that the control overheads of MDSDV0 are too high as it broadcasts and rebroadcasts routing information (*Update packets* and *Error packets*) to the entire network. A revised protocol, MDSDV, is designed, implemented, and validated to minimize control overheads by broadcasting route information only to its neighbours [Chapter 5].

2. A performance study is presented in chapters 6, 7, and 8. We start by investigating the control overheads of each of the MDSDV control packets [Chapter 6].

   As MDSDV is a proactive routing protocol and based on the well-known DSDV routing protocol, we present a systematic comparative evaluation of MDSDV with DSDV. The results show that the performance of MDSDV is superior to standard DSDV, improving the Packet Delivery Fraction, reducing the Average end-to-end Delay in low mobility environments, providing lower routing load in low mobility, and dramatically decreasing the number of dropped packets [Chapter 7].

   We present a systematic comparative evaluation of MDSDV with two most popular on-demand routing protocols: AODV and DSR. The results show that MDSDV has similar performance to AODV and DSR at light traffic loads, and

outperforms them in heavy traffic situations. The difference increases as the network size increases or mobility decreases [Chapter 8].

3. Implementing a routing protocol is non-trivial and to support further research we provide a complete implementation of MDSDV in the industry-standard NS2 simulator both in Appendix E, and publicly at http://www.macs.hw.ac.uk/∼etorban/.

1.4 Thesis Outline

The rest of the thesis is structured as follows:

- **Chapter 2:** Begins with background information that is required to understand mobile ad hoc networks. It presents brief descriptions of their characteristics and applications. Also, a brief description of three network simulators and three mobility models is given. Finally, this chapter articulates some techniques that are used to establish communications in MANETs.

- **Chapter 3:** describes number of routing protocols and their operations. As our protocol is based on the DSDV routing protocol, an extensive description of DSDV is presented. Moreover, some of the proposed algorithms to improve the performance of DSDV are described in this chapter.

- **Chapter 4:** presents the preliminary design of our new multipath routing protocol. The new protocol is referred to as Multipath Destination Sequenced Distance Vector (MDSDV0) [54]. The protocol guarantees loop freedom and disjointness of alternative paths. MDSDV0 finds node-disjoint paths which do not have any common nodes between a source and a destination.

- **Chapter 5:** Due to the huge number of routing packets that are transmitted by the preliminary version (MDSDV0), this chapter describes the modifications that have been done to reduce the control packets and improve the performance. The final version is referred to as MDSDV.

- **Chapter 6:** Investigates the overheads, i.e., the control packets generated by MDSDV.
- **Chapter 7:** As MDSDV is based on DSDV, this chapter gives a simulation and performance evaluation analysis of MDSDV with DSDV under different environments.
- **Chapter 8:** gives a simulation and performance evaluation analysis of MDSDV with two well known reactive routing protocols: AODV and DSR.
- **Chapter 9:** concludes the thesis with a summary of the major achievements of the thesis, presents the limitations of MDSDV, and outlines future works.

1.5 Publications

1. A DSDV-based Multipath Routing Protocol for Mobile Ad-Hoc Networks. In Proceedings of the 8th Annual PostGraduate Symposium on The Convergence of Telecommunications, Networking and Broadcasting, United Kingdom, 2007 [54]
2. A Performance Comparison of MDSDV with AODV and DSDV Routing Protocols. In Proceedings of the 25th UKPEW2009, Performance Engineering Workshop, United Kingdom, 2007 [32]

Chapter 2

Mobile Ad Hoc Networks

This chapter summarises Mobile Ad Hoc Networks (MANETs). We discuss the motivation for MANETs in section 2.1, and outline their characteristics in section 2.2. Some of the typical applications used in MANETs are presented in 2.3. A brief description of three simulators (NS2, GloMoSim, and JIST/SWANS) is given in section 2.4. We discuss three mobility models in section 2.5. Finally, section 2.6 describes some techniques that are used to establish communications in MANETs.

2.1 Introduction

Recently, network researchers are studying networks based on new communication techniques, especially wireless communication. Mobile networks have been of significant interest in the past ten years because of their improved flexibility and reduced costs. Compared to wired networks, mobile networks have unique characteristics and differ in the way of communication. Wired networks transfer data packets through physical cables; whereas, in mobile networks, the communication between different devices can be either wireless or wired.

In mobile networks, node mobility makes the network topology change frequently, which is rare in wired networks. Mobile networks have a high error rate, bandwidth

limitations and power restrictions. Due to the impacts from transmission power, receiver sensitivity, noise, fading and interference, wireless link capacity continually varies. Wireless networks can be deployed quickly and easily, and users stay connected to the network while they are moving around. Also, they play an important role in both civilian and military fields. We have seen great developments in Wireless networks infrastructure, availability of wireless applications, and proliferation of Wireless devices everywhere such as laptops, PDAs, and cell phones.

According to the deployment of network infrastructure, Wireless networks can be divided into two types [104]. The first type are Infrastructure-based wireless networks and the second are infrastructure-less mobile networks, commonly known as ad-hoc networks. Infrastructure networks are those networks with fixed and wired gateways. The bridges for this type of networks are known as base stations. A mobile node connects to the nearest base station which is within its communication radius. As the mobile travels out of range of one base station and into the range of another, a "handoff" occurs from the old base station to the new, and the mobile is able to continue communication seamlessly throughout the network

A mobile ad-hoc network (MANET) is a group of wireless mobile nodes dynamically establishing a short live network without any use of network infrastructure or centralized administration. In addition to the high degree of mobility, MANET nodes are distinguished by their limited resources such as power, bandwidth, processing, and memory. If two mobile nodes need to communicate with each other, they can communicate directly if they are within the transmission range of each other, otherwise intermediate nodes (nodes in between) should forward the packet from one of them to the other. Thus, each node in the network acts both as a host and router and must therefore be willing to forward packets to other nodes. All nodes in mobile ad hoc networks are free to move, and the link between two nodes is broken when one of them moves out of other's transmission range, and hence the network topology may change frequently.

2.2 Characteristics

Compared to other wired or infrastructure-based wireless networks and according to [25], Mobile ad hoc networks have the following characteristics.

- *Dynamic topology* All nodes of mobile ad hoc network are free to move causing network topology changes rapidly at unpredictable times. Links between nodes are expected to break much more frequently than with wired and infrastructure-based wireless networks.
- *Self-organization:* Due to the lack of infrastructure or central administration, nodes should be able to form themselves into a network.
- *Multi-hopping:* In a mobile ad hoc network, nodes use a wireless channel to transmit data, and due to the limited number of a node's neighbours, intermediate nodes are used to relay the packets.
- *Resource conservation:* In mobile ad hoc networks, the nodes are limited in both energy supply and processing power. Power conservation becomes a very important factor to be considered when designing a network. Therefore, optimizing all operations may minimize the energy consumption.
- *Limited security:* Mobile ad hoc networks are more prone to security threats than wired networks or infrastructure-based wireless networks because of their unique characteristics. Each mobile node in an ad hoc network can function as a router or packet forwarder for other nodes, both legitimate users and malicious attackers can access the wireless channel, and there is no well place where access control mechanisms can be deployed. As a result, separating the inside of the network from the outside world becomes imprecise.
- *Scalability:* In some applications (e.g., battlefield deployments), mobile ad hoc networks may grow up to several thousand nodes. Mobile ad hoc networks

suffer from scalability problems in channel capacity, because channel capacities are very limited and maximum use of channel capacity can be reached faster. Due to the multihoping nature of mobile ad hoc networks, their scalability is related to the routing protocols they employ.

2.3 Applications

Mobile ad-hoc networks are used in many applications, ranging from small, static networks, to large, highly dynamic networks such as virtual classrooms, conferencing, emergency services, military applications.

- *Military applications:* Mobile ad hoc networks satisfy several military needs such as battlefield survivability. In such environments, setting up of an infrastructure for communication between soldiers in battlefield could be impossible. The wireless devices carried by soldiers can form a mobile ad hoc network to support communication among them.

- *Conferencing:* perhaps the prototypical application requiring the establishment of a mobile ad hoc network is mobile conferencing. One common use is to create a temporary network to support a meeting in a conference room.

- *Disaster relief operations:* Each year natural disasters (e.g., earthquake, flood), destroy people's lives around the world. As the importance of the Internet grows, the loss of network connectivity during such disasters will be a more noticeable effect of the misfortune. So, it is important to find ways to enable the operations of networks even when infrastructure elements are disabled as a result of the disaster.

- *Personal area networks:* The idea of a Personal Area Network (PAN) is to create a network that consists of nodes which are associated with a single person. These nodes may be placed in a person's clothes, belt or carried in handbags.

2.4 Simulators:

Numerous simulators have been developed and used in simulation studies. Some of these simulators are open source such as Network Simulator2 (NS2) [33], GloMoSim [6], and Jist/SWANS [7], and some of them are commercial such as QualNet [1]. NS2 and GloMoSim are the most popular simulators for simulation studies [64]. Brief descriptions of three simulators are given in the next subsections. The performance comparisons among simulation results have to be based on the same simulator. In our thesis, all simulations and performance evaluations are based on use of the NS2 simulator. We used NS2 as it is widely used in the networking research community and accepted as a tool to experiment with new protocols and distributed algorithms [29].

2.4.1 Network Simulator (NS2):

Network simulator version 2 (NS2) [33] is an object-oriented, discrete event driven network simulator. It has been developed at the University of California in Berkeley. The core of NS2 is written in C++ and the configuration depends on OTCL scripts. NS2 is a very widely used simulator for the simulation of a variety of routing protocols, several QoS (Quality of Service) mechanisms, and more. It is open source and several extensions have been provided. Dricot and Doncker in [31] proposed a highly accurate physical model based on ray tracing Markov chains which can be very useful for the simulation of mobile ad hoc networks.

A project at Carnegie Mellon University (CMU) [46] provided wireless extensions, which include mobile nodes and wireless communication. The Random WayPoint (RWP) model has been implemented in this extension. It can be used to create a topology scenario based on some parameters, such as the number of nodes, pause time between direction changes, and the speed of nodes. The extension also provides

implementations of some routing protocols, such as DSDV [94], AODV [93], DSR [47], and TORA [90].

NS2 does not provide any statistics. Instead, every event produced by the simulation is written to a trace file that can be processed to extract the desired information. The generated trace file is large even with a low scale simulation scenario. A simple C program or an AWK script can be used to extract and analyze the data stored in the trace file. One of the drawbacks of NS2 is its complexity. The mix of C++ and OTCL increases the complexity of the implementation. Modifying an existing component or adding new ones requires a developer to write or rewrite several software modules. Therefore, NS2 is not easy to use in simulation experiments, and it is not easy to learn how to add new components or modify existing ones because its documentation is not very easy to understand.

The Network Animator (NAM), is a Tcl/TK based animation tool which can be used to view NS-2 trace files for post-processing, analysis and replay of simulations

2.4.2 GloMoSim

Global Mobile Information System Simulator (GloMoSim) [6] is a scalable simulation library, developed at the University of California, to support studies of large-scale network models with thousands of nodes. GloMoSim is the second most used simulator after NS2, in the research community [29]. It has a layered model of network communication according to ISO/OSI model. GloMoSim is written in C with Parsec extensions [5]. Thus, new modules for GloMoSim need to be written in Parsec as well. GloMoSim code is free, but Parsec is not.

In GloMoSim, network layers are represented as objects called entities and events are represented as time-stamped messages handled by entities. The GloMoSim's network model does not define every network node as an entity, because this leads to too

many objects. Instead, GloMoSim uses entities to model network layers, and makes messages cross the layer stack by being interchanged by the entities.

2.4.3 JIST/SWANS

Java in Simulation Time (JiST) [7] is a high-performance discrete event simulator that runs over a standard Java virtual machine. It was developed by Rimon Barr et al. at Cornell University. Its main idea is to transform the Java virtual machine into a scheduler for events. The JiST system is composed of four components: a compiler, a byte code rewriter, a simulation kernel and a virtual machine. Figure 2.1 presents the architecture of JiST/SWANS where a simulation is first compiled, then dynamically rewritten as it is loaded, and finally executed by the virtual machine with support from the language-based simulation time kernel.

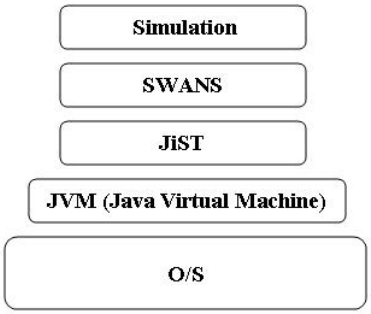

Figure 2.1: JiST/SWANS architecture

The Scalable Wireless Ad hoc Network Simulator (SWANS) [8] is a collection of components to simulate ad hoc networks based on the JiST simulation engine. It provides all the mechanisms needed to simulate MANETs. Figure 2.2 shows the SWANS architecture where every SWANS component is encapsulated as a JiST entity (i.e., it stores its own state and interacts with other components via interfaces). Nodes consist of several entities that are linked together and represent the different stack elements of a network application. SWANS is able to simulate large networks. According to the

survey in [64], 2 out of 29 (7%) simulation experiments have used the JIST/SWANS simulator.

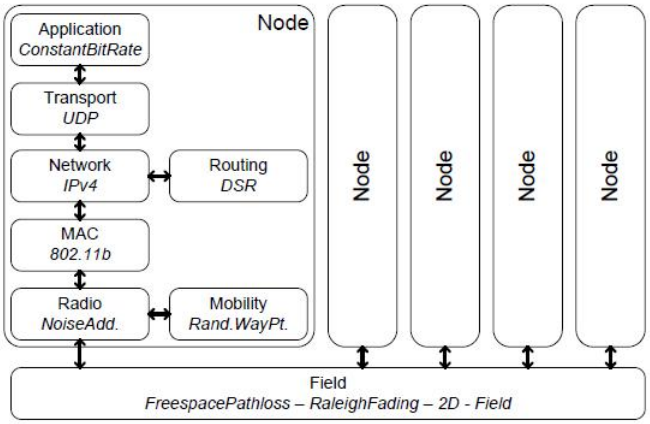

Figure 2.2: SWANS architecture [50]

2.5 Mobility Models:

There are several mobility models that can be used in the simulation of ad hoc networks [17]. Mobile Nodes (MNs) within an ad hoc network are free to move from one location to an other. However, finding a way to model these movements is not obvious. In order to simulate a new routing protocol for an ad hoc network, it is necessary to use a mobility model that precisely represents the Mobile Nodes (MNs) that will utilize the protocol. Currently, two types of mobility models are used in simulations of networks: traces and synthetic models [106]. Traces are the mobility patterns which are observed in real life systems. Traces provide accurate information, especially when a large number of participants are involved. But, privacy may prevent the collection and distribution of such information. Moreover, it is not easy to model ad hoc networks if traces are not created. In this case, it is necessary to use synthetic models. Synthetic models attempt to realistically represent the behaviours of nodes without using traces. Selecting an appropriate mobility model may not be

a simple task. A proper mobility model should mimic the movements of real MNs. MNs should not travel in a straight lines at constant speeds throughout the entire simulation because real MNs do not travel in such a manner. The next three subsections discuss three popular synthetic mobility models: The Random Walk Mobility Model, the Random Waypoint Mobility Model, and the Random Direction Mobility Model.

2.5.1 Random Walk Mobility Model

Since many entities move in unpredictable ways, the Random Walk Mobility Model was developed to mimic this erratic movement [17]. In this kind of mobility model, a mobile node randomly chooses a direction and speed to move from its current location to a new location. The speed and direction are chosen from pre-defined ranges, [minimum speed, maximum speed] and [0,2π] respectively. If a mobile node reaches a simulation boundary, it bounces off the simulation border with an angle determined by the incoming direction. The node then continues along this new path. Several varieties of the model have been developed such as the 1-D, 2-D, 3-D, and n-D walks. Because the Earth's surface is usually modelled using a 2-D representation, the 2-D Random Walk Mobility Model is of special interest. The Random Walk Mobility Model is widely used [17], and it is a memoryless mobility pattern because it does not have any knowledge concerning its past locations and speed values [65][66]. The current direction and speed of the node are independent of its past direction and speed [39]. This model may generate unrealistic movements such as sudden stops and sharp turns.

2.5.2 Random Waypoint Mobility Model

The Random Waypoint Mobility Model is the most widely used mobility model. Many researchers use it to compare the performance of various mobile ad hoc network routing protocols [127]. In [64], the author stated that 19 out of 32 (60%)

simulation experiments used the random waypoint model. This model includes pause times between changes in direction and/or speed. Using the waypoint mobility model, each node starts the simulation by remaining stationary for pause-time seconds. Then, it randomly chooses a destination in the simulation area and moves towards that destination at a speed uniformly chosen between 0 and maximum speed. When the node reaches the selected destination, it halts again for pause-time, selects another destination and starts to move towards the new destination. This process is repeated for the duration of the simulation. In [127], it has been shown that the average speed of a mobile node decays with time. This is because of the fact that low speed nodes spend more time to reach their destinations than high speed nodes. It is also shown that increasing the speed of nodes results in increased network connectivity. Another reason for the popularity of the Random Waypoint mobility model is that NS2 and GloMoSim have it built in.

2.5.3 Random Direction Mobility Model

The Random Direction Mobility Model [103] was developed to overcome the clustering of nodes in one part of the simulation area, produced by the Random Waypoint Mobility Model. The clustering occurs near the centre of the simulation area. In the Random Waypoint Mobility Model, the probability of choosing a new destination that is located in the centre of the simulation area, or a destination that requires travel through the centre of the simulation area is high. In the Random Direction Mobility model, instead of selecting a destination, nodes select a direction in which to travel (direction is measured in degrees). Each node starts the simulation by selecting a degree between 0 and 359, and finds a destination on the boundary in this direction of travel. Then, it selects a speed and travels to the selected destination at the selected speed. Upon reaching the destination, the node pauses for some pre-defined pause time, and selects a new direction between 0 and 180 degrees (Direction is limited because the node is on the boundary). Next, the node identifies the destination on

the boundary in this line of direction, selects a new speed, and resumes travel. In the modified version of the Random Direction Mobility Model [103], nodes continue to choose random directions but they are not forced to travel to the boundary before stopping to change the direction. Instead, the node chooses a random direction and selects a new destination anywhere along the selected direction. Then, it pauses for a certain time before choosing a new random direction.

2.6 Issues in Mobile Ad Hoc Networks

In the following subsections, we describe some techniques that are used to establish communications in MANETs. In particular, we concentrate on three areas: Medium Access Control (MAC), Energy Conservation, and Security issues. Another important issue is routing which is described in detail in chapter 3.

2.6.1 MAC-Layer Protocols for Ad Hoc Networks:

The applicability of the existing MAC-layer protocol to the radio environment is limited by two interference mechanisms: the hidden terminal and the exposed terminal problems.

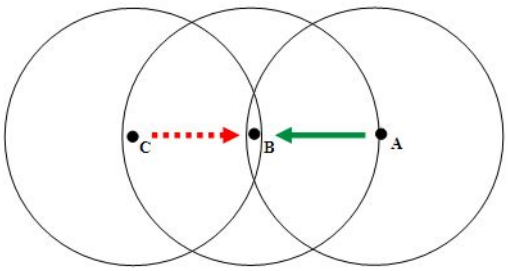

Figure 2.3: An example of the hidden terminal problem

The hidden terminal problem occurs because the radio has limited range. Thus, two nodes which maintain connectivity to a third node may not hear each other. Consider the situation in Figure 2.3. Node A and node C are in the transmission range of node B, but there are not in the transmission range of each other. Node A is currently transmitting data to node B. Node C wishes to communicate with node B as well. Following the Carrier Sense Multiple Access (CSMA) protocol, node C listens to the medium, but since node A is too far from node C, node C does not detect node A's transmissions, and decides the medium is free. Consequently, node C transmits to the medium, causing collisions at node B with A's transmissions to B.

The exposed terminal problem is illustrated in Figure 2.4. Node A is transmitting data to node B, while node C needs to transmit data to node D. Following the CSMA protocol, node C listens to the medium and hears that node A is using the medium. This causes node C to delay in transmitting to the medium. However, there is no reason for preventing node C from transmitting concurrently with the transmission of node A, as the transmission of node C would not interfere with the reception at node B due to the distance between the two. The reason here is, the fact that the collisions occur at the receiver node, while the CSMA protocol checks the medium's status at the transmitter.

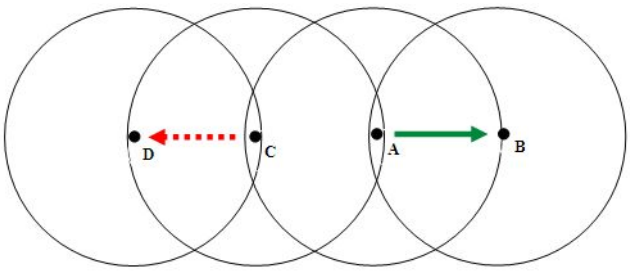

Figure 2.4: An example of the exposed terminal problem

The hidden terminal problem reduces the capacity of a network because of increases

in the number of collisions, while the exposed terminal problem reduces the network capacity because nodes unnecessarily delay in transmitting. IEEE 802.11 uses a four-way RTS/CTS/Data/Ack exchange to reduce collisions caused by hidden terminals. Before sending a data packet, a node sends an RTS (Request to Send) packet to the destination. The destination responds with a CTS (Clear to Send), if the network is idle. The sender then starts to transmit the data packet, and waits for an ACK (ACKnowledgement) from the receiver. The node considers the medium to be busy for some time, if it overhears an RTS or CTS [61]. In this case, the node defers its transmission to prevent a collision. The RTS and CTS packets include the time that the medium will be busy for the remainder of the exchange.

Several MAC schemes have been developed for wireless ad hoc networks to improve the performance of MAC-layer protocols such as Multiple Access Collision Avoidance (MACA) [51], Media Access Protocol for Wireless LANs (MACAW) [10], and Floor Acquisition Multiple Access (FAMA) [34].

2.6.2 Energy Conservation

Mobile devices in mobile ad hoc networks rely on batteries for energy. Limitation in battery power and the additional energy needed to support network operations (e.g., routing) makes energy conservation one of the main concerns in ad hoc networks.

Power control approaches in mobile ad hoc networks can be classified into two categories: power controlled topology management and power aware routing. Power controlled topology management schemes find the lowest transmission power level for each link. On the other hand, power aware routing schemes find routes that consist of links consuming the least energy.

Different routing protocols dealing with energy issues have been proposed for mobile ad hoc networks [30][48][74][84][109][112][113][115][130]. Some of these proto-

cols use conventional metrics such as minimum hop and delay, whereas others use new metrics such as load balancing, stability, and power consumption [109][113][115].

Jung et al. have proposed a Power Control MAC protocol (PCM) [48], that periodically increases the transmit power during DATA transmission. A power-aware source routing protocol for MANETs is presented in [74], to increase the network lifetime. A policy is applied to fetch paths from the cache, to make sure that no path is overused and each selected path has minimum battery cost among all paths between two nodes. Singh et al. have presented power-aware cost metrics in [109], to determine routes in wireless ad hoc networks. They used five metrics based on battery power consumption at nodes. They showed that using these metrics in a shortest-cost routing algorithm reduces the cost/packet of routing packets by between 5% and 30% over shortest-hop routing. Moreover, using these metrics, the mean time to node failure is significantly increased. Toh have presented a new power-aware routing protocol called Conditional Max-Min Battery Capacity Routing (CMMBCR) [115], to maximize the lifetime of ad hoc networks by achieving two objectives: the power consumption rate of each node is evenly distributed, and minimizing the transmission power for each connection request. The proposed scheme chooses the shortest path if nodes in all routes have sufficient battery capacity. If the battery capacity of some nodes goes below a predefined threshold (γ), routes using these nodes will be avoided. Thus, the time until the first node power-down is extended. The value of γ can be adjusted to maximize either the time that the node takes to power down or the life time of most of the nodes in the network.

Zhaoxiao et al. proposed a new mechanism of energy-aware for Ad Hoc networks, named Energy-aware Ad hoc On-demand Distance Vector (EAODV) [130]. EAODV is based on the classical AODV routing protocol. Combined with the value of the remaining battery capacity, the link dynamic priority-weight metric is used to establish whether a node can be a part of an active route. In EAODV, the route which spends less energy and owns larger capacity is selected by synthetic analysis. When a node receives the first copy of RREQ, it records the residual battery energy and

the consumed energy into the *Residual-Energy* (RE) and the *Consumed-Energy* (CE) fields respectively. Next, the node uses the formula $\beta_i = \left(\frac{RE_i}{CE_i}\right)^2$ to compute the dynamic route priority-weight value between itself and the source, and to supersede the Route_β value of the RREQ. Finally, the node establishes a route to the source and updates the routing table by inserting a new entry as a route to the source. When the node receives another copy of the RREQ, it compares the Route_β value of RREQ with the Route_β value in the routing table. The RREQ is forwarded if the Route_β value in RREQ is higher, or equal to the Route_β value of routing table with less hop, otherwise the RREQ is discarded.

2.6.3 Security Issues

Nodes in MANETs are free to join and leave the network at any time without any notice. Thus, it may be difficult to have a clear picture of ad hoc network membership. Consequently, no trust relationships among nodes can be assumed [89]. In such an environment, a path between two nodes can not be guaranteed to be free of malicious nodes, which might try to harm network operations.

Nodes in MANETs exchange routing information in order to establish routes between them. Such information may become a target for malicious adversaries. The threats to routing protocols may come from attackers that provide erroneous routing information, replaying old routing information, or distorting routing information [131]. As a result, the attacker could partition the network or introduce excessive traffic load into the network.

Cryptography and Misbehaviour Detection are two schemes that have been used for security, and a brief description of each of them is given in the following paragraphs:

Symmetric vs. Asymmetric Cryptography: If all routing messages in MANET are encrypted with a symmetric cryptosystem, every participant node has to know the key. This is not a problem if the participants are a team that meet to share the team-key and create the network. In this case team members trust and authorize each

other to change their routing tables. But suppose that we are in a meeting room or a campus, and we need to create a MANET where everyone can participate. In this case everyone does not trust the others. With this scenario, the best option is to use an asymmetric cryptosystem (with private and public key pairs).

Misbehaviour Detection Schemes: some work uses misbehaviour detection schemes [78] to secure ad hoc networks, but this kind of approach has two problems: sometimes it is not feasible to detect several kinds of misbehaviour (especially to distinguish misbehaviour from transmission failures), and it is too hard to guarantee the integrity of the routing messages. So, using this approach, any malicious node can generate false misbehaviour reports.

There are many published work on security issues in ad hoc networks [4][40][41][62] [67][79][88][89][105][107][110][119][128][129]. An On-Demand Secure Routing Protocol is proposed in [4] to provide resilience to byzantine failures caused by individual or colluding nodes. The technique used in this protocol detects a malicious link after log n faults have occurred (n is the length of the path). These links are avoided by using a route discovery protocol to find a least weight path to the destination. In the Secure Efficient Ad hoc Distance vector routing protocol (SEAD) [40], hash chains are used in combination with DSDV-SQ [15] (to authenticate hop counts and sequence numbers). At every time, each node has its own hash chain which is divided into segments, and elements in a segment are used to secure hop counts. The protocol determines the size of the hash chain when it is generated. Upon using all the elements of the hash chain, a new one should be computed. Ariadne is a protocol presented in [41] to provide security against one compromised node and active attackers, and relies only on efficient symmetric cryptographic operations. Papadimitratos et al. proposed the Secure Routing Protocol (SRP) [89], which can be applied to several routing protocols. Using SRP, it is required that the source and destination must have a security association between them for every route request. Sanzgiri et al. proposed the Authenticated Routing for Ad hoc Networks (ARAN) [107]. ARAN uses authentication and requires the use of cryptographic certificates to offer routing security. In

ARAN, every node forwarding a route request or route reply messages should sign it (this consumes power and increases the size of the routing message). The Secure Ad hoc On Demand Distance Vector (SAODV) [128] is an extension of AODV and can be used to protect the route discovery mechanism by providing some security features. Two mechanisms are used to protect AODV messages: *digital signature* to authenticate the non-mutable fields of the message, and *hash chains* to secure the hop count information. SAODV uses a different manner to protect the error message because they have a big amount of mutable information.

2.7 Research Methodology

The research methodology used is simulation-based prototyping. That is, we designed and implemented an initial routing protocol MDSDV0 that extends the well-studied DSDV protocol. The new protocol is validated, and the performance is measured using the NS2 industry standard discrete event network simulator. We revise the protocol based on these performance measurements to produce the final protocol, MDSDV. We again use simulation to validate MDSDV, and to make performance comparisons with NS2 implementations of popular routing protocols.

Simulation in general and the NS2 simulator in particular are widely used to evaluate network protocols. They have significant advantages over other methodologies such as direct experiments and mathematical modelling. A *computer simulation* is an application designed to mimic a real-life situation. Compared to other approaches such as mathematical models or physical experiments, simulation models have several advantages.

One of the advantages of simulators is that they are able to provide users with practical feedback when designing real world systems. Consequently, the designer can determine the correctness and efficiency of a design before the system is actually

constructed. Simulators permit system designers to study a problem at several different levels of abstraction. By approaching a system at a high level of abstraction, the designer can understand the behaviour and interactions of all components of the system, and is therefore better equipped to counter the system's complexity.
A simulation and modelling process, using computer software, is a low-cost alternative when compared to a real implementation without a prior simulation study. Particularly taking into account all the tests that may have to be carried out repeatedly using different parts to reach the objective. Using simulators, it is possible to compare alternative designs and select the optimal system.

Despite the advantages of simulation presented above, simulators do have their drawbacks. One of the disadvantages of using a network simulator for testing a distributed application stem from the fact that there is no real network involved in the simulation. Thus, it is difficult to accurately model cross traffic, resource contention, or failures that may occur. As a result, the ability of the developer is limited to test the application under realistic network conditions. Another disadvantage of simulators is that they do not run real application code. Hence, the application must be rewritten for the target simulation platform, which runs the risk of introducing problems that do not exist in the real code or masking problems that do exist.
In general there is no simulator that gives 100% guaranteed and perfect results, although there are differences between the simulators, they do try to provide as accurate as possible results.

Real experiments are difficult to conduct and expensive in terms of hardware and development time. They are inflexible when trying to change parameters, and sharing them with other researchers is difficult. The results may only apply to one situation. Moreover, it is difficult to replicate real experiments in some cases.

A *mathematical model* is a description of a system using mathematical concepts and language. It usually describes a system by a set of variables and a set of equations that establish relationships between the variables. In order to be tractable, mathematical

models require assumptions and restrictions to be placed on the model and this can lead to inaccuracy. Generally speaking network protocols are too complex to model mathematically. Moreover any such model must be carefully evaluated to determine it's suitability both in terms of fit to empirical data and applicability to the problem domain, and as discussed above, obtaining empirical data for a real implementation of a novel networking protocol is extremely expensive.

The advantages of simulation over mathematical models and real experiments mean that novel network protocols are almost invariably evaluated by simulation. Indeed industry standard tools like NS2 [33] and GloMoSim [6] have emerged to meet this need. This thesis follows this practice.

Chapter 3

MANET Routing Protocols

Routing is a fundamental problem in mobile ad hoc networks. This chapter presents an overview of several routing protocols that have been proposed by researchers. We present a number of ways to classify the routing protocols in section 3.2. A description of some single path proactive, reactive, and hybrid routing protocols is presented in sections 3.3, 3.4, and 3.5 respectively. As our protocol is a multipath one, section 3.6 presents a brief description of some existing multipath routing protocols.

3.1 Introduction

Routing is an important and challenging issue in dynamic multi-hop networks. Thus, many routing protocols algorithms have been proposed in recent years. A routing protocol is used to discover routes between nodes allowing communication within the network. The main goal of such a routing protocol is to establish a correct and efficient route between a pair of nodes, so that messages can reach their destination in a timely manner. During the last two decades, many mobile ad hoc network routing protocols have been proposed because of their importance in dynamic networks [69]. It is not possible to consider a particular algorithm or class as the best for all

scenarios. Each protocol has its own advantages and disadvantages and may only be suited for certain situations [104]. Due to a variety of challenges, designing a mobile ad hoc network routing protocol is a tough task. Firstly, in mobile ad hoc networks, the topology changes frequently because of node mobility. Secondly packet losses may occur frequently because of the variable and unpredictable capacity of wireless links. Furthermore, the broadcast nature of the wireless medium introduces the hidden terminal and exposed terminal problems that are mentioned in subsection 2.6.1. Finally, mobile nodes have limited power, limited bandwidth resources and require effective routing schemes.

3.2 Classification of Routing Protocols

There are different criteria to classify routing protocols for wireless ad hoc networks [132]. We present some of these criteria in the following subsections:

3.2.1 Link State Routing (LSR) vs. Distance Vector Routing (DVR)

Link State Routing protocols (LSR) take into consideration link variables, such as bandwidth, delay, reliability and load. In LSR [111], nodes exchange routing information in the form of link state packets (LSP), which include link information for all their neighbours. As links change state, LSPs are flooded immediately into the entire network. By receiving LSPs, every node is able to construct and maintain a global network topology. On the other hand, Distance Vector Routing protocols (DVR) determine the best path in terms of how far the destination is. In DVR, every node maintains a distance vector that may include destination ID, next hop, and distance for each destination. Each node exchanges distance vectors with its current neighbours. Upon receiving such information, a node constructs new routes and updates its distance vector.

3.2.2 Proactive, Reactive and Hybrid routing

Depending on how routes are maintained, routing protocols can be classified as proactive, reactive or hybrid as shown in figure 3.1. The proactive routing protocols constantly propagate and maintain routing information. As a result, a route to every other accessible node in the network is always available, regardless of whether it is needed or not. These protocols respond to topology changes by propagating updates throughout the entire network. The reactive routing protocols create routes only when needed by the source node. When a source node needs to send data to a destination node and does not have a valid route to that destination, it initiates a *Route Discovery process* to establish such a route. Once a route has been established, it is maintained by some form of maintenance procedure. The hybrid routing protocols combine both proactive and reactive routing strategies to benefit from advantages of both types and overcome their shortcomings.

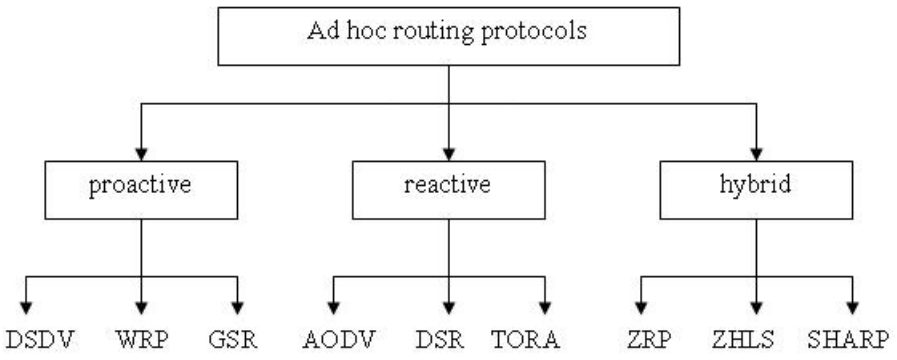

Figure 3.1: Categorization of Ad hoc Routing Protocols

Sections 3.3, 3.4, and 3.5 present an overview of a number of previously proposed protocols for routing in ad hoc networks. It includes both protocols that are exclusively proactive or reactive in their nature, as well as those that are a hybrid of the two types.

3.2.3 Flat Structure vs. Hierarchical Structure

In a flat structure, all nodes participating in the network are at the same level and perform the same routing functions. It is simple and efficient for small networks. But, in large networks the amount of routing information is large and takes a long time to arrive at remote nodes. In large networks, hierarchical (cluster-based) routing can be used to address these issues. In a hierarchical structure, nodes are dynamically organized into clusters (partitions). Then, the clusters are aggregated into superclusters (larger partitions) and so on. A node may be expected to have complete topology information about its cluster hence proactive routing can be used. If the destination is in different cluster, intercluster routing must be used which is generally reactive, or a combination of both, such as with the Zone-based Hierarchical Link State (ZHLS) routing protocol [44].

3.2.4 Source Routing vs. Hop-by-Hop Routing

Some of the routing protocols include the entire route in the headers of data packets (e.g., DSR [47]). Thus, intermediate nodes forward these packets according to the route included in the header. This mechanism is called "source routing". Its advantage is that intermediate nodes forward the packet without maintaining routing information, since the packets contain all the routing decisions. On the other hand, in hop-by-hop routing, the route to a destination is calculated step by step at each intermediate node.

3.2.5 Single Path vs. Multiple Paths

Some routing protocols maintain only a single path for each destination. This results in a simple protocol and saves storage. Single Path routing protocols are incapable of load balancing traffic. Other routing protocols maintain multiple routes for each

destination. Multipath routing protocols have the advantages of easier recovery from link failure and being more reliable and robust. Furthermore, the source is able to select the best route among multiple available routes. Section 3.6 discusses some of the existing multipath routing protocols.

3.3 Proactive Routing Protocols

A proactive routing protocol is also called a "table driven" routing protocol. Using one of the proactive routing protocols, nodes in a mobile ad hoc network continuously evaluate routes to all reachable nodes and modify routing information. Thus, a source node can get a routing path immediately as soon as it needs one. In proactive routing protocols, each node maintains routing information to every node in the network. The routing information is stored in a number of tables. These tables are periodically updated and updated if there is a significant change in the network topology. The difference between existing proactive routing protocols lies in the way that the routing information is updated, and the type of information stored in each routing table. Moreover, each routing protocol may maintain a different number of tables. Several proactive routing protocols have been proposed, such as Destination Sequence Distance Vector (DSDV) [94], the Wireless Routing Protocol (WRP) [83], and the Fisheye State Routing (FSR) [91] [92]. The following subsections present a brief description of some proactive routing protocols.

3.3.1 Destination Sequenced Distance Vector (DSDV)

DSDV [94][38] is a proactive routing protocol which maintains routes regardless of their usage. It is based on the Bellman-Ford routing algorithm, which can become unacceptable in mobile ad hoc networks because of its long convergence time. Numerous extensions or modifications to DSDV have been proposed to improve its performance such as [3][13][52][60][70][71][72][118]. DSDV is a distance vector routing

protocol and it solves the major problem associated with the Distance Vector routing of wired networks (i.e., Count-to-infinity), by using destination sequence numbers. Also, at all times, the DSDV protocol guarantees loop-free paths to each destination.

Using DSDV, each mobile node maintains a routing table, that lists one route for each destination. Each routing table entry consists of the destination node, the first hop towards the destination, the metric (number of hops to reach the destination), and the sequence number which is originally generated by the destination node. Sequence numbers are used to distinguish the new routes from the stale routes. The routing table is used to transmit packets between the nodes of the network.

In DSDV, each mobile node advertises its routing table (e.g., by broadcasting its entries) to its current neighbours. The entries in the routing table may change dynamically over time, so the routing information should be advertised to ensure that every node can always locate every other mobile node. Additionally, each mobile node agrees to relay data packets to other nodes upon request. Before each advertisement of a new routing table, mobile node increases its sequence number by 2.

DSDV takes care of topology changes by using a certain procedure which is based on two kinds of updating: *time-driven updates*, which are periodic transmissions of a node's routing table, and *event-driven updates* which react to link failures. Nodes schedule the newly recorded routes for immediate advertisement to the current node's neighbours. Routes with an improved metric are scheduled for advertisement at a time which depends on the average settling time for routes to the particular destination under consideration.

To reduce the amount of information in the routing information packets, DSDV uses two different types of update packets: a "full dump" or an incremental update. A full dump is the sending of all the routing table entries to the current neighbours and could span many packets. In contrast, in an incremental update, the node only sends those entries that are changed since the last full dump. The incremental update must fit in only one Network Protocol Data Unit (NPDU). The node implements some

means to determine which routes have significant changes, and includes them in each incremental advertisement. For example, if a stabilized route shows a better metric for some destination, it is likely to be considered as a significant change. But if a route with a new sequence number and the same metric is received, it is unlikely to be considered as a significant change.

As mobile nodes are free to move from place to place, this movement may cause a link breakage. The broken link may be detected by the MAC layer or by not receiving broadcasts for a certain time from a former neighbour. When a link to a neighbour is broken, the route to that neighbour and any route through that neighbour is immediately assigned a ∞ metric. Because this change qualifies as a substantial route change, such updated routes are immediately broadcast as new routing information.

When a node receives new routing information, it compares this information with the information that is already available in its routing table. Any route with a greater sequence number is used. Routes with a lower sequence number are ignored. A route with the same sequence number as an existing one is chosen if it has a better metric, and the existing one is discarded. The metrics of the chosen routes from the received information are each incremented by one hop. The new stored routes are scheduled for immediate advertisement to the current neighbours.

DSDV uses the settling time table to prevent fluctuations of routing table entry advertisements. The settling time table fields are *Destination address*, *Last settling time*, and *Average settling time*. A node consults its settling time table to decide how long to wait before advertising the new route. When new routing information is received by a node, and applying the updates to the table, processing occurs to delete stale routes. A node expects to receive a regular update from its neighbours; when no updates are received from a neighbour, the node may decide that this neighbour is no longer available as a neighbour. When that occurs, any route that uses that neighbour as a next hop should be deleted, including that node as the actual destination. The main problem of DSDV comes from the time it needs to converge, because a route

can not be used after a certain time elapse from the periodic broadcast. This may not be acceptable in mobile ad hoc networks, where the topology changes frequently. Moreover, the periodic broadcast adds a great amount of overhead.

For illustration, Figure 3.2 presents an example of an ad hoc network consists of 8 nodes before and after the movement of node N1. Table 3.2 is the routing table of node N6 at the moment before the movement. The Install time field is used to determine when to delete stale routes.

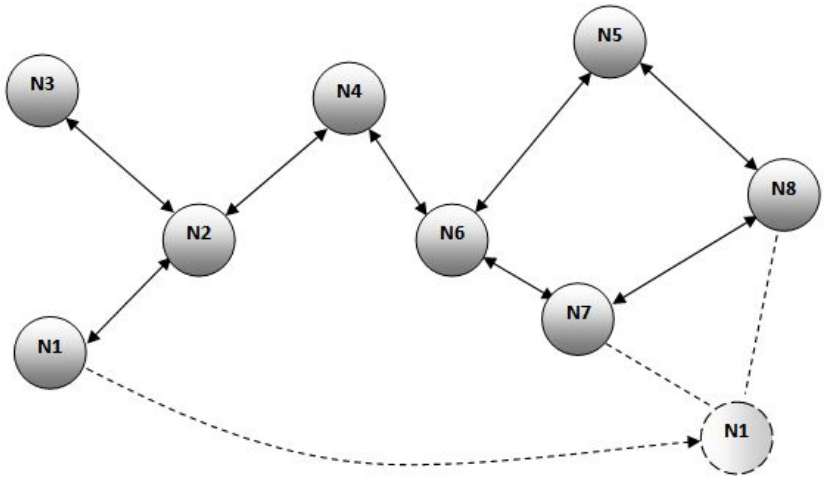

Figure 3.2: An example of Ad hoc Network

Routing Update Process: In the routing table updating process, the node tags each update packet with a sequence number to distinguish stale from new updates. This sequence number is an increasing number that uniquely identifies each update packet from a given node. The update packet may contain more than one entry. So, if a node receives an update packet, the sequence number of each entry is compared with the sequence number of the corresponding node already stored in the routing table. If the sequence number of the update packet is smaller, the newly received information should be ignored because it is stale. If the sequence number of the received information is larger, the entries should be entered into the routing table. If the sequence numbers are equal, then the metric is compared and the route with the

lower metric will be used.

Suppose that node N7 in Figure 3.2 advertises its routing information by broadcasting an update packet to its neighbours as shown in table 3.1

Destination	Next hop	Metric	Seq. No
N7	N7	0	S238_N7
N1	N1	1	S516_N1
N2	N6	3	S228_N2
N3	N6	4	S764_N3
N4	N6	2	S820_N4
N5	N8	2	S502_N5
N6	N6	1	S204_N6
N8	N8	1	S148_N8

Table 3.1: An *Update Packet* advertised by node N7

When node N6 receives the update packet, it checks the received routing information of each entry contained in both the update packet (Table 3.1) and its routing table (Table 3.2). From the table 3.3 which shows the routing table of node N6 after dealing with the received update packet, we can see the following:

- The entries with higher sequence numbers are always entered into the routing table, regardless of metric value. For example, the entry belongs to node N1 has higher sequence number (S516_N1) in the update packet (Table 3.1). This entry is entered into the routing table of node N6 (Table 3.3).
- If sequence numbers are the same, the metric is compared. If the metric in the update packet is smaller, the entry is entered into the routing table, otherwise, the entry is ignored. For example, the entry belongs to node N5 has the same sequence number (S502_N5) in both the update packet and the routing table, but the entry in the routing table (Table 3.2) has a lower metric, so the entry in the update packet is ignored.

- The entries with old sequence numbers in the update packet are always ignored. For example, entries belonging to nodes N2 and N8 have old sequence numbers respectively in the update packet, so both of them are ignored.

Destination	Next hop	Metric	Seq. No	Install
N1	N4	3	S406_N1	T001_N6
N2	N4	2	S238_N2	T001_N6
N3	N4	3	S764_N3	T001_N6
N4	N4	1	S820_N4	T002_N6
N5	N5	1	S502_N5	T812_N6
N6	N6	0	S204_N6	T001_N6
N7	N7	1	S238_N7	T002_N6
N8	N5	2	S160_N8	T811_N6

Table 3.2: Routing table of node N6 when receiving the *Update Packet* from node N7

Destination	Next hop	Metric	Seq. No	Install
N1	N7	2	S516_N1	T810_N6
N2	N4	2	S238_N2	T001_N6
N3	N4	3	S764_N3	T001_N6
N4	N4	1	S820_N4	T002_N6
N5	N5	1	S502_N5	T812_N6
N6	N6	0	S204_N6	T001_N6
N7	N7	1	S238_N7	T002_N6
N8	N5	2	S160_N8	T811_N6

Table 3.3: Routing table of node N6 after dealing with the *Update Packet*

Link Failure Process: To describe the Link Failure Process, let's assume that node N1 moves away from node N7 (Figure 3.2). Node N7 detects that the link between itself and node N1 is broken. Node N7 immediately assigns ∞ metric and increments

the sequence number of node N1 (becomes odd sequence number S517_N1). Next, it generates, and broadcasts an update packet as shown in Table 3.4. When node N6 receives the update packet, it updates its routing table (Table 3.3) with the received information (odd sequence number and ∞ metric) in the entry that belongs to node N1 as shown in Table 3.5. This means that link to node N1 is broken. The route to a lost node is re-created when that node comes back again, and broadcasts an update packet with an equal or greater sequence number and finite metric.

Destination	Next hop	Metric	Seq. No
N7	N7	0	S238_N7
N1	N1	∞	S517_N1
N2	N6	3	S228_N2
N3	N6	4	S764_N3
N4	N6	2	S820_N4
N5	N8	2	S502_N5
N6	N6	1	S204_N6
N8	N8	1	S148_N8

Table 3.4: An *Update Packet* advertised by node N7 with a broken link

Destination	Next hop	Metric	Seq. No	Install
N1	N7	∞	S517_N1	T810_N6
N2	N4	2	S238_N2	T001_N6
N3	N4	3	S764_N3	T001_N6
N4	N4	1	S820_N4	T002_N6
N5	N5	1	S502_N5	T812_N6
N6	N6	0	S204_N6	T001_N6
N7	N7	1	S238_N7	T002_N6
N8	N5	2	S160_N8	T811_N6

Table 3.5: Routing table of node N6 after dealing with the broken link

3.3.2 Improvements on DSDV

Several algorithms have been proposed to improve the performance of the DSDV routing protocol, but to the best of our knowledge, none of them uses the multipath notion. The following briefly describes some of the proposed algorithms.

Ahn et al. present a control mechanism called Adapting to Route-demand and Mobility (ARM) [3]. The mechanism allows any proactive routing protocol to adapt to changes in node mobility and workload route demands. Using this mechanism, each node maintains two metrics to adjust the content and the period of routing updates: a *route-demand metric* indicating which destinations are currently forwarding data and a *mobility metric* indicating how fast its neighbours are currently changing. Due to the decentralization of ARM, each node can adapt independently. ARM is applied to the DSDV routing protocol, coming up with ARM-DSDV. The authors conclude that compared to DSDV, ARM-DSDV achieves a better delivery ratio, while spending a reasonable amount in routing costs.

Khan et al. propose an Efficient DSDV routing protocol for Ad Hoc networks (Eff-DSDV) [52], to improve the performance of DSDV by overcoming the problem of using stale routes. In case of link failure, the proposed protocol creates a temporary link through a neighbour which has a valid route to the destination. When a node discovers a broken link, it broadcasts a Route Request (RREQ) packet to its neighbours. In turn, any neighbour that has a valid route to the destination but where the RREQ sender is not the next hop on the route, returns a ROUTE-ACK packet. The ROUTE-ACK packet includes an *update time* field which is used to select the temporary route. In the case of receiving multiple ROUTE-ACK packets with the same number of hops, the receiving node selects the route which has the latest update time.

Lee et al. propose a proactive routing protocol for multi channels (DSDV-MC), by extending the DSDV routing protocol to a multi-channel version [60]. The protocol uses multiple channels, where multiple useful transmissions can occur simultane-

ously. DSDV-MC divides the network layer into control and data planes. Nodes in the network use the control channel to send routing updates, and use the data channel to send user packets. The authors conclude that DSDV-MC improves the network capacity by exploiting multiple channels. Compared to DSDV, DSDV-MC increases the network throughput in both single-hop and multiple-hop network scenarios, and decreases the packet drop rate when the number of channels increases because packets are distributed over multiple channels.

Most routing protocols assume that all nodes are trustworthy and cooperative. Thus, a single misbehaving node can disrupt the routing operations of a whole network. As a result, a packet may not reach the desired destination or the packet may be routed through a route in the control of an adversary. For example, a misbehaving node may advertise routes with fraudulent sequence numbers or cost metrics, to taint the routing tables of other nodes or to affect routing operations. Wan et al. propose a secure routing protocol based on DSDV, called S-DSDV [118]. S-DSDV uses consistency checks to discover sequence number frauds and distance frauds in DSDV. The protocol has two security properties, provided that no two nodes are in collusion: detection of any distance fraud (longer, same, or shorter), and detection of both larger and smaller sequence number fraud. In S-DSDV, the misinformation can be stopped before it spreads into the entire network, since the misbehaving node is surrounded by well-behaved nodes that can contain it.

Wang et al. present a Secure Destination-Sequenced Distance-Vector routing protocol for mobile ad hoc networks (SDSDV) [119]. SDSDV is based on the regular DSDV protocol. Using SDSDV, each node maintains two one-way hash chains for each node in the network. The hash chain approach is used to secure the sequence numbers and metrics. Two additional fields, called AL (alteration) and AC (accumulation) fields, are added to each entry of the update packets to carry the hash values. Using AL and AC fields in the entry, any node in a route cannot arbitrarily increase or decrease the sequence number and metric. Thus, SDSDV can provide better protection of routing messages.

Kumar et al. propose an Optimal Path Routing protocol (OPR) [57]. OPR is a DSDV Based New Proactive Routing Protocol, and works proactively using an optimal path routing mechanism. The protocol assumes that each node in the network is equipped with GPS receivers, to locate the node positions. OPR keeps the routing overheads to the minimum by maintaining only the routing tables of neighbours and neighbours of the neighbour nodes (i.e., Each node stores routing information of 1-hop and 2-hop nodes). When a node sends a packet, the neighbour node, which is closest to the destination is selected to forward the packet.

Chang et al. propose a method called Light DSDV (LDSDV), to reduce the routing overhead in DSDV [20]. The method benefits from the nature of DSDV, such as shortest path and loop-free, but alleviates a flooding problem of control messages when the network topology changes. LDSDV takes advantage of the spanning trees and maintains the relationship between nodes under each spanning tree. When a node receives a routing message of a destination node, it runs a procedure to determine whether the message should be forwarded. The authors conclude that LDSDV alleviates routing overheads by filtering out a great amount of redundant messages at leaf nodes, especially for Ad Hoc networks with high density.

3.3.3 The Wireless Routing Protocol (WRP)

The Wireless Routing Protocol (WRP) [82][83] is a proactive unicast routing protocol, and was one of the first routing protocols for mobile ad hoc networks. WRP uses an improved version of the distance vector routing protocol that uses the Bellman-Ford algorithm to calculate paths.

Each mobile node maintains four tables: Distance table, Routing table, Link-Cost table and Message Retransmission List table (MRL). Each entry in the routing table contains the destination's id, the distance (metric) to the destination node, the predecessor node and the successor node of the chosen shortest path to the destination, and a tag to identify the state of the path (i.e., is it a simple path, a loop or invalid).

The predecessor and successor are stored in the routing table to avoid the counting-to-infinity problem, and to detect routing loops. A mobile node creates an entry for each neighbour in a separate table called the link-cost table. The entry contains the cost and status of the link to each current neighbour. The MRL table contains the following fields: the sequence number of the update message, a retransmission counter which is decremented each time the node sends a new update message, a flag that specifies whether the node has sent an ACK, and a list of updates sent in the update message.
In WRP, mobile nodes are required to broadcast an update message periodically. If there is no change in its routing table since the last update, a node is required to send a Hello message to ensure its connectivity. The update message contains a list of updates (destination, distance to the destination, and the predecessor of the destination), and a list of responses that indicate which nodes should acknowledge (ACK) the update. After sending an update message to its neighbours, a node expects to receive an ACK from all of them. If an ACK has not been received from a particular neighbour, the node will record the nonresponding neighbour in its MRL table, and send another update to that neighbour later.

One of the main drawbacks of WRP is that it needs a large amount of memory storage to maintain several tables. Moreover, as a proactive routing protocol, it has limited scalability and is not suitable for large mobile ad hoc networks [69].

3.3.4 The Fisheye State Routing (FSR)

Fisheye State Routing (FSR) [91][92] is a proactive routing protocol based on the Link State routing algorithm. As the name indicates, FSR utilizes a function similar to a fish eye, where the eye captures the pixels with high detail near the focal point. As the distance from the focal point increases, the detail decreases. FSR maintains accurate distance and path quality information about its immediate neighbouring nodes.

FSR uses the notion of multi-level fisheye scope to reduce the routing update overhead in large networks. The scope is defined as a number of nodes which can be reached within a specified number of hops. The number of levels and the radius of each scope depend on the size of the network.

FSR is similar to many LS Routing protocols in that it maintains a topology map at each node. The main difference is the way that routing information is propagated. Instead of flooding link state information into the entire network, nodes in FSR maintain a link state table according to up-to-date information received from their neighbours, and periodically exchange it only with neighbouring nodes. By this exchange process, table entries are updated where entries with smaller sequence numbers are replaced by those with larger sequence numbers.

In dynamic networks, the network topology changes frequently. Exchanging the entire topology table among neighbours consumes a considerable amount of bandwidth. Instead of being event driven, FSR uses periodic updates to avoid the excessive overhead caused by flooding link state updates. Moreover, FSR uses different exchange periods for different entries in the routing table to reduce the routing update overhead. FSR avoids extra work to find the destination (as in a reactive routing) by retaining a routing entry for each destination node.

The authors in [91] conclude that FSR is a flexible solution to the challenge of maintaining accurate routes in ad hoc networks, if the number of scope levels and radius size are properly chosen.

3.3.5 The Optimized Link State Routing Protocol (OLSR)

The Optimized Link State Routing (OLSR) protocol [23][42] is a proactive routing protocol and it is an Optimization over the pure link state protocol. In OLSR each node maintains topology information by periodically exchanging link-state messages. OLSR minimises the size of the control packet by including only a subset of its current

neighbours, and minimizes flooding by the use of a MultiPoint Relay (MPR) strategy. These two optimizations make OLSR suitable for use in large and dense networks

Using the MPR technique, each node selects a number of its current neighbours as its MPRs which are allowed to rebroadcast control packets. When a node receives a control packet, it only rebroadcasts the packet if it is a MPR of the sending node. Otherwise, it only reads and processes it but does not rebroadcast the control packet. To determine the MPRs, every node periodically broadcasts a *Hello message* containing a list of its one hop neighbours and their link status (Symmetric or Asymmetric). When a node receives a Hello message, it selects a subset of one hop neighbours, which covers all of its two hop neighbours. One issue with OLSR is how its nodes decide whether a link is symmetric. The answer is simple, if a node receives a Hello message and sees its own address in the sender's Hello message, then it considers that the link is symmetric.

Instead of using a simple flooding mechanism, OLSR uses MPR-flooding which aims to minimize the problems caused by duplicate reception of a message within a region. MPRs are used to disseminate topology information through the network. Each node acting as a MPR creates and broadcasts Topology Control (TC) messages to all its 1-hop neighbour nodes. Also, MPRs rebroadcast to their 1-hop neighbours the TC messages that are received from nodes within its MPR Selector Set. A TC message contains a list of neighbour nodes that selected the TC's sender node as a MPR and a MPR Selector Sequence Number (MSSN) which is incremented for every new TC message created.

OLSR maintains a *neighbours* table, where a node records the information about one hop neighbours, the status of the link, and a list of two hop neighbours which these one hop neighbours can give access to. Upon receiving Hello messages, a node can construct its *MPR Selector* table that contains the nodes who have selected it as MPR. Each node in the network maintains another table called a *topology* table where it stores the topological information about the network. The topology table

contains the address of the destination node (T_dest), the address of the last hop to the destination (T_last), the sequence number of the TC message (T_seq), and a holding time which indicates the time that this tuple expires (T_time). Finally, each node uses the information in the neighbour table and the topology table to construct its routing table. Each entry in the routing table consists of the destination node (R_dest), the next hop to the destination node (R_next) and number of hops to the destination node (R_dist). During route evaluation, the shortest path algorithm is used.

3.3.6 The Global State Routing (GSR)

The Global State Routing (GSR) protocol [21] is based on the traditional Link State algorithm. As in link state protocols, routing messages are generated on a link change. GSR restricts the update messages to be between intermediate nodes only. Each node in GSR maintains 1 list and 3 tables:

- *Neighbours List:* contains a set of nodes that are in the node's transmission range.
- *Topology Table:* has an entry for each destination, where each entry contains the link state information as reported by the destination node and the timestamp of this information.
- *Next Hop Table:* For every destination, the Next Hop Table contains the next hop to forward the packets to this destination.
- *Distance Table:* contains the shortest path to each destination node.

The details of GSR protocol can be summarized as follows: At the beginning, each node starts with an empty Neighbour List, and an empty Topology Table. Nodes learn about their neighbours by examining the sender field of each packet in its inbound queue and add all routing packet senders to its Neighbour List. Then, the

node processes the received routing message and updates its Topology Table if the sequence number of the message is newer than the sequence number that is stored in the table. After the routing messages are examined, the node rebuilds its routing table according to the newly computed Topology Table and broadcasts the new information to its neighbours. The main difference between GSR and traditional LS is the way that the node uses to disseminate the routing information. In LS, link state packets are generated and flooded into the entire network whenever topology changes are discovered. Whereas, nodes in GSR maintain their link state tables according to up to date information received from their neighbours. The authors in [21] conclude that GSR is more desirable for a mobile environment where mobility is high and bandwidth is relatively low.

3.3.7 Clusterhead Gateway Switch Routing (CGSR)

The Clusterhead Gateway Switch Routing (CGSR) [22] is a hierarchical routing protocol where the nodes are grouped into clusters. Each cluster is maintained by a cluster-head, which is a mobile node that is elected to manage all the other nodes within the cluster. The elected node controls the transmission medium and inter-cluster communications occur via this node. The advantage of CGSR is that it has low routing overheads because each node only maintains routes to its cluster-head. However, the disadvantage of having a cluster head scheme is that the frequent cluster head changes may affect the routing protocol performance because nodes are busy in cluster head selection rather than packet forwarding. So, a Least Cluster Change (LCC) clustering mechanism is invoked. Using LCC, the cluster heads change only if two cluster heads come into contact or when a node moves away from all the other cluster heads.

CGSR has much of the same overhead as DSDV because it uses DSDV as the underlying routing scheme. In order to improve DSDV, CGSR uses a routing approach to forward traffic from the source node to the destination node called the cluster-head-

to-gateway routing approach. Gateway is a node located in the transmission range of two or more cluster heads. When a node plans to send a packet, the packet is first sent to its cluster head, and then it is forwarded to a gateway to another cluster head. This process is repeated until the cluster head of the destination node is reached. Then, the packet is transmitted to the destination node.

Nodes in CGSR should maintain a cluster member table, to store the destination cluster head for each node in the network, and these tables are periodically broadcast using the DSDV algorithm. Each node updates its cluster member table on reception such a table. In addition to the cluster member table, each node maintains a routing table that is used to determine the next hop in order to reach the destination node. Upon receiving a packet, the node consults its cluster member table and routing table to determine the nearest cluster head to reach the destination. Then, it checks its routing table to determine the next hop used to reach the cluster head.

3.3.8 Topology Broadcast based on Reverse Path Forwarding (TBRPF)

TBRPF [87] is a proactive routing protocol designed to be used in mobile ad-hoc networks. It is a link state based routing protocol, which performs hop-by-hop routing. TBRPF uses the concept of Reverse-Path Forwarding (RPF) to broadcast its update packets in the reverse direction along the spanning tree, which is made up of the minimum-hop paths from other nodes leading to the source of the update message. Also, the protocol uses the topology information that is received along the broadcast trees to compute the minimum-hop paths from the trees themselves.

TBRPF has two modes: partial topology (PT) and full topology (FT). TBRPF-PT achieves a further reduction in control traffic, by providing each node with the state of only a small subset of links. As a result, each node reports only changes to a small part of its source tree. TBRPF-PT is recommended for dense and large networks where it achieves less control traffic than TBRPF-FT. In contrast, TBRPF-FT provides each

node with the state of all links in the network. It uses the concept of reverse-path forwarding to reliably broadcast each topology update in the reverse direction along the dynamically changing broadcast tree formed by the minimum-hop paths from all nodes to the source of the update. So, each node gets the state of each link in the network. TBRPF-FT computes optimal routes based on the advertised link states. It is recommended for sparse networks and when full topology information is needed.

3.3.9 Distance Routing Effect Algorithm for Mobility (DREAM)

DREAM [9] is a location based routing protocol. DREAM can be considered to be a proactive routing protocol in the sense that a mechanism is defined for the dissemination and updating of location information.

In DREAM, each node maintains a routing table (called a location table) which contains location information of all other nodes in the network. An entry in the routing table includes a node identifier, the direction of and distance to the node, as well as a time of generating this information. Each node determines its own position using GPS [18], or some other type of positioning service. Each node periodically broadcasts control packets that contain its location information to all other nodes. The frequency of broadcasting control packets is determined by the distance and node mobility rate. Nodes that are far apart, need to update each other's locations less frequently than nodes that are closer together, and the faster a node moves, the more often it needs to advertise its new location.

If a source needs to send a data packet, it calculates the direction towards the destination, and selects a set of one-hop neighbours that are located in the destination's direction. Then, the set is enclosed in the data packet header and transmitted with the data. Only nodes that are specified in the packet header are qualified to receive and process the data packet. The receiving nodes select their own set of one-hop neighbours, update the data packet header and send the packet out. When the destination

node receives the data, it responds by sending an ACK to the source node in a similar way. If the source does not receive an ACK for data sent through a designated set of nodes, it retransmits the data again by pure flooding.

3.3.10 Summary of Proactive Routing

In summary, routing information is always propagated and maintained in proactive routing protocols. Thus, a route to every other node in the network is always available, regardless of whether or not it is needed. Although this feature is useful for datagram traffic, it incurs substantial signalling traffic and power consumption. Because of the limited bandwidth and battery power of mobile nodes, this becomes a serious limitation. Proactive routing protocols can be categorised as flat routed global routing protocols and hierarchically routed global routing protocols.

Flat routed global routing protocols do not scale very well, because they use updating procedures that consume a significant amount of network bandwidth. Using this type of protocol, the routing overhead increases when the network becomes larger. Compared to other flat structure protocols, OLSR and DREAM scale the best because they reduce the amount of overhead transmitted through the network. OLSR reduces the control overhead using the MultiPoint Relay (MPR) strategy, and DREAM reduces the control overhead by exchanging location information rather than complete (or partial) link state information. In contrast, The hierarchically routed global routing protocols scale better than most flat routed protocols. This is because they introduce a network structure to control the amount of overhead transmitted through the network. This is achieved by allowing selected nodes (e.g., clusterhead) to rebroadcast control information.

3.4 Reactive Routing Protocols

Reactive routing is also known as "on-demand" routing. It creates routes only when needed by the source node. They are based on some kind of "query-reply" dialogue and they do not maintain an up-to-date topology of the network. As a node has some data to be sent, it invokes a route discovery procedure to find a route to the desired destination. Such a procedure uses flooding the network with the route discovery, and terminates if the query reaches the destination or reaches an intermediate node that has an active route to the destination. Compared to the proactive routing protocols for mobile ad hoc networks, reactive routing protocols have less control overhead. Thus, reactive routing protocols have better scalability than proactive routing protocols in mobile ad hoc networks. On the other hand, using reactive routing protocols, the source nodes may suffer from long delays before they become able to forward data packets. The following subsections give brief description of three reactive routing protocols: Ad hoc On- demand Distance Vector routing (AODV) [95], Dynamic Source Routing (DSR) [47], and Temporally Ordered Routing Algorithm (TORA) [90].

3.4.1 Ad hoc On- demand Distance Vector routing protocol (AODV)

Ad hoc On-Demand Distance Vector Routing Protocol (AODV) [19][93][95] is a unicast reactive routing protocol, where the routes are constructed only when needed. AODV maintains a routing table where routing information about the active paths is stored.

AODV protocol use four control packets: Hello messages, Route Requests (RREQs), Route Replies (RREPs), and Route Errors (RERRs). Each node maintains a routing table which contains: Destination, Next Hop, Number of hops (metric), Sequence number for the destination, Active neighbours for this route, and Expiration time for the route table entry. Each time a route entry is used, the timeout of the entry is reset

to the current time plus active route timeout. The sequence number is used to ensure loop freedom in distance vector routing protocols. The sequence number is sent with RREQ (for source) and RREP (for destination) and stored in the routing table. The larger the sequence number the newer the route information. If a new route is offered, the sequence numbers of the new route and the existing route are compared. The route with the greater sequence number is used. If the sequence numbers are the same, then the new route is selected only if it has fewer number of hops.
AODV is composed of two mechanisms: Route Discovery and Route Maintenance:

1. *Route Discovery:* When a node needs to send data to a destination, it checks its routing table if it has a valid route to that destination. If a route is found, the node starts to send the data to the next hop. Otherwise, it begins a route discovery procedure. In the route discovery procedure, a route request (RREQ) and route reply (RREP) packets are used to establish a route to the destination. RREQ is broadcast throughout the entire network. Upon receipt of RREQ, the node creates a reverse routing entry towards the source, which can be used to forward replies later. The destination or an intermediate node, which has a valid route towards the destination, answers with a RREP packet. When a node receives RREP, a reverse routing entry towards the originator of RREP is also created, the same as with the processing of RREQ. Associated with each routing entry is a so-called precursor list, which is created at the same time. The precursor list contains the upstream nodes which use the node itself towards the same destinations.

2. *Route Maintenance:* Each node along an active route periodically broadcasts *HELLO messages* to its neighbours. If the node does not receive a *HELLO message* or a data packet from a neighbour for a certain amount of time, the link between itself and the neighbour is considered to be broken. In case of the destination with this neighbour as the next hop is not far away (from the invalid routing entry), a local repair mechanism may be started to rebuild the route towards the destination; otherwise, a Route Error (RERR) packet is sent

to the neighbours, which in turn propagates the RERR packet towards nodes whose routes may be affected by the broken link. Then, the affected source can re-initiate a route discovery process if the route is still needed.

3.4.2 Dynamic Source Routing Protocol (DSR)

The second reactive routing protocol is the Dynamic Source Routing Protocol (DSR) [47]. It is based on the concept of source routing. Unlike other unicast routing protocols, DSR does not maintain a routing table, but uses a Route Cache to store the full paths to the known destinations. Unlike other protocols, DSR requires no periodic packets. For example, it does not use any periodic routing advertisements. The lack of periodic activity may reduce the control overhead. The protocol is composed of two mechanisms to discover and maintain the source routes: Route discovery and Route Maintenance.

1. *Route discovery:* When a node has a ready data packet to send, it first searches for a route to the destination in its route cache. If an active route entry towards the destination is found, it uses the found route to send the data packet. Otherwise, the source node initiates route discovery by broadcasting a Route Request (RREQ) packet. The RREQ packet contains the source node's address, the destination node's address, and a unique request id. Also, each RREQ contains a record listing the address of each intermediate node that forwarded the packet. Each intermediate node receiving the RREQ packet checks whether it has a route to the destination. If it does not have a route, it adds its own address to the route record of the packet and then broadcasts it to its neighbours. To limit the number of route requests propagation, the node only broadcasts the RREQ if it has been received for the first time.

   A route reply (RREP) is generated, when the RREQ is received by the destination or an intermediate node that has an unexpired route to the destination.

If the receiving node is the destination, it places the route record contained in the RREQ into the RREP. If the receiving node is an intermediate node, it appends its cached route to the route record and then generate the RREP. If the responding node has a route to the RREQ initiator, the route can be used to return the RREP packet. Otherwise, if symmetric links are supported, the responding node may reverse the route in the route record. If symmetric links are not supported, the node initiates its own route discovery and piggybacks the RREP packet on the new route request.

2. *Route maintenance:* Unlike proactive routing protocols and AODV, DSR does not introduce a periodic HELLO message. Every node along the path is responsible for the validity of the downstream link connecting itself with the next hop. If a broken link is detected, route maintenance is invoked. This phase is accomplished through the use of Route Error (RERR) packets and acknowledgements. A RERR packet is generated at a node that discovers a link failure and sent to the source node. When an RERR packet is received, the hop in error is removed from the node's route cache and all routes containing the hop are truncated at that point. In addition to RERR packets, acknowledgements are used to verify the correct operation of the route links. When the source node receives the RERR packet, it may re-initiate route discovery if an alternate route is not found.

3.4.3 The Temporally Ordered Routing Algorithm (TORA)

The Temporally Ordered Routing Algorithm (TORA) [90] is a reactive routing algorithm based on the concept of link reversal. TORA is proposed to operate in large, dense mobile networks. In these kind of networks, the protocol involves a localized single pass of the distributed algorithm as a reaction to link failures.

TORA uses three control packets: query (QRY), update (UPD), and clear (CLR).

QRY packets are used to create routes, UPD packets are used to create and maintain routes, and CLR packets are used to erase routes.

In TORA, the network topology is regarded as a directed graph. A Directional Acyclic Graph (DAG) is accomplished for the network by assigning each node *i* a height metric h_i. The height of a node is defined as a quintuple which includes the logical time of a link failure, the unique ID of the node that defines the new reference level, a reflection indicator bit, a propagation ordering parameter and a unique ID of the node. The first three elements represent the reference level which is defined each time a node loses its last downstream link, and the last two elements define an offset with respect to the reference level. A link direction from *i* to *j* means $h_i > h_j$ (i.e., the height of node *i* is greater than the height of node *j*). DAG provides TORA with the capability that many nodes can send packets to a given destination and guarantees that all routes are loop-free.

The protocol can be divided into three functions: creating routes, maintaining routes, and erasing routes.

Creating routes: This operation starts with setting the height (propagation ordering parameter in the quintuple) of the destination to 0 and heights of all other nodes to NULL (i.e., undefined). The source broadcasts a Query packet containing the destination's ID. A node with a non-NULL height responds by broadcasting an update packet containing the height of its own. Upon receiving an update packet, a node sets its height to one more than that of the update packet generator. A node with higher height is considered as upstream and the node with lower height is considered as downstream. In this way, a directed acyclic graph is constructed from the source to the destination and multiple paths route may exist.

Maintaining routes: The DAG in TORA may be disconnected because of node mobility. Thus, route maintenance operation is an important part of TORA. TORA has the unique feature that control messages are localized into a small set of nodes near the occurrence of topology changes. After a node loses its last downstream link, it generates a new reference level and broadcasts the reference to its neighbours. There-

fore, links are reversed to reflect the topology change and adapt to the new reference level.

Partition detection and erasing routes: When a node detects a network partition, it initiates the process of erasing the invalid routes. Network partition is detected when a 'reversal' reaches a node with no downstream links and all of its neighbours have the same "reflected reference level". The erase operation in TORA floods CLR packets through the network and erases invalid routes.

3.4.4 Summary of Reactive Routing

In reactive routing protocols the route is established only when it is needed. In other words, when a node has ready data to send, it initiates route discovery to establish a route to the destination. Most of the reactive routing protocols have the same routing cost when considering worst-case scenarios. This is because they use similar route discovery and maintenance procedures. The worst-case scenario applies to most routing protocols during the initial stages (i.e., when there is no previous communication between the source and the destination). As nodes stay active for a longer time, they become more aware of their neighbours. Unlike proactive routing protocols, reactive protocols do not use periodic route updates which may decrease the control overhead. This kind of protocols suffers from delay, and the signalling traffic generated grows when the mobility of the nodes increases.

3.5 Hybrid Routing Protocols

Hybrid routing protocols are a new generation of protocols [2], that combine the advantages of both proactive and reactive routing protocols and overcome their shortcomings. Normally, this approach exploits hierarchical network architectures. A proactive routing approach and a reactive routing approach are used at different hierarchical levels, respectively.

Scalability is an important factor in designing an efficient routing protocol for wireless ad hoc networks. A good routing protocol has to be scalable and adaptive to the changes in the network topology. Thus a scalable protocol should perform well as the network grows larger or as the workload increases. The key characteristic of a reactive protocol is that the source node invokes a route discovery procedure to discover routes. Whenever a source node needs a route to an unknown destination, it initiates a route discovery process by flooding a route request for the destination and waits for a route reply. Each route discovery flood is associated with significant latency and overhead. This is particularly true for large networks. Excessive flooding can lead to network clogging and as a result the protocol's performance becomes worse. Hence it is desirable to keep the route discovery frequency low to improve the scalability of reactive routing protocols. Multipath routing and hierarchical routing can reduce the frequency of on-demand route discovery.

Hybrid routing protocols are designed to increase scalability by reducing route discovery overheads. This can be achieved by proactively maintaining routes to the near nodes and determining routes to far nodes using a route discovery mechanism. Most of the proposed hybrid protocols are zone-based, which means that the network is partitioned into zones by each node. Others group nodes are partitioned into trees or clusters. In the next subsections, we describe some typical hybrid mobile ad hoc network routing protocols. Specifically, we introduce the Zone routing protocol (ZRP) in subsection 3.5.1, Zone-based Hierarchical Link State (ZHLS) in subsection 3.5.2, Distributed Spanning Trees based routing protocol (DST) in subsection 3.5.3, Distributed Dynamic Routing (DDR) in subsection 3.5.4, and Sharp Hybrid Adaptive Routing Protocol for Mobile Ad Hoc Networks (SHARP) in subsection 3.5.5,

3.5.1 Zone Routing Protocol (ZRP)

The Zone Routing Protocol (ZRP) [36] is based on the notion of routing zones that are defined for each node. This includes all nodes whose distance is less than some

predefined number; the distance is referred to as the zone radius. Each node has to know the topology of the network within its zone, and nodes are updated about topological changes within their zone. Thus, even if the network is very large, the updates are only propagated locally. For a zone radius greater than one, routing tends to be robust.

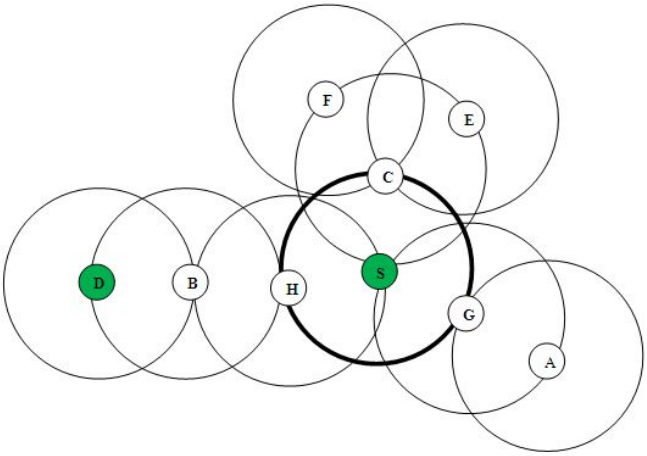

Figure 3.3: Example of Zone Routing

Figure 3.3 illustrates the route discovery of the protocol. Suppose node *S* needs to send data to node *D*. At the beginning, node *S* verifies that node *D* is not within its zone. Then node *S* sends a request to all nodes on the Periphery of its zone (i.e, nodes *C*, *G*, and *H*). Upon receiving the query, each node verifies that node *D* is not within its zone, and broadcast the query to their peripheral nodes. In this example, node *H* receives the query and sends the query to node *B*, which recognizes that node *D* is in its routing zone. Node *B* responds to the query, indicating that the forwarding path is *S-H-B-D*. Node *B* learns about the forwarding path using the Route accumulation mechanism.

A hop-count is included in the query message to limit its forwarding process. The initial value of hop-count is set to a maximal value and decreased by one each time the query is forwarded. When the value of the hop-count reaches zero, the message is discarded.

The behaviour of ZRP can be adjusted by changing the value of the zone radius. For a large zone radius, ZRP works as proactive protocol, whereas for small zone radius, ZRP works as a reactive protocol. Route evolution and routing zone update protocols are two ZRP related protocols that work in conjunction with the route discovery protocol. The route evolution changes the network-wide routes as a response to the changes in the connectivity of nodes on the path. The route zone update protocol allows each node to learn the complete topology of its zone.
Generally, in ZRP, different routing approaches are used for inter-zone and intra-zone packets. The proactive routing approach called Intra-zone Routing protocol (IARP), is used inside routing zones, and the reactive routing approach called Inter-zone Routing Protocol (IERP), is used between routing zones, respectively. Therefore, if the source and destination nodes are in the same zone, a route can be available immediately. Most of the existing proactive routing schemes can be used as the IARP for ZRP. The IERP reactively initiates route discovery when the source node and the destination are located in different zones.

3.5.2 Zone-based Hierarchical Link State (ZHLS)

Zone-based Hierarchical Link State (ZHLS) [44] is a routing protocol that employs a hierarchical structure and divides the network into non-overlapping zones. Each node participating in the network has a node ID and a zone ID, which is calculated using a GPS. The hierarchical topology is made up of two levels: node level topology and zone level topology. Unlike CGSR [22] protocol, ZHLS does not use a cluster-head or location manager to coordinate the data transmission, which means that there is no processing overhead associated with the cluster-head or location manager selection. This also means that traffic bottlenecks can be avoided. When a route to a remote destination is required (i.e., the destination is in another zone), the source node broadcasts a zone level location request to all other zones, which generates significantly lower overhead when compared to the flooding approach in reactive protocols.

The disadvantage of ZHLS is that all nodes must have a preprogrammed static zone map in order to function. This may not feasible in applications where the geographical boundary of the network is dynamic. Nevertheless, it is highly adaptable to dynamic topologies and it generates far less overhead than pure reactive protocols, which means that it may scale well to large networks.

3.5.3 Distributed Spanning Trees based routing protocol (DST)

In the Distributed Spanning Trees based routing protocol (DST) [96] the nodes in the network are grouped into a number of trees. A node participating in the network can be either the root of a tree or an internal node of a tree. The root controls the structure of the tree and whether the tree can merge with another tree. Each node can be in one of three different states: router, merge and configure. The state of the node depends on the type of task that it is trying to perform.

To determine a route, DST proposes two different routing strategies: Hybrid Tree Flooding (HTF) and Distributed Spanning Tree shuttling (DST). In HTF, control packets are sent to all neighbours and adjacent bridges in the spanning tree, Moreover, packets are held at each node for a period of time called the *holding time*. During this time, new bridges are made at the node, and the packets are sent along the bridges. In DST, the control packets are sent from the source along the tree edges. When a control packet reaches down to a leaf node in the tree, it is sent up the tree until it reaches a certain height, called the *shuttling level*. When the shuttling level is reached, the control packet can be sent down the tree or to the adjacent bridges.
The main disadvantage of the DST algorithm is that it relies on a root node to configure the tree, which creates a single point of failure. Furthermore, the holding time used to buffer the packets may introduce extra delays.

3.5.4 Distributed Dynamic Routing (DDR)

Distributed Dynamic Routing (DDR) [86] is another tree-based routing protocol. In DDR, neighbouring nodes periodically exchange beaconing messages to construct a forest. Each tree of the constructed forest forms a zone, and each zone is assigned a zone ID. The entire network is partitioned into a number of non-overlapping zones. Each zone is connected via gateway nodes. Gateway nodes are the nodes that are in the transmission range of each other, but belong to different trees. Each node in the network can either be in a router or non-router mode with regard to its position in its tree.
Each node computes periodically its zone ID independently. Each zone is connected via nodes that are not in the same tree but are in the direct transmission range of each other. So, the whole network can be seen as a set of connected zones.

The DDR protocol consists of six phases: preferred neighbour election, forest construction, intra-tree clustering, inter-tree clustering, zone naming, and zone partitioning.
Each node starts by electing the preferred neighbour node, which is the node that has the most number of neighbours. After that, a forest is constructed by connecting each node to its preferred neighbour. Then, the intra-tree clustering algorithm is invoked to determine the structure of the zone and build up the intrazone routing table. Next, the inter-tree algorithm is used to determine the connectivity with neighbouring zones. Each zone is assigned a name by running the zone naming algorithm, and finally, the network is partitioned into a set of non-overlapping zones.
To establish routes, a hybrid ad hoc routing protocol (HARP) [85] works on top of DDR. HARP uses the intra-zone and inter-zone routing tables created by DDR to determine a path between the source and destination nodes.

The advantage of DDR is that it does not rely on a static zone map to perform routing. Also, it does not require a root node or a clusterhead, to coordinate data and control packet transmission between different nodes and zones. In contrast, the disadvantage

of DDR is that the nodes that have been selected as preferred neighbours may become performance bottlenecks, because they may transmit more routing and data packets than the other nodes. Furthermore, if a node is elected as a preferred neighbour for many of its neighbours, they may need to communicate with it. This may lead to increasing in channel contention around the preferred neighbour, and will result in larger delays being experienced by all neighbouring nodes before they can reserve the medium.

3.5.5 Sharp Hybrid Adaptive Routing Protocol for Mobile Ad Hoc Networks (SHARP)

Sharp Hybrid Adaptive Routing Protocol (SHARP) [99] is a routing protocol that adapts efficiently between proactive and reactive routing strategies. SHARP adapts between the two strategies by dynamically varying the amount of routing information that is shared proactively. This is done by defining a proactive zone around some nodes, and letting a node-specific *zone radius* determine the number of nodes within each proactive zone. Each node at a distance less than or equal to the zone radius is considered as a member of the proactive zone for that node. All nodes that are not in the proactive zone of a given destination use reactive routing protocols to establish routes to that node.

SHARP is composed of two components: a proactive routing component, and a reactive routing component.

Proactive Routing Component: SHARP uses SHARP Proactive Routing protocol (SPR) which is based on DSDV [94] and TORA [90]. SPR performs proactive routing by generating and maintaining a Directed Acyclic Graph (DAG) rooted at the destination. The DAG is generated periodically using a construction protocol. The destination node initiates the construction process by generating a DAG *construction packet*, which carries the zone radius and a sequence number to distinguish a new DAG from an old DAG. The time to live (TTL) field is set to the zone radius and is

propagated within the proactive zone. It builds a DAG by assigning a height to each node. The height corresponds to the distance of the node from the destination. When a node receives a *construction packet* from another node, it adds the link between the two nodes to the DAG, increments its height by one, and rebroadcasts the construction packet. For example, if node B receives a construction packet from a node A, node B adds the link $A \rightarrow B$ to the DAG and rebroadcasts the construction packet after incrementing its height by one.

Each node in the proactive zone broadcasts an update packet which includes its current height. When a node receives update packets from its neighbours, it records the height of each neighbour. The update packets serve as HELLO beacons to detect the construction of new links and the breakage of current links. A new link is constructed when receiving a new update packet from a new neighbour, whereas the link is considered as broken link, when at least *beacon_loss* consecutive update packets are not received from a neighbour.

Reactive Routing Component: SHARP reactive routing protocol is based on AODV [93]. SHARP uses the standard AODV with some optimizations such as route caching and expanding ring search. If the source is outside the proactive zone of the destination, the source node uses AODV to broadcast a route request. Nodes in the proactive zone of destination respond by sending a route reply to the source node.

SHARP integrates both of the proactive and the reactive components without incurring additional overhead. When the source is within the proactive zone, routing is performed proactively. Otherwise, route requests are broadcast by AODV, and the nodes in the proactive zone of the destination respond by generating a route replies.

3.5.6 Summary of Hybrid Routing Protocols

Hybrid routing protocols combine the advantages of proactive and of reactive routing protocols, and overcome their shortcomings. These protocols are designed to increase scalability, by allowing nodes that are in close proximity to each other to

work together to reduce route discovery overheads. This can be achieved by proactively maintaining routes to near nodes, and reactively determining routes to far away nodes. Many hybrid routing protocols are zone-based, where the network is partitioned or can be seen as a number of zones by each node. Other hybrid routing protocols group nodes into trees or clusters.

3.6 Multipath routing protocols

Multipath routing is a routing technique that is used to find multiple paths between a single source and a single destination. It is one of the ways to improve the reliability of the transmitted information. Multiple paths can be used to provide load balancing, fault tolerance, and bandwidth aggregation [117]. Recently, several multipath routing protocols have been proposed, and many of them are based on the popular on-demand routing protocols, DSR and AODV [2]. In the case of using a reactive routing protocol, maintaining multiple routes for each destination increases the reliability of the protocol by selecting an alternative route without initiating a route discovery procedure.

Numerous of the proposed multipath routing protocols produce disjoint paths which have the desirable property that they are more likely to fail independently. Thus they have a better utility. There are two types of disjoint paths: node disjoint paths and link disjoint paths. Node disjoint paths do not have any nodes in common, except for the source and the destination. Whereas, link disjoint paths do not have any common links, but may have common nodes. Multipath routing protocols can be categorized into two types according to how they use multiple routes: as backup routes for fault tolerance [53][58][75], and as data transfer routes for load balancing [28][59][120][124]. Some of the proposed multi-path algorithms are presented in the following subsections.

3.6.1 Ad hoc On-demand Multipath Distance Vector Routing (AOMDV)

Ad hoc On demand Multipath Distance Vector (AOMDV) [75][76][77], is an extension to the AODV routing protocol. AOMDV is designed to provide efficient recovery from route failures and efficient fault tolerance. To achieve these goals, AOMDV computes multiple loop-free and link-disjoint paths. A notion of *advertised hopcount* is used to guarantee loop freedom, and a particular property of flooding is used to achieve Link-disjointness of multiple paths. The advertised hopcount of a node for a destination represents the maximum hopcount of multiple paths for the destination at the node.

Figure 3.4 shows the structure of the routing tables entries for both AODV and AOMDV routing protocols.

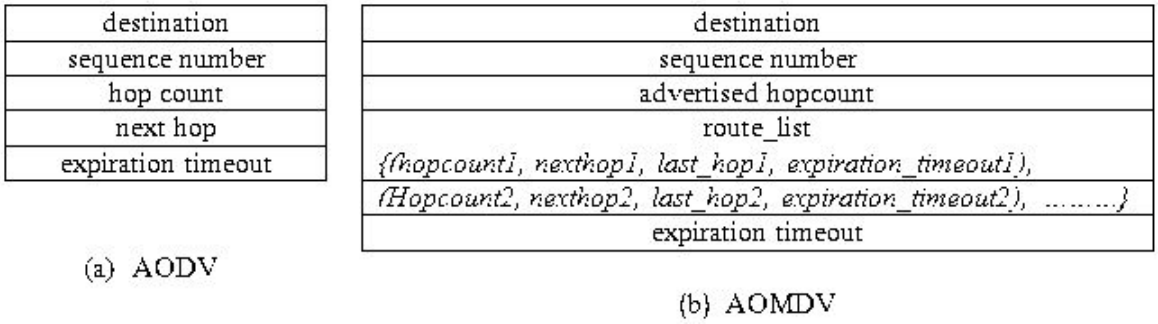

Figure 3.4: The structure of routing table entries for AODV and AOMDV [77]

When the AODV single path routing protocol is used, new route discovery is needed in response to every route break. This inefficiency can be avoided by having multiple paths for each destination. In this case, new route discovery is only needed when all paths are broken. The AOMDV protocol has two components: a rule to create and maintain multiple loop free paths, and a distributed protocol to find link-disjoint paths. The basic idea for finding link-disjoint paths is as follows. To consider the paths between a pair of nodes as disjoint paths, it is necessary that all but the first and last hops of those paths are distinct. AOMDV augments the AODV route discovery procedure in two ways:

1. By exploiting the routing information obtained via duplicate route request copies, alternate loop-free reverse paths are formed at the intermediate and the destination nodes.

2. The destination node generates multiple route replies, that travel along multiple loop-free reverse paths to the source established during the route request phase to get multiple loop-free forward paths to the destination.

As in AODV, AOMDV uses destination sequence numbers to ensure loop-freedom. Every node maintains one or more paths to a destination corresponding to the highest sequence number for that destination. Route maintenance in AOMDV is similar to that in AODV [117]. The difference is that, in AOMDV, a node only generates or forwards a RERR packet for a destination when all paths to the destination break.

3.6.2 Ad hoc On-demand Distance Vector Multipath (AODVM)

Ad hoc On-demand Distance Vector Multipath (AODVM) [125][126] is a modification to the AODV routing protocol that discovers multiple node-disjoint paths from a source to a destination. Instead of discarding the duplicate Route Request (RREQ) packets, intermediate nodes store the information included in these packets in a table called RREQ table (Figure 3.5).

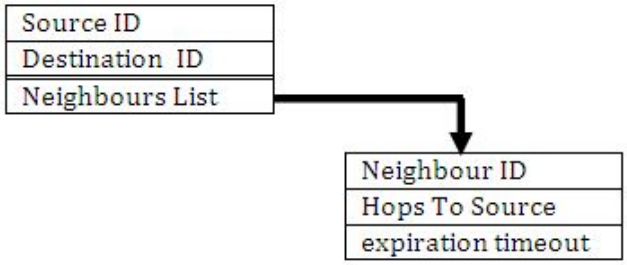

Figure 3.5: The Structure of each RREQ table entry in AODVM

When an intermediate node receives a RREQ packet, it records the following information in its RREQ table: Id of the source node that generated the RREQ, Id of the destination node for which the RREQ is intended, the Id of the neighbour node that the RREQ is received from, and the hop count. Moreover, the intermediate relay nodes are prohibited from sending a RREP packet directly to the source node.

When a destination node receives the first RREQ packet, it updates its sequence number and generates a Route Reply (RREP) packet, which contains an additional field called "Route_ID". The destination assigns a unique Route_ID for each path discovered during a single route discovery instance. The RREP packet is unicast to the source via the neighbour that forwards the RREQ packet. If the destination receives duplicate copies of the RREQ from other neighbours, it updates its sequence number, generates and unicasts RREP packets to the source, after assigning a unique Route_ID for each of them.

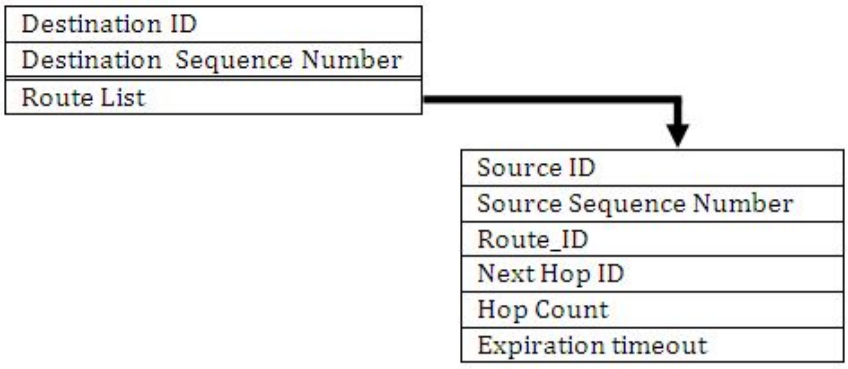

Figure 3.6: The Structure of each Routing table entry in AODVM

Once an intermediate node receives a RREP packet from a neighbour, it updates its RREQ table by deleting the entry corresponding to this neighbour, and adds a routing entry to its routing table as a route to the destination (Figure 3.6). Next, the node determines the neighbour in the RREQ table via the shortest path to the source, and forwards the RREP packet to that neighbour. Then, the entry that corresponds to

this neighbour is deleted from the RREQ table. If an intermediate node is unable to forward a received RREP packet (No entries in its RREQ table), it generates a Route Discovery ERror (RDER) message and unicasts it to the neighbour that the RREP is received from. Upon receiving a RDER packet, the neighbour will try to forward the RREP to a different neighbour which may forward it further towards the source node.

To ensure that a node does not participate in more than one path, when a node overhears any node broadcasting a RREP packet, it deletes the entry that belongs to that node from its RREQ table.
When the source node receives a RREP packet, it should confirm each received RREP packet by means of a Route Reply Confirmation packet (RRCM), which can be piggybacked on the first data packet transmitted on the corresponding route.

In AODVM, the sequence number is used to prevent loops. When a source node initiates a RREQ, it increases its sequence number and the destination's sequence number. Both sequence numbers are included in the RREQ packet. When the destination receives a new RREQ packet, it computes a new sequence number and includes it in the RREP packet.

3.6.3 AODV-BR: Backup Routing in Ad hoc Networks

AODV-BR (AODV with Backup Routes) [58] is an AODV-based protocol. It creates a mesh and provides multiple alternate routes for each desired destination, without transmitting extra control messages. AODV-BR has two phases: Route Construction, and Route Maintenance and Mesh Routes.

Route Construction: As mentioned, AODV-BR is based on the AODV routing protocol. AODV-BR builds routes on demand via a query and reply procedure. The protocol uses the AODV's RREQ (Route Request) process with no modification. The mesh structure and alternate paths are established during the route reply phase. Thus, a slight modification has been made to the route reply process.

Due to the broadcast nature of wireless communications, a node can overhear packets that are transmitted by its neighbours. From these packets a node obtains alternate path information and becomes part of the mesh. When a node that is not part of the primary route overhears a RREP packet transmitted by a neighbour that is not directed to itself, it records that neighbour as a next hop to the destination in its alternate route table. If a node overhears a number of RREP packets for the same route, it chooses the best route and inserts it into the alternate route table.

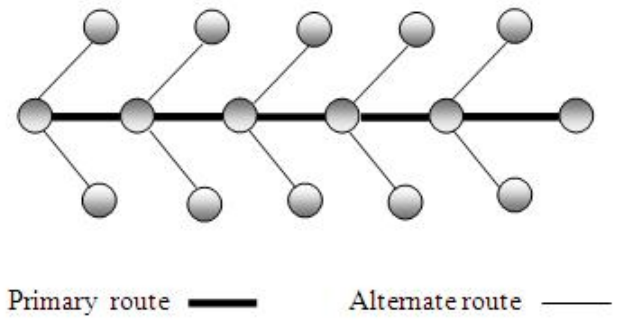

Figure 3.7: Multiple Routes Forming a Fish Bone Structure

The primary route between the source and the destination is established, when the RREP packet reaches the source. Any node that has an entry to the destination in its alternate route table is a part of the mesh. The primary route and alternate routes establish the mesh structure as shown in Figure 3.7. The mesh structure that is established by the primary route and the alternate routes is similar to a fish bone structure.

Route Maintenance and Mesh Routes: Nodes use the primary route to deliver data packets unless a link failure is encountered. When a node detects a link break, it performs a one hop data broadcast to its current neighbours. The node identifies in the data header that the link is disconnected and that the packet is a candidate for alternate routing. When a neighbour receives this packet and has an entry belonging to the destination in its alternate route table, it unicasts the packet to its next hop node. Thus, data packets are delivered using one or more alternate routes.

To prevent packet looping, the mesh nodes forward the data packet only if the packet is not received from their next hop node to the destination, and is not a duplicate.The node that detects the link failure sends a Route Error (RERR) packet to the source node to initiate new route discovery.

Alternate route utilization of AODV-BR is similar to the mechanism of DSR with some differences. AODV-BR uses the mesh link only to go around the broken part of the route, whereas in DSR the node that detects the link failure salvages the data by replacing the entire remaining route with an alternate route stored in its route cache. Another difference is that in DSR, the node sends a RERR packet only when it has no alternate routes. Thus, routes in DSR are less fresh compared to AODV-BR.

3.6.4 Split Multipath Routing with Maximally Disjoint Paths in Ad hoc Networks (SMR)

Split Multipath Routing protocol (SMR) [59] is an on demand routing scheme that builds and utilizes maximally disjoint paths. The protocol uses a per-packet allocation scheme for distributing data packets into multiple paths. SMR splits the data traffic into multiple routes to prevent nodes from being congested and to use network resources efficiently. To achieve this goal the destination node should know the full path of all available routes. SMR uses the source routing approach, where the RREQ packet includes information on all nodes that constitute the route. Moreover, intermediate nodes do not send RREP packets back to the source node even if they have route information regarding the destination. SMR uses two procedures: Route Discovery and Route Maintenance.

Route Discovery: SMR builds multiple routes using request/reply cycles. When a source node has data ready to send and does not have a route to the destination, it broadcasts a RREQ packet which contains the source node ID and a unique sequence number to identify the packet. If an intermediate node receives copies of a RREQ, it

forwards duplicate packets through a different incoming link than the link from which the first RREQ is received, and whose hop count is not larger than that of the first received RREQ. When the destination node receives the RREQs, it selects two routes that are maximally disjoint. The first route is the shortest delay route, which is taken by the first RREQ that the destination receives. When receiving the first RREQ, the destination records the entire path and unicasts a RREP packet to the source through this route. Next the destination waits for a period of time to receive more RREQs and learn more possible routes. It then selects the route that is maximally disjoint to the route that it has already replied along.

Route Maintenance: When a node fails to deliver a data packet to the next hop of the route, it considers the link as a broken link. The node responds by unicasting a Route Error (RERR) packet in the upstream direction of the route. The RERR packet contains the route to the source, and the immediate upstream and downstream nodes of the broken link. When the source node receives an RERR packet, it removes every entry in its route table that uses the broken link. If a source node is informed of a broken link and the session is still active, it uses the remaining valid route to deliver data packets. But, if both routes of the session are broken, the source node initiates the route recovery process.

3.6.5 Stable Node-Disjoint Multipath Routing with Low Overhead (NDMR)

The Node-Disjoint Multipath Routing Protocol (NDMR) [63] is a modification to the AODV routing protocol by including path accumulation in RREQ packets. The main goal of NDMR is to create multiple node-disjoint paths with a low routing overhead. To achieve this goal, the destination node should know the entire routing path list of all routes. Thus, the destination can confirm if the path is node-disjoint or not.

Like AODV, NDMR has two mechanisms: Route discovery and Route maintenance.

- *Route Discovery mechanism:* This mechanism depends on three features which

are: Path accumulation, choosing and recording the shortest routing hops of loop-free paths, and selecting Node-Disjoint paths. When a node generates or forwards a RREQ packet, it appends its own address to the packet. When the packet reaches its destination, the destination is responsible for deciding whether the routing path is a node-disjoint one or not. After confirming a node-disjoint path, the destination generates and unicasts a Route Reply (RREP) to the source node that generated the RREQ packet via the reverse path of the node-disjoint route. The RREP packet contains the all nodes of the whole route path. Upon receiving a RREP packet by an intermediate node, it updates its routing table entry and its reverse routing table entry using the nodes list included in the RREP packet. Figure 3.8 shows the Route Request Process of NDMR.

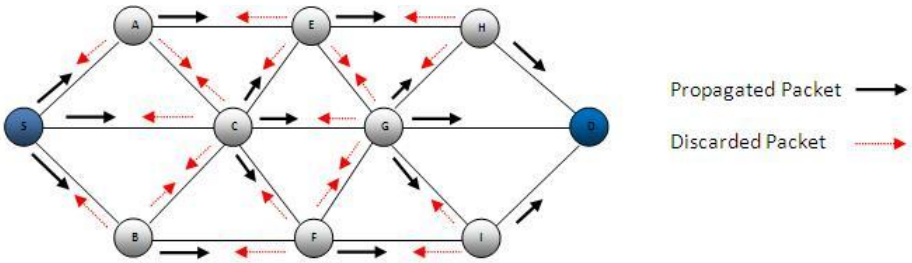

Figure 3.8: Route Request Process with Low Overhead

NDMR uses an approach that records the shortest routing hops of loop-free paths. If a node receives a RREQ packet for the first time, it checks the path accumulation list from the packet and determines the number of hops from itself to the source node. Next, it records the number as the shortest number of hops in its reverse route table entry. If the node receives a copy of the same RREQ again, it calculates the number of hops and compares it with the number recorded in the reverse route table entry. If the number of hops is greater than the shortest number of hops in the reverse route table entry, the RREQ packet

is dropped. Otherwise, the node appends its address to the route path list of the RREQ and broadcasts it to its current neighbours. This approach guarantees loop-free paths and decreases the routing overhead.

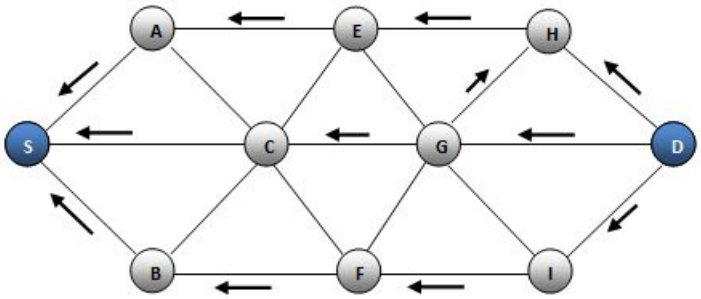

Figure 3.9: Node-Disjoint Paths

As mentioned, the destination node is responsible for selecting multiple node-disjoint paths. When a destination node receives the first RREQ, it records the list of node IDs for the entire path in its reverse route table, and unicasts a Route Replay (RREP) packet, which includes the path towards the source via the reverse route. When the destination receives a duplicate RREQ, it compares the whole route path in the RREQ with all the existing node-disjoint paths stored in its reverse route table. If there is no common node between the path in the RREQ packet and all the paths in the reverse route table (except source and destination nodes), the path in the RREQ is accepted and recorded in the reverse route table. Otherwise, the current received RREQ is discarded.

When an intermediate node receives a RREP packet, it records a a forward path to the destination in its route table, and records a reverse path to the source in its reverse route table. The forward path is through the neighbour from which the RREP arrived, and reverse path is through the next hop to the source. Finally, the intermediate node forwards the RREP towards the source node along the reverse route path.

When the source node receives the RREP packet, it establishes a route by recording the next hop to the destination into its multiple route forward path entry. Figures 3.8 and 3.9 illustrate the idea behind building multiple Node-Disjoint paths.

In NDMR, each node is dependent on sending HELLO messages to inform its neighbours about its existence. If a node does not receive such a message from a neighbour for a certain time, the node considers the link between itself and that neighbour is broken. A Route Error (RERR) packet is propagated from the upstream node of the link failure to the source node. When an intermediate node receives an RERR packet, it invalidates the route to the destination, and propagates the RERR packet to its precursor node along the reverse route path. As the source node receives the RERR packet, it invalidates the route path to destination and selects a valid node-disjoint routing path as the active routing path from the routing table to continue sending data packets on.

3.6.6 Scalable Multipath On-demand Routing for Mobile ad hoc Networks (SMORT)

Scalable multipath on-demand routing protocol (SMORT) [100] is a multipath extension to the AODV routing protocol. The main objective of SMORT is to reduce the amount of routing overhead using multipath routing. Reduction in the control overhead allows the protocol to scale to larger networks. SMORT uses the idea of *Fail-safe* multiple paths. The path between the source and the destination is considered as a Fail-safe to the primary path, if it bypasses one or more intermediate nodes on the primary path.

Multiple paths between a source and a destination nodes can be divided into two types, namely node-disjoint and link-disjoint multiple paths. Node-disjoint paths are the paths that do not have any common nodes, except the source and destination nodes. In contrast, Link-disjoint paths do not have common links, but may have

common nodes. Fail-safe multiple paths are different from node-disjoint and link-disjoint multiple paths, where the Fail-safe multiple paths can have nodes and links in common. The Fail-safe path is used to send data packets when the bypassed node(s) on the primary path leave the network or move away. Figure 3.10 shows a set of Fail-safe paths which together bypass all nodes on the primary path. For example, the paths S-A-H-C-E-D and S-A-B-C-L-D are Fail-safe paths to the primary path S-A-B-C-E-D (Figure 3.10). Even if node B and node E move a way, the session between node S and node D is unaltered because the packets can be redirected to the Fail-safe paths.

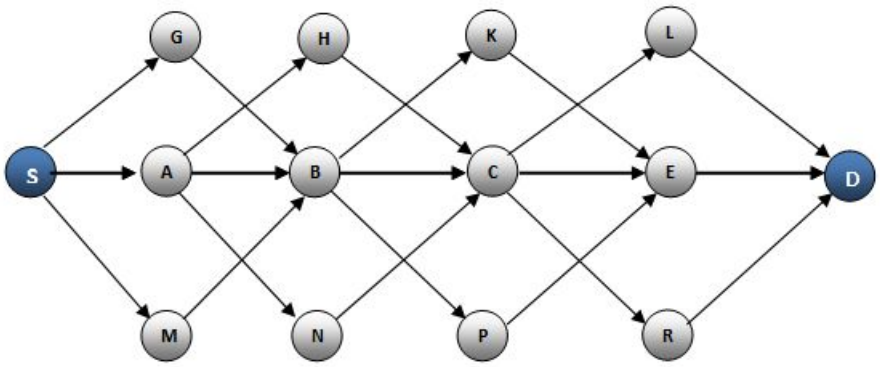

Figure 3.10: Fail-safe multiple paths

As SMORT is a reactive routing protocol based on AODV, it has three basic phases: Route Discovery, Route Reply, and Route Maintenance. To enable computation of multiple Fail-safe paths, SMORT allows all nodes to accept multiple copies of Route Request (RREQ) packets, and the destination node replies to multiple copies of RREQ.

When a source node needs to communicate with a destination for which it does not have a route, it initiates and broadcasts a RREQ packet containing the address of the destination. An intermediate node, upon receiving a RREQ packet, responds by

sending a Route Reply (RREP) packet if it has a route to the destination. Otherwise, it re-broadcasts the RREQ. Nodes re-broadcast only the first copy of the RREQ, and store the information of all RREQ copies in a *request-rcvd* table. Each entry in the *request-rcvd* table contains the address of the previous node that the RREQ is received from (Last-Hop), and the number of hops (distance between the node and the source).

If an intermediate node or a destination needs to send a RREP packet, it should follow the reverse paths stored in the request-rcvd table to reach the source node. Compared to the RREP packet of AODV, the RREP packet of SMORT contains three extra fields (shown in bold letters in Figure 3.11), to eliminate routing loops, and to compute Fail-safe multiple paths. The *Node list* field consist of a list of all nodes that the route reply has passed so far. The *Reply generator* field is used to store the address of the packet generator. The *Multiple reply* field is a Boolean variable to distinguish the first RREP packet. Nodes accept only the first received RREP, and store the route information carried in it in its routing table.

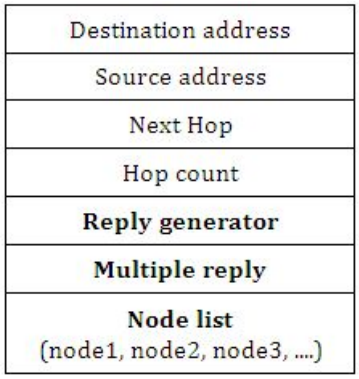

Figure 3.11: Route Reply packet structure of SMORT

Instead of using RREQ packets to avoid loops in the routes, RREP packets are used where they carry the full path to the destination. This is because the number of RREP packets communicated are limited compared to RREQ packet transmissions. Intermediate nodes may receive multiple RREP packets, but they relay only the first one. In order to limit the multipath computation overhead, each node on the primary path accepts at most two secondary paths.

3.6.7 Disjoint Multi-Path Source Routing in Ad Hoc Networks: Transport Capacity (DMPSR))

The Disjoint Multi-Path Source Routing (DMPSR) [124] is a protocol that allows packets originating from the same source to be statistically multiplexed onto multiple disjoint routes. DMPSR consists of three phases: Route Discovery, Route Maintenance, and Route Destruction. When a source node needs to start the communication, it initiates the Route Discovery process, by broadcasting a Route Request (RREQ) packet. To minimize the routing overhead, the source node broadcasts the RREQ packet with probability p = 1, while the other nodes broadcast the packet with probability $p < 1$. This probability is referred to as the critical probability below which the network lacks connectivity (i.e., the network is in sub-critical mode).

When an intermediate node that knows how to reach the destination or the destination itself receives a RREQ packet, it generates and sends a Route Reply (RREP) packet back to the source node. The source node gathers information from all RREP packets and selects as many disjoint routes as possible. The purpose of choosing multiple routes is to increase the connectivity of the network (i.e., the source stays connected to the destination for a longer time).

When a link failure occurs, the source node continues sending packets over alternative routes and only reinitiates the Route Discovery process if all the routes are invalid. In the sub-critical mode, the method increases the chance of delivering the message to the destination, because DMPSR is designed to utilize all possible routes simultaneously.

At the end of the communication session, the source node informs the destination node and all the relay nodes about closing the connection to release the resources. The informed nodes either choose to erase the route information from their caches or wait for a timer to expire before doing so. The authors in [124] present an analytical framework to derive the transport capacity of the network with DMPSR both with

and without load balancing. They concluded that if no load balancing is used, the DMPSR's transport capacity is more than that of traditional source routing, where the spatial density of the network is below some critical threshold.

3.6.8 Node-Disjoint Multipath Routing with Zoning Method in MANETs

Wang et al. propose Multiple Zones-based routing protocols (M-Zone) [35], to discover node-disjoint paths in large scale MANETs. M-Zone uses a multiple zoning method based on location to guarantee that the paths between the source and the destination have no common nodes. M-Zone combines the advantages of topology-based routing and location-based routing and can be used in large scale MANETs using segment-by-segment route discovery. The proposed protocol divides the region between the source and the destination into multiple zones to find node-disjoint multiple paths, and uses two approaches to maintain the routes: *local route maintenance* and *global route maintenance*. The local route maintenance ensures that the broken path is repaired quickly, and global route maintenance initializes route discovery periodically. Compared to GZRP [14], the authors conclude that the average path length of M-Zone is close to that of GZRP, and the average packet delivery ratio is significantly improved.

3.6.9 Summary of Multipath Routing Protocols

Multiple paths can be used to provide load balancing, fault tolerance, and bandwidth aggregation. Load balancing can alleviate congestion and bottlenecks. It can be achieved by disseminating the traffic through multiple routes. From a fault tolerance perspective, multipath routing reduces the probability that communication is disrupted when a link failure occurs. Moreover, if congestion occurs, multipath routing protocols can divert traffic through alternate paths to alleviate the burden of the congested link. When data is split into multiple streams and routed simultaneously

through multiple paths, the aggregate bandwidth of the paths may satisfy the bandwidth required for the application.

3.7 Summary

Several routing protocols for wireless ad hoc networks have been presented in this chapter. In this section, we present a summary the most routing protocols introduced in this chapter, and list the differences between them in two tables according to different criteria.

AODV is one of the most popular and widely researched on-demand ad hoc routing protocols. One advantage of AODV is that less memory space is required as only information on active routes is maintained, in turn increasing its performance. AODV is also adaptable to highly small dynamic networks. However, a node may experience large delays during route construction, and link failure may initiate more route discovery, which introduces extra delays and consumes more bandwidth as the size of the network increases. Moreover, the protocol is not very scalable.
The DSDV routing protocol is a basis for several protocols such as AODV. It guarantees loop free paths and reduces the Count to infinity problem. DSDV is well suited to small ad hoc networks where changes in the topology are limited. The DSDV protocol overhead is directly proportional to the number of nodes in the network. Therefore the protocol will not scale well in large networks since a large portion of the network bandwidth would then be used in the updating procedures. DSR is a source-routed on-demand protocol. An advantage of DSR is that nodes can store multiple routes in their route cache, which means that there is no need to initiate route discovery if a valid route is available there. This is beneficial in networks with low mobility, since the routes stored in the route cache will be valid longer. Another advantage of DSR is that it does not require periodic routing packets, therefore nodes can enter sleep mode to conserve their power. This also saves a considerable amount of bandwidth in

the network. Since DSR discovers routes on-demand, it may have poor performance in terms of control overhead in networks with high mobility and heavy traffic loads. DSR has a high delay especially for networks with large traffic loads. The main reason for this is the lack of a mechanism that can expire unused routes from caches, together with the aggressive use of caching.

The advantage of TORA is that it has reduced the scope of control messages to a set of neighbouring nodes, where a topology change has occurred. TORA can be used in conjunction with a lightweight adaptive multicast algorithm (LAM) to provide multicasting. The disadvantage of TORA is that the algorithm may also produce temporarily invalid routes.

ZRP is based on the notion of routing zones that are defined for each node. An advantage of this protocol is that it has significantly reduced the amount of communication overhead when compared to pure proactive protocols. It has also reduced the delays associated with pure reactive protocols such as DSR, by allowing routes to be discovered faster. This is because, to determine a route to a node outside the routing zone, the routing only has to travel to a node which lies on the boundaries of the desired destination. A disadvantage of ZRP is that for large routing zones the protocol behaves like a pure proactive protocol, while for small ones it behaves like a reactive protocol.

DDR is a tree-based routing protocol. An advantage of DDR is that it does not rely on a static zone map to perform routing and it does not require a root node or a clusterhead to coordinate data and control packet transmission among different nodes and zones. However, the nodes that have been selected as preferred neighbours may become performance bottlenecks. This is because, they may transmit more routing and data packets than every other node. This means that these nodes would require more recharging as they will have less sleep time than other nodes. Furthermore, if a node is a preferred neighbour for many of its neighbours, many nodes may need to communicate with it. This means that channel contention would increase around the preferred neighbour, which could result in larger delays experienced by all neighbouring nodes before they can reserve the medium. In networks with high traffic, this

may also result in significant reduction in throughput, due to packets being dropped when buffers become full.

In DST the nodes in the network are grouped into a number of trees. Each tree has two types of nodes: route nodes and internal nodes. A major disadvantage of the DST algorithm is that it relies on a root node to configure the tree, which creates a single point of failure. Furthermore, the holding time used to buffer the packets may introduce extra delays in the network.

ZHLS is a hybrid routing protocol. It is highly adaptable to dynamic topologies and it generates far less overhead than pure reactive protocols, which means that it may scale well to large networks. A disadvantage of ZHLS is that all nodes must have a preprogrammed static zone map in order to function. This may not feasible in applications where the geographical boundary of the network is dynamic.

In WRP, each node maintains four routing tables. This introduces a significant amount of memory overhead at each node as the size of the network increases. Another disadvantage of WRP is that it ensures connectivity through the use of hello messages, which are exchanged among neighbouring nodes whenever there is no recent packet transmission. This consumes a significant amount of bandwidth and power as each node is required to stay active at all times (i.e., they cannot enter sleep mode to conserve their power).

In GSR, each node maintains a link state table based on the up-to-date information received from neighbouring nodes, and periodically exchanges its link state information only with neighbouring nodes. This significantly reduces the number of control message transmitted through the network. However, the size of update messages is relatively large, and as the size of the network grows they get even larger. Therefore, a considerable amount of bandwidth is consumed by these update messages.

FSR reduces the size of the update messages by updating the network information for nearby nodes at a higher frequency than for the remote nodes, which lie outside the fisheye scope. This makes FSR more scalable to large networks than other protocols. However, scalability comes at the price of reduced accuracy. This is because as mobility increases the routes to remote destination become less accurate. This can be

overcome by making the frequency at which updates are sent to remote destinations proportional to the level of mobility. However it would increase its use of bandwidth. In DREAM, each node knows its geographical coordinates using a GPS sensor. These coordinates are periodically exchanged among nodes which enables each node to obtain location information about other nodes in the network. The coordinates are stored in a routing table called a *location table*. The advantage of exchanging location information is that it consumes significantly less bandwidth than exchanging complete link state or distance vector information, which means that it is more scalable. In DREAM, routing overhead is further reduced, by making the frequency with which update messages are disseminated proportional to mobility and the distance effect. This means that stationary nodes do not need to send any update messages.

CGSR is a hierarchical routing protocol where the nodes are grouped into clusters. An advantage of this protocol is that it can reduce the routing table size by storing only one entry for all nodes in the same cluster. Thus, the broadcast packet size of the routing table is reduced. A disadvantage of CGSR is the difficulty of maintaining the cluster structure in a mobile environment.

OLSR is a proactive link-state routing protocol and does not need a central administrative system to handle its routing process. One of its advantages is that it immediately knows the status of the link, so that nodes know the quality of the route. OLSR is more efficient in networks with high density and highly sporadic traffic. However, a drawback of the OLSR protocol is that it makes each node periodically broadcast updated topology information throughout the entire network, which increases the protocol's bandwidth usage. But the flooding is minimised by the MPRs, which are the only nodes that are allowed to forward the topological messages. When the number of nodes increases, the overhead from control message traffic also increases. So, the scalability is constrained.

TBRPF is a link-state based routing protocol, which performs hop-by-hop routing. The protocol uses the reverse-path forwarding (RPF) to disseminate its update packets in the reverse direction along the spanning tree. In TBRPF, each node reduces that overhead by reporting only part of its source tree to its neighbours. The reportable

part of each source tree is exchanged with neighbouring nodes by periodic and differential hello messages. Differential hello messages only report changes in the status the neighbouring nodes. As a result, hello messages in TBRPF are smaller than in protocols which report the complete link-state information.
The routing protocols that are based on the source routing protocol (DSR) such as SMR and NDMR cannot scale to large networks because source routing requires every data packet to carry the full path to its destination. In large networks, the size of data packets become prohibitively high due to the long paths that they carry.

Table 3.6 and Table 3.7 present some properties of the protocols that are discussed in this chapter. As many routing protocols use distance vector or link state as their underlying mechanism to transmit update packets and compute routes, we consider Link State Routing (LSR) and Distance Vector Routing (DVR) as the basis of this comparison. The meanings of some items in the tables are presented below.
Route Computation indicates when the route is computed (precomputed, on-demand, or hybrid). The computation in proactive protocols may be done by the nodes themselves or collaboratively. However, in reactive protocols, the computation is done by broadcasting a QUERY message that propagates through the entire network to discover a route. *Stored Information* is information stored in each node. *Update Period* is applicable to proactive protocols and assumes values such as “periodical”, “event-driven” or “hybrid”. For reactive protocols, when a link failure occurs, route maintenance is activated. This is called event-driven route maintenance. *Update Information* denotes the type of information that is included in the update and the *Update Destination* indicates the nodes that receive the information. Generally, the *Update Information* is the link state and *Update Destination* is the “neighbours”. However, for event-driven route maintenance, the *Update Information* is generally by an “ERROR” message and the *Update Destination* is the source node. Finally, the *Update Method* indicates how the information is disseminated (broadcasting, unicasting, etc.)
Note: *BR* stands for **B**eacon **R**equirement.(or Hello Message Requirement)

Protocols	Route Computation	Structure	Number of Routes	Source Routing	Route Reconfiguration Methodology	BR*
LSR	Proactive/itself	Flat	Single or multiple	No, may Yes	N/A	No
DVR	Proactive/distributed	Flat	Single	No	N/A	No
AODV	Reactive/broadcast QUERY	Flat	Multiple	No	Erase route, Notify source	Yes
AODV-BR	Reactive/broadcast QUERY	mesh	Multiple	No	Local repair, Notify source & neighbours	Yes
AODVM	Reactive/broadcast QUERY	Flat	Multiple	No	Erase route, Notify source	yes
AOMDV	Reactive/broadcast QUERY	Flat	Multiple	No	Erase route, Notify source	yes
CGSR	Proactive/distributed	Hierarchy	Single	No	N/A	No
DDR	Proactive(intra)/Reactive(inter)	Hierarchy	Multiple	No	Notify source to initiate route discovery	yes
DMPSR	Reactive/broadcast QUERY	Flat	Multiple	yes	Erase route, Notify source	No
DREAM	Proactive/distributed	Flat	Multiple	No	N/A	No
DSDV	Proactive/distributed	Flat	Single	No	N/A	No
DSR	Reactive/broadcast QUERY	Flat	Multiple	Yes	Erase route, Notify source	No
DST	Reactive/broadcast QUERY	Hierarchy	single but may multiple	No, may yes	Route repair	No
FSR	Proactive/distributed	Hierarchy	Single or multiple	No, may Yes	N/A	No
GSR	Proactive/distributed	Flat	Single or multiple	No, may Yes	N/A	No
MDSDV	Proactive/distributed	Flat	Multiple	No	Local repair, Notify source & neighbours	Yes
NDMR	Reactive/broadcast QUERY	Flat	Multiple	Yes	Erase route, Notify source	Yes
OLSR	Proactive/distributed	Hierarchy	Single	No	N/A	Yes
SHARP	Proactive/Reactive (zone radius)	Flat	Single	No	Link reversal, Route repair	Yes
SMORT	Reactive/broadcast QUERY	Flat	Multiple	No	Replace primary route with secondary route	No
SMR	Reactive/broadcast QUERY	Flat	Multiple	yes	Erase route, Notify source	No
TBRPF	Proactive/distributed	Flat	single but may multiple	No	Select a new parent, Notify source	Yes
TORA	Reactive/broadcast QUERY	Flat	Multiple (DAG)	No	Link reversal, Route repair	No
WRP	Proactive/distributed	Flat	Single	No	N/A	Yes
ZHLS	Proactive(intra)/Reactive(inter)	Hierarchy	Single	No	Location request	No
ZRP	Proactive(intra)/Reactive(inter)	Flat	Single or multiple	Yes for interzone	Route repair	No

Table 3.6: Comparison of MANET Routing Protocols

Protocol	Stored Information	Update Period	Update Information	Update Destination	Method
LSR	Entire topology	Hybrid	Neighbours' link state	All nodes	Flooding
DVR	Distance-vector	Periodical	Distance vector	Neighbours	Broadcast
AODV	Next hop for desired destination	. Event-driven	Route Error packet	Source	Unicast
AODV-BR	Next hop, number of hops, destination	Event-driven	Route Error packet	Source	Unicast
AODVM	Source Id, Next hop, last hop, hop count	Event-driven	Route Discovery Error message	Source	Unicast
AOMDV	Next hop, last hop, hop count for desired dest	Event-driven	Route Error packet	Source	Unicast
CGSR	Cluster member table, Distance Vector	Periodical	Clus. member tab., Distance Vec.	Neigh.&Clus. head	Broadcast
DDR	Preferred neighb., neighboring zones inform.	Periodical	Id numbers & degree of neighb.	Neighbours	Broadcast
DMPSR	Full path (From source to Destination)	Event-driven	Route Error packet	Source	Unicast
DREAM	Location information of all other nodes	Periodical	geographical coordinates	All nodes	Broadcast
DSDV	Distance vector	Hybrid	Distance vector	Neighbours	Broadcast
DSR	Full path (From source to Destination)	Event-driven	Route Error packet	Source	Unicast
DST	Distance/routing/query tables	Event-driven	Routing table	Neighbours	Broadcast
FSR	Entire topology	Periodicals(dif. freq.)	Link state of fisheye scope	Neighbours	Broadcast
GSR	Entire topology	Periodical	All nodes link state	Neighbours	Broadcast
MDSDV	First and second hop, number of hops, destination	Period./Event-driven	Distance vector	Neighbours	Broadcast
NDMR	Full path (From source to Destination)	Event-driven	Route Error packet	Source	Unicast
OLSR	Neighbour, Topology, Routing tables	Periodical	partial link state information	All nodes	Flooding
SHARP	The height information of all neighbours	Periodical	Neighbours' heights	Neighbours	Broadcast
SMORT	Next hop, number of hops, life time, full path	Event-driven	Route Error packet	all source nodes	Unicast
SMR	Full path (From source to Destination)	Event-driven	Route Error packet	Source	Unicast
TBRPF	Neighbour/Topology/Routing table	Event-driven	Link state	Neighbours	Broadcast
TORA	The height information of all neighbours	Event-driven	Node's height	Neighbours	Broadcast
WRP	Distance/routing/link-cost tables, MRL	Hybrid	Distance Vector, List of responses	Neighbours	Broadcast
ZHLS	Local/zone topology	Period./Event-driven	Node/Zone link state	Zone/all nodes	Broadcast
ZRP	Local (within zone), topology	Periodical	Link state of nodes in the zone	Neighbours	Broadcast

Table 3.7: Comparison of MANET Routing Protocols (cont.)

Chapter 4

Preliminary Design of MDSDV (MDSDV0)

This chapter describes the preliminary design of the MDSDV routing protocol (MDSDV0). Section 4.1 outlines the design goals. Section 4.3 gives an overview of the protocol. Section 4.4 presents the pseudocode with an example to illustrate the MDSDV0's phases. Section 4.5 explains how MDSDV0 builds node disjoint paths, and finally Section 4.6 presents the limitations of the preliminary design of MDSDV.

4.1 Introduction

To date, the majority of multipath ad hoc routing protocols are based on an ***on demand*** model, for example [59][58][63][77][121][125] (Section 3.6). So in this thesis we present a new proactive multipath routing protocol to investigate the strengths and weaknesses of this type of protocol.

Providing multiple routes helps to keep the route recovery process short and control packet overhead. We believe utilizing multiple routes is beneficial in network

communications, particularly in mobile wireless networks where routes are disconnected frequently because of mobility and poor wireless link quality. In this chapter, we develop a new table-driven multipath distance vector protocol for mobile ad-hoc networks. Specifically, we present multipath extensions to a well known single path routing protocol known as Destination Sequenced Distance Vector (DSDV). The resulting protocol is referred to as Multipath Destination Sequenced Distance Vector (MDSDV) which guarantees loop freedom and disjointness of alternative paths. MDSDV extends the DSDV protocol to store multiple node-disjoint paths for each destination in the network.

Two new fields called *second hop* and *link_id* are added to the routing table. The link_id is a unique number that is generated by the destination node. Both the second hop and link_id are used to insure these paths are disjoint from any source to any destination. MDSDV employs a unique method for creating routing tables containing the best and disjoint paths to every destination.

Note: In this thesis we shall term the best path to be the shortest path (the one with least number of hops). If two paths have the same number of hops, the one with the highest sequence number is the best.

4.2 NS2 extensions to support MDSDV

This section gives details on how the network simulator NS2 (Figure 4.1) has been extended to support MDSDV. The diagram in Figure 4.2 shows the modules that are used to support MDSDV.

The diagram in Figure 4.2 is divided into three parts (A, B, and C). Part ***A*** represents the unmodified modules, part ***B*** represents the modified modules, and Part ***C*** represents the added modules. The model code is presented in Appendix E. It is also available at: http://www.macs.hw.ac.uk/∼etorban/.

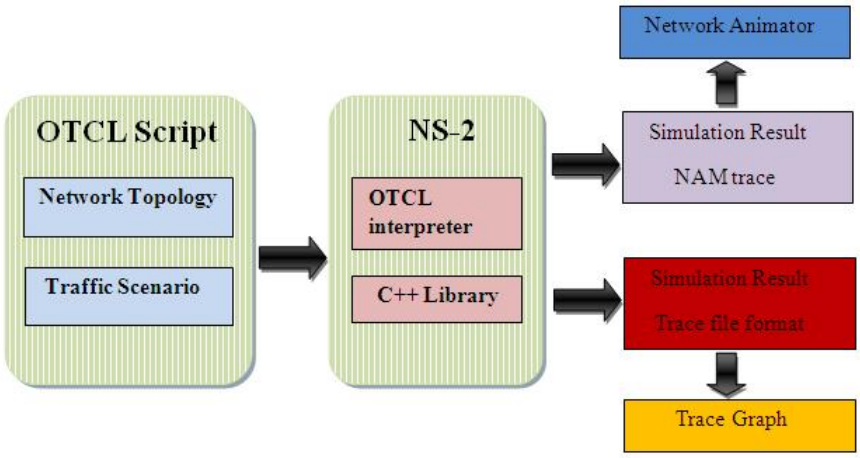

Figure 4.1: The Overall architecture of NS2

MDSDV routing protocol is implemented using C++ under NS2, and the simulations scenarios are described using Tcl scripts. We created a new directory called mdsdv to allocate our code inside the NS2 base directory (NS-2.30). Then, we created the protocol "physical" structure by creating the following files.

mdsdv/mdsdv.h This is the header file where we define all necessary timers and routing agents which perform protocol's functionality.

mdsdv/mdsdv.cc In this file, all timers, routing agent and Tcl hooks are actually implemented.

mdsdv/rtable.h This is the header file where the Routing Table, Neighbours Table, and Queue Table are declared.

mdsdv/rtable.cc Implementation of the Routing Table, Neighbours Table, and Queue Table.

Secondly, we created the protocol logical structure (classes), by creating an agent which is inherited from Agent class. Agent represents the endpoints where the networks-layer packets are created and consumed, and is used in the implementation of the protocol at various layers. Agent is the main class to implement the protocol. In addition,

this class offers a linkage with the Tcl interface to control the protocol through Tcl scripts.

The routing table is a collection of entries or routes gathered by a node to the destinations in the network. The routing agent maintains a routing table and an internal state, which can be represented as an attributes collection or a new class inside the routing agent itself. We utilise the routing table as a new class.

MDSDV defines new control packets which represents the format of its control packets. These packets are defined in the *common/packet.h* header file. When the protocol needs to send packets periodically or after some time from the occurrence of an event, it is useful to count on a Timer class. Control packets of the preliminary version (MDSDV0) and the final version (MDSDV) are presented in subsection 4.3.2 and section 5.3) respectively.

The other important class is the Trace class, which is the base for writing log files with information about what happened during the simulation.

Another file that should be modified is the *tcl/lib/ns-lib.tcl* file. Whenever we export a C++ variable, it is recommended that we set the default value for that variable in the "tcl/lib/ns-lib.tcl" file. Otherwise we will get a warning message when we create an instance of our new object. When everything is implemented, we only need to compile it. To do so we modify *Makele* file by adding our object files inside OBJ CC variable.

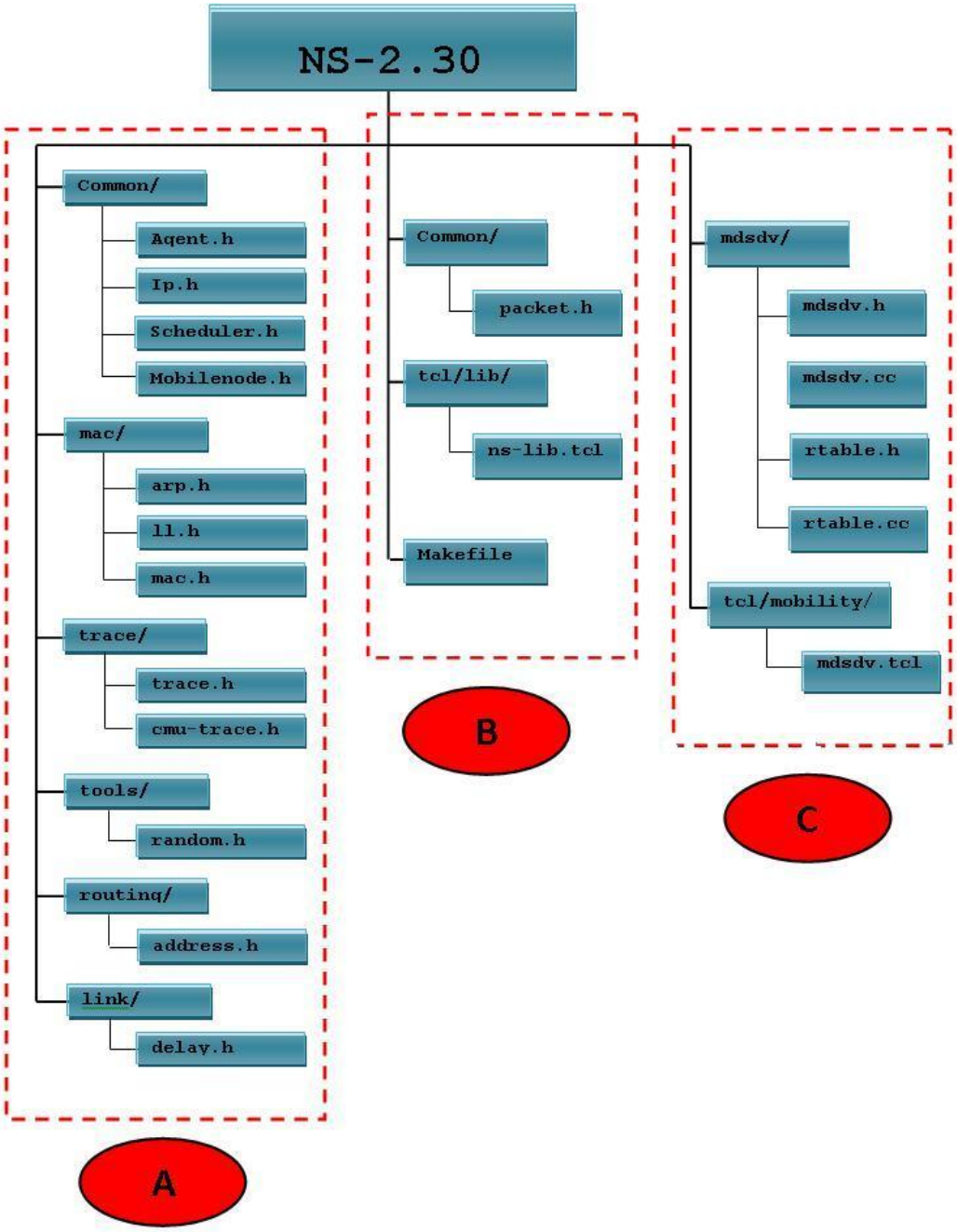

Figure 4.2: MDSDV Modules: Unmodified (A), Modified (B) and New (C)

4.3 MDSDV0 Overview

Since MDSDV0 like DSDV is proactive, it has the same advantages as DSDV where it maintains an up-to-date view of the network, and every node has a readily available route to every destination node in the network. Nodes in MDSDV0 periodically broadcast *Hello Messages* or *Available Messages* (depending on the Neighbours Table (NT)). If the node's NT is empty, the node broadcasts a *Hello Message*, otherwise it broadcasts an *Available Message*. If a new node is detected, it will receive copies of the routing tables of all its neighbours (*Full Dumps*), and perform a filtering operation to initialise its own routing table. After creating its routing table, the new node broadcasts *Update Packets* (the number of Update Packets depends on the number of neighbours) to inform nodes of network topology changes. Failing transmissions cause the transmitter to report the link as a failure in an *Error Packet* which it propagates to all nodes using that link. When an intermediate node fails to forward a data packet, it unicasts a *Failure Packet* to the source node to stop using the link included in the *Failure Packet*.

4.3.1 MDSDV0 Tables

Using MDSDV0, each node maintains two tables: a Neighbours Table (NT) and a Routing Table (RT) which are described below:

- ***Neighbours Table (NT):*** each node in the network maintains a *Neighbours Table* which contains all its neighbours. A node periodically checks its NT to decide whether to broadcast a *Hello Message* or an *Available Message*. If this table is empty, the node considers itself as an isolated node which means that it has to propagate a *Hello Message* (the node will be considered as a new node). Otherwise, it broadcasts an *Available Message*. Also, this table is used when the new node needs to initiate Update Packets. The NT is updated when the node

receives a control packet from a neighbour or when one of its neighbours goes out of its transmission range. Table 4.1 shows the structure of a Neighbours Table entry.

Field name	Description
Neighbour id	Address of the neighbour node
Link_id	An identifier generated by the new node for the newest routes
Sequence Number	The most recent sequence number generated by the neighbour
Flag	This flag is set to 1 when the neighbour is new, otherwise is set to 0

Table 4.1: Neighbours table structure (NT)

- ***Routing Table (RT):*** each node maintains its routing table that lists a number of paths for each destination in the network. The routing table is used to transmit packets through the network. Nodes have to update their routing tables when there is a significant change in the network. The TimeOut field is only used for adjacent nodes, i.e., nodes that are within wireless transmission range. For all other nodes it is simply set to Null. Table 4.2 shows the structure of a routing table entry.

Field name	Description
Destination	Address of the destination node
Next hop	The first hop to the destination
Second hop	The second hop to the destination
Number of hops	Number of hops to the destination
Link_id	An identifier generated by the new node for the newest routes
Sequence number	A sequence number to distinguish stale routes
Time	The time that the path was obtained
TimeOut	The time that the node is considered as a neighbour node

Table 4.2: Routing Table structure (RT) entry

4.3.2 MDSDV0 Control Packets

The Control Packets required to implement the MDSDV0 routing protocol are:

1. *Hello Message:* Periodically, each node checks its NT. If the node's NT is empty (no neighbours), the node increments its sequence number and broadcasts a *Hello Message* that includes the new sequence number. The message is received by neighbours and will not be retransmitted.

2. *Available Message:* If the node has at least one neighbour (by checking the NT), it increments its sequence number and broadcasts an *Available Message* which includes the new sequence number. The message is used to inform adjacent nodes that it is still available in the network.

3. *Full Dump:* When a node receives a *Hello Message*, it responds by unicasting a Full Dump of its routing table to the Hello Message sender. It includes the best route for each destination.

4. *Update Packet:* is propagated by the new node to its neighbours after creating and filtering its routing table. The number of Update Packets depends on the number of neighbours. Table 4.3 shows the structure of the Update Packet.

Sender	First hop	Second hop	Number of hops	Destination	Link_Id	Neighbour	Sequence Number

Table 4.3: The Update Packet structure

5. *Error Packet:* is propagated when a link failure is detected. The node that detects the link failure initiates and propagates this type of packet to its neighbours. The packet is rebroadcast through the entire network. When a node receives an Error Packet and updates its RT by deleting any entry, it should rebroadcast the Error Packet to its neighbours. Table 4.4 shows the structure of the Error Packet.

Sender node	Destination node	Broken Link-Id

Table 4.4: The Error Packet structure

6. *Failure Packet:* The intermediate node should forward the data to the node that its address is included as a second hop in the header of the data packet. If the link between the intermediate node and this node is broken, the intermediate node unicasts a *Failure Packet* to the source node to stop sending data through this broken link. The *Failure Packet* includes the destination, the first hop that the source used to send data, and the broken link id. Any node receives this kind of packet can update its RT by deleting any entry with the same link id that is included in the Failure Packet. Table 4.5 shows the Failure Packet's structure.

Sender	First hop	Failed Link-Id	Destination

Table 4.5: The Failure Packet structure

The difference between the *Hello Message* and the *Available Message* is the following:

- The *Hello Message* is generated and broadcast only when a node has no known neighbours. In contrast, the *Available Message* is broadcast when a node has at least one neighbour.

- The node that receives a *Hello Message* responds by unicasting its routing table (*Full Dump*) to the *Hello Message* sender. In contrast, the node that receives an *Available Message* responds by updating the TimeOut field only.

Also the difference between the *Error Packet* and *Failure Packet* is the following:

- The *Error Packet* is generated and broadcast when a node discovers a broken link (detected by the MAC layer or invalid TimeOut). In contrast, the *Failure Packet* is generated and unicast to the source node when an intermediate node fails to forward a data packet to the node that is specified in the packet's header.

- The *Error Packet* is broadcast to the entire network. Any node that receives and uses an Error Packet to update its routing table, should rebroadcast it. In contrast, the *Failure Packet* is unicast only to the source node of the data packet. Meanwhile, the forwarder node may update its RT by deleting entries with the same link_id as the one included in the *Failure Packet*.

4.3.3 MDSDV0 Phases Overview

There are 4 phases that describe the MDSDV0 routing protocol as follows. These phases are specified as pseudocode and illustrated by example in the following section.

1. **Routing Initialization:** This phase allows a new node to get multiple paths to each node in the entire network. As soon as a new node joins the network or a node becomes isolated, it broadcasts a *Hello Message*. Hello Messages are not forwarded. Any neighbour (any node in the transmission range of the new node) that receives the message responds by adding an entry in its NT showing the new node as a neighbour and adding an entry in its RT as a direct route to this new node. The *Link_id* and *Time* fields are set to 0, whereas the TimeOut field is set to a certain time. After adding a route, each neighbour unicasts a Full Dump of its routing table to the new node. On receiving a Full Dump from its neighbours, the new node starts to create its tables (Neighbours and Routing Tables). Afterwards, the new node selects the entries that have link_ids equal to 0, assigns a new link_id to each of them, and then initiates and broadcasts Update Messages to all neighbours to update their link_ids where the link_id is 0, and to get new routes to the new node's neighbours.

2. **New Route Propagation:** This phase describes how other nodes can discover multiple paths to the new node and also how other nodes get new paths that pass through the new node. The new node uses its NT to initiate and broadcast *Update Packets* to its neighbours. The number of Update Packets depends on the number of entries where the *Flag* field = 1. Table 4.3 shows the structure of the *Update Message*. Where:

   Sender: Address of the node that sends the Update Packet.
   First hop: First hop to the destination.
   Second hop: Second hop to the destination.
   No. of hops: Number of hops to the destination.
   Destination: Address of the destination node.
   Link_id: Indicates link between the new node and its 1st hop neighbour
   Neighbour: Address of one of neighbours of the new node.
   Sequence Number: A sequence number generated by the destination node.

   Note: the difference between the sequence number and the link_id is that the sequence number is used to distinguish between fresh and stale routes in the same way as DSDV, whereas the link_id is generated by a new node to distinguish between links to each one hop neighbour.

3. **Route Maintenance:** When a link failure occurs, as detected by no packets being received in an interval or by it failing to forward a packet, the node that detects the failure updates its routing table by deleting any entry that uses the unreachable node as a first hop. Next, it initiates and broadcasts an *Error Packet* to its neighbours. The *Error Packet* includes: Address of the node that detects the failure, the unreachable node address, and the link_id between itself and the unreachable node. Any node that receives this *Error Packet*, checks its routing table and deletes entries where the link id is equal to the link_id that included in the *Error Packet*. If the received node deletes any entry, it should rebroadcast the

Error Packet. By this means, all nodes in the network delete the routes that are using the broken link.

If a source node uses a route with a broken link to send data, the intermediate node that discovers the link failure, selects an alternative route to forward the data. Next, it initiates and sends a *Failure Packet* to the source node to stop using the broken link. The node that detects the failure includes in the *Failure Packet* its address, the First Hop that the source node used to send the data, link_id for the broken link, and the unreachable node address. Table 4.4 shows the *Error Packet* structure and table 4.5 shows the *Failure Packet* structure.

4. **Data Forwarding** When a node has ready data to send, it searches for the best route to the destination and uses it to send its data. The node includes the second hop in the header of the packet to force the intermediate node to use the route where the second hop in the header of the data packet is the first hop in that route. As the intermediate node receives and plans to forward a data, it searches for a route to the destination via the second hop that is included in the data packet's header.

   Note: The best route is the one that has the least number of hops. If two routes have the same number of hops, the one with highest sequence number is the best.

4.3.4 How link failures are modelled in MDSDV simulation

In MDSDV, the broken link is detected by the MAC layer protocol, or it is inferred if no broadcasts have been received for a certain time from a former neighbour. A broken link is described by a metric of ∞ (i.e., any value greater than the maximum allowed metric). When a link to a next hop becomes broken, any route through that next hop is immediately assigned an ∞ metric.
In this respect, we developed two models: *Forward_Error Model* and *TimeOut_Error Model*.

Forward_Error Model:

When the MAC layer reports a broken link during a data-packet transmission, the function *lost_link* function (Function 19 in Appendix E) is called to achieve three tasks.

- Reporting a lost link by calling a *helper_callback* function (Function 18 in Appendix E), which is described in the next model.
- Generating and unicasting a *Failure packet* to the source node to stop sending data through the broken link. The *Failure packet* includes four fields (Sender, First hop, Failed link id, Destination).
- Finding an alternative path to forward a data packet. If an alternative path is not available, the packet is queued.

When the source node receives the *Failure packet*, it updates its routing table by deleting the entry where the destination and first hop are the same as the destination and first hop in the *Failure packet*. Next, the source starts to use an alternative path (if exist) to continue sending data packets.

TimeOut_Error Model:

MDSDV maintains a Neighbour table which contains an entry for each neighbour. One of the fields of the entry is the *TimeOut* field. This field is updated when a control packet is received from the neighbour. When the time in this field is expired, the link with the neighbour is considered as a broken link. Consequently, the function *helper_callback* is called to deal with the broken link.
Using this function, the node updates its routing table by assigning an ∞ metric to each entry where the unreachable neighbour acts as a next hop. Then, the node

broadcasts an *Error packet* which contains three fields (sender node, Destination, link Id). Any neighbour that receives the *Error packet* must delete any entry where the *Link id* is equal to the *Link id* that is included in the *Error packet*.

4.4 Algorithms with an Example

To illustrate the four phases of MDSDV0, we present this example which describes a network of 8 nodes. Figure 4.3 shows the new node (node 8) broadcasting a *Hello Message* and receiving routing tables information (*Full Dumps*) from its neighbours.

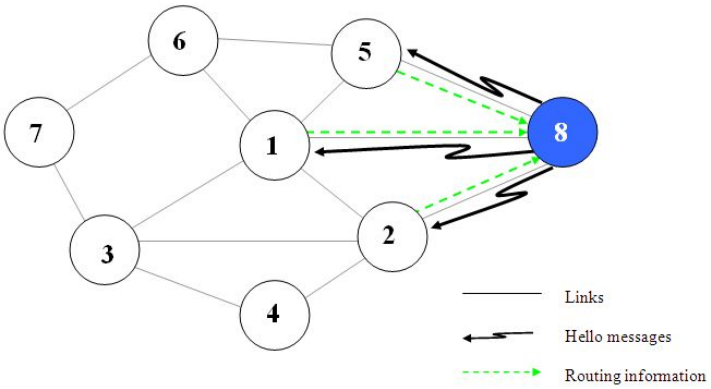

Figure 4.3: Node 8 sends and receives Routing Messages

4.4.1 Routing Initialization

As node 8 joins the network, it increments its sequence number and broadcasts a *Hello Message*. All neighbours of node 8 (node 1, node 2, and node 5) receive the *Hello Message*, use the ***Receive Hello Message*** algorithm (Figure 4.4) to add an entry as a direct path to node 8, increment their sequence numbers, and respond by unicasting their routing tables (Full Dump) to node 8 as shown in Figure 4.3. Each neighbour includes its new sequence number in the Full Dump. Tables 4.6a, 4.6b, and 4.6c show the first 6 columns of the routing tables of the neighbours (nodes 1, 2, and 5) after using the ***Receive Hello Message*** algorithm.

```
01  The neighbour node adds a new entry in its RT by filling the fields
    as follows:
02      Destination ← Address of the node that sent the Hello Message
03      First hop ← Address of the node that sent the Hello Message
04      Second hop ← Null
05      Number of hops ← 1
06      Link_id ← 0
07      Seq_No ← The sequence number field in the received Hello Message
08      Time ← now
09  Send copy of routing table to node that sent the Hello Message.
```

Figure 4.4: Receive *Hello Message* Algorithm

Ds	Fh	Sh	Nh	Ln	Sn
2	2	Null	1	20001	...
2	3	2	2	30002	...
3	3	Null	1	30001	...
3	2	3	2	30002	...
3	6	7	3	70001	...
4	2	4	2	40001	...
4	3	4	2	40002	...
5	5	Null	1	50001	...
5	6	5	2	60002	...
6	6	Null	1	60001	...
6	5	6	2	60002	...
6	3	7	3	70002	...
7	3	7	2	70001	...
7	6	7	2	70002	...
8	**8**	**Null**	**1**	**0**	**2**

(a) Routing table of node 1

Ds	Fh	Sh	Nh	Ln	Sn
1	1	Null	1	20001	...
1	3	1	2	30001	...
3	3	Null	1	30002	...
3	1	3	2	30001	...
3	4	3	2	40002	...
4	4	Null	1	40001	...
4	3	4	2	40002	...
5	1	5	2	50001	...
6	1	6	2	60001	...
6	3	7	3	70002	...
7	3	7	2	70001	...
7	1	6	3	70002	...
8	**8**	**Null**	**1**	**0**	**2**

(b) Routing table of node 2

Ds	Fh	Sh	Nh	Ln	Sn
1	1	Null	1	50001	...
1	6	1	2	60001	...
1	8	1	2	80001	...
2	1	2	2	20001	...
3	1	3	2	30001	...
3	6	7	3	70001	...
4	1	2	3	40001	...
6	6	Null	1	60002	...
6	1	6	2	60001	...
7	6	7	2	70002	...
7	1	3	3	70001	...
8	**8**	**Null**	**1**	**0**	**2**

(c) Routing table of node 5

Table 4.6: Routing tables of the neighbours

When node 8 receives the routing tables from its neighbours, it invokes the ***Creating the routing table*** algorithm shown in figure 4.5 to create its own routing table by adding an entry for each route, and generates link_ids (Ln) for the new routes (where link_id = 0) between itself and each neighbour (Ds). Then, it invokes the ***Filtering the routing table*** algorithm (Figure 4.6) to delete undesired routes and generate disjoint paths. By this means, node 8 (new node) gets multiple routes for each destination in the entire network. Table 4.7 shows the routing table of node 8 after creating and filtering. Next, node 8 creates its Neighbours Table (NT) by adding an entry for each neighbour as shown in table 4.8.

Generating link ids is done according to the equation (4.1). In this example, node 8 generates three new link ids which are:

- 80001 is between itself and node 1.
- 80002 is between itself and node 2.
- 80005 is between itself and node 5.

$$\textit{New Link Id} = \textit{My Address} * 10000 + \textit{Address Of The Neighbour} \quad (4.1)$$

Destination	First Hop	Second Hop	Number Of hops	Link Id	Sequence Number	Time	... ...
1	1	Null	1	**80001**	56		
1	2	1	2	20001	...		
1	5	1	2	50001	...		
2	2	Null	1	**80002**	24		
2	1	2	2	20001	...		
3	1	3	2	30001	...		
3	2	3	2	30002	...		
3	5	6	4	70001	...		
4	2	4	2	40001	...		
4	1	3	3	40002	...		
5	5	Null	1	**80005**	42		
5	1	5	2	50001	...		
6	5	6	2	60002	...		
6	1	6	2	60001	...		
7	2	3	3	70001	...		
7	5	6	3	70002	...		

Table 4.7: Node 8 routing table after filtering

Neighbour Id	Link-Id	Sequence Number	Flag
1	80001	56	1
2	80002	24	1
5	80005	42	1

Table 4.8: Neighbours Table (NT) of node 8

```
While (There are more entries received from the neighbour)
{
   Fill the entry fields in my routing table as follows...
   First hop field ← address of the neighbour.

   If (Link_id field = 0)
   {
       The node generates a link id as follows:
       Link_id field ← My Address * 10000 + Address Of The Neighbour.
       Update my NT with an entry for the neighbour.
   }
   Else
       Link_id field = Link_id

   If (My address = the address in the destination field)
   {
        Second hop field ← Second hop
        Destination field ← address of the neighbour
        Number of hops field ← Number of hops
        Seq_No field ← sequence number of the neighbour
   }
   Else
   {
      Second hop field ← First hop
      Destination field ← Destination
      Number of hops field ← Number of hops + 1
      Seq_No field ← sequence number in the received entry
   }
}
```

Figure 4.5: Creating the Routing Table Algorithm

```
01  While (There are more entries in the routing table)
02  {
03    If there are two or more entries that have the same destination, same first hop,
      and a different number of hops
04    {
05       delete the entry with stale route or bigger number of hops.
06    }
07    If a second hop of an entry is equal to a first hop of another entry
08    {
09       delete the entry with the second hop if it is not the destination.
10    }
11    If there are two entries with the same destination and same link_id
12    {
13       delete the stale or longer route.
14    }
15    If there are two entries with the same destination, same first hop, and same number
      of hops
16    {
17       delete one of the entries
18    }
19    If there are two entries with the same destination, different first hop, and
      same second hop
20    {
21       delete one of the entries if the second hop is not the destination
22    }
23  }
```

Figure 4.6: Filtering the Routing Table Algorithm

4.4.2 Routing Updating

Tables 4.7 and 4.8 show that there are three new link ids(i.e., 80001, 80002, 80005). Therefore, node 8 uses ***Initiating an Update Packet*** algorithm (Figure 4.7) to initiate 3 Update Packets, and broadcasts them as shown in Figure 4.8. Table 4.9 shows the format of the three Update Packets that are generated by the new node (node 8).

When the neighbour receives the Update Packets, it uses the ***Receiving an Update Packet*** algorithm shown in Figure 4.9 to update its RT. Next, it updates the Update

Packets according to the ***Updating the Update Packet*** algorithm (Figure 4.10), and broadcasts them to its neighbours as well. Table 4.10 shows the Update Packets that node 5 updated and broadcast.

```
01  The new node initiates an Update Packet by setting the fields of the packet
    as follows:

02    Sender  ←  Address of the new node that initiates the update packet
03    First hop  ←  Null
04    Second hop  ←  Null
05    Number of hops  ←  0
06    Destination  ←  Null
07    Link_id  ←  The link id between the new node and its neighbour node
08    Neighbour  ←  address of the neighbour
09    Sequence Number  ←  Sequence number generated by the neighbour
```

Figure 4.7: Initiating an *Update Packet* Algorithm

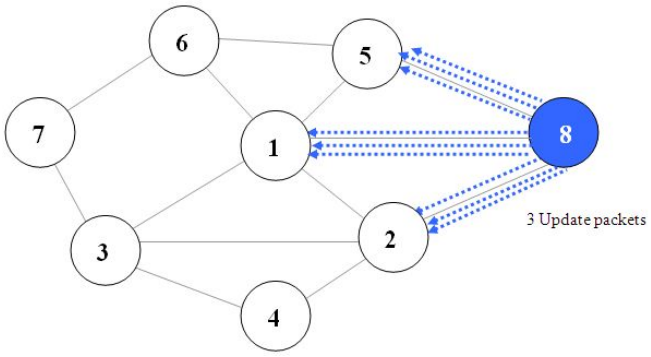

Figure 4.8: Node 8 Initiates and broadcasts 3 *Update Packets*

Sender	First Hop	Second Hop	Number Of hops	Destination	Link Id	Neighbour	Sequence Number
8	Null	Null	0	Null	80001	1	56
8	Null	Null	0	Null	80002	2	24
8	Null	Null	0	Null	80005	5	42

Table 4.9: The Update Packets generated by Node 8

```
01  Whenever a node receives an Update Packet, it checks..
02  If my address is equal to destination or first hop fields in the Update Packet
03  {
04      the Update Packet is discarded because the packet is generated by me.
05  }
06  else
07  {
08      Number of hops in the received Update Packet is checked....
09      If (number of hops  =  0)
10      {
11         This means that the packet is received from a new node.
12         Invoke algorithm shown in Figure 4.11
13      }
14      else
15      {
16         This means that the packet is received from a known node
17         Invoke algorithm shown in Figure 4.13
18     }
19  }
```

Figure 4.9: Receiving an *Update Packet* Algorithm

```
01  When a node receives an Update Packet, it modifies the packet's fields as follows:
02  Sender ←  my address
03  Copy Link id, Neighbour, and Sequence Number fields
04  If (number_of_ hops  =  0 and node address ≠ neighbour field)
05  {
06     First hop  ←  sender field in the Update Packet
07     Second hop  ←  Neighbour field in the Update Packet
08     Number_of_hops  ←  2
09     Destination  ←  Address in the neighbour field
10  else
11  {
12     First hop  ←  sender field in the Update Packet
13     Second hop  ←  First hop in the Update Packet
14     Number_of_hops  ←  Number of hops in the Update Packet + 1
15     Destination  ←  Destination in the Update Packet
16  }
```

Figure 4.10: Updating the *Update Packet* Algorithm

Sender	**First Hop**	**Second Hop**	**Number Of hops**	**Destination**	**Link Id**	**Neighbour**	**Sequence Number**
5	8	1	2	1	80001	1	56
5	8	2	2	2	80002	2	24
5	8	Null	1	8	80005	5	42

Table 4.10: Node 5 updates and broadcasts the Update Packets

By updating its routing table according to the received Update Packets, each node in the entire network gets multiple paths to the new node and other paths to the neighbours of the new node. Tables 4.11a - 4.11g show the new routes that are obtained by each node in the network after receiving the Update Packets.

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
2	8	2	2	8-2	24
5	8	5	2	8-5	42
8	8	Null	1	8-1	2
8	2	8	2	8-2	2
8	5	8	2	8-5	2
..	..	..	..	..	..
..	..	..	..	..	..

(a) Routing table of node 1

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
1	8	1	2	8-1	56
5	8	5	2	8-5	42
8	8	Null	1	8-2	2
8	1	8	2	8-1	2
8	3	7	5	8-5	2
..	..	..	..	..	..
..	..	..	..	..	..

(b) Routing Table of node 2

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
5	2	8	3	8-5	42
8	1	8	2	8-1	2
8	2	8	2	8–2	2
8	7	6	4	8-5	2
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..

(c) Routing Table of node 3

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
5	2	8	3	8-5	42
8	2	8	2	8-2	2
8	3	1	3	8-1	2
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..

(d) Routing table of node 4

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
1	8	1	2	8-1	56
2	8	2	2	8-2	24
8	8	Null	1	8-5	2
8	1	8	2	8-1	2
8	6	7	5	8-2	2
..	..	..	..	..	..
..	..	..	..	..	..

(e) Routing Table of node 5

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
2	5	8	3	8-2	24
8	1	8	2	8-1	2
8	5	8	2	8-5	2
8	7	3	4	8-2	2
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..

(f) Routing Table of node 6

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
8	3	2	3	8-2	2
8	6	5	3	8-5	2
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..

(g) Routing Table of node 7

Table 4.11: Routing Tables of the nodes after receiving the Update Packets

```
01  When a node receives an Update Packet from a new node, it checks...
02  If my address = neighbour field in the Update Packet
03  {
04     My RT is updated by setting the link_id field of the corresponding entry
05     to the link_id field in the Update Packet. Then, the algorithm shown in
06     Figure 4.10 is used to update the Update Packet. Finally, I broadcast
07     the Update Packet to my neighbours.
08  }
09  Else
10  {
11    Invoking the algorithm shown in Figure 4.12 to add an entry as a path
12    to the node in the neighbour field through the new node. Then, the
13    algorithm shown in Figure 4.10 is used to update the Update Packet.
14    Finally, I broadcast the Update Packet to my neighbours.
15  }
```

Figure 4.11: Receiving an *Update Packet* from a new neighbour Algorithm

```
01 If number_of_ hops  =  0  AND  node number <> neighbour field
02 {
03    The node adds an entry as a path to the neighbour of the new node by setting the
      fields as follows...
04    First hop field  ←  Sender field in the Update Packet
05    Second hop field  ←  neighbour field in the Update Packet
06    Number of hops field  ←  2
07    Destination field  ←  neighbour field in the Update Packet
08    Copy Link_id, Sequence Number fields
09 }
```

Figure 4.12: Getting a route to the neighbour of the new node Algorithm

```
01  If the destination and first hop in any entry are equal to the destination and first
    hop in the Update Packet. Also, the first hop and the destination in the Update
    Packet are not the same. Finally, if the received Sn is smaller {
02      the Update Packet is discarded
03  }
04  Else{
05       If the destination and link_id in any entry are equal to the destination and
         link_id in the received Update Packet {
06           If number of hops in the Update Packet ≥ number of hops in the entry{
07               the Update Packet is discarded
08           }
09           Else{
10              If sender field in the Update Packet  ≠  first hop field in the entry
11                 the Update Packet is discarded
12              Else
13                 the entry is updated according to Update Packet
14           }
15       }
16       Else{
17           If the destination in the Update Packet is equals to destination in any entry{
18               If the sender is equal to the first hop{
19                  If number of hops in the Update Packet ≥ number of hops in the entry
20                     the Update Packet is discarded
21                  Else
22                     the entry is updated according to the Update Packet
23               }
24           }
25           Else{
26              If any entry has the same destination and the same link-id{
27                 If number of hops in the Update Packet ≥ number of hops in the entry
28                    the Update Packet is discarded
29                 Else
30                    the entry is updated according to the Update Packet
31              }
32              Else
33                 My RT is updated according to the algorithm in figure 4.14 by adding an
                   entry as a route to the destination. Then, the algorithm in figure 4.10 is
                   used to update the Update Packet. Finally, the packet is broadcast to
                   neighbours.
34           }
35       }
36  }
```

Figure 4.13: Receiving an *Update Packet* from a known neighbour Algorithm

```
01 The node updates an entry in its RT by setting the fields as follows...
02  First hop field  ←   Sender in the Update Packet
03  Second hop field  ←   First hop in the Update Packet
04  Number of hops field  ←   Number of hops in the Update Packet + 1
05  Destination field  ←   Destination in the Update Packet
06  Link_id field  ←  Link_id in the Update Packet
07  Sequence No. field  ←  Sequence No. in the Update Packet
```

Figure 4.14: Updating an existing entry Algorithm

4.4.3 Link Failure

We assume that node 8 moves away from node 2, and node 2 has discovered the link failure as shown in Figure 4.15. Node 2 does the following in response to the discovered link failure.

- Deletes any entry where node 8 acts as a first hop.
- Initiates and broadcasts an *Error Packet* as shown in figure 4.15 to all its one hop neighbours (nodes 1, 3, and 4) telling them that node 8 is unreachable. The *Error Packet* includes its address (node 2), the unreachable node address (node 8), and the link id between these two nodes (80002). Table 4.12 shows the structure of the *Error Packet* initiated by node 2.

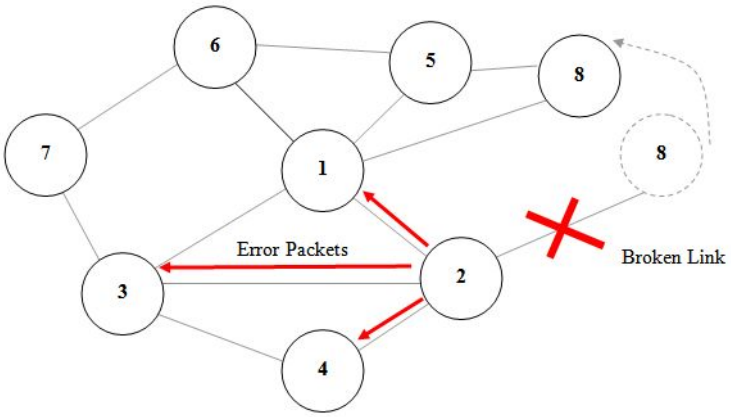

Figure 4.15: Node 2 initiates and broadcasts an *Error Packet* to its neighbours

Table 4.13 shows the routing table of node 2 and the deleted entries where node 8 is a first hop. Any node that receives the *Error Packet* uses the algorithm in figure 4.16 to update its routing table as follows:

- Discard the packet if its address is equal to the sender field of the *Error Packet*. This is because the packet was generated by the same node.
- If the received node address is equal to the destination field, the node deletes any entry where the first hop is equal to the sender field. This means that the sender is telling the received node that the link between the sender and the receiver is broken. So, the receiver node has to delete any route where the sender acts as a first hop.
- If the received node address is neither equal to the sender nor the destination field of the *Error Packet*, the node deletes any entry where the link_id is equal to the link_id field of the *Error Packet*.

Sender node	Destination node	Broken Link-Id
2	8	8-2

Table 4.12: The Error Packet initiated by node 2

Destination	First Hop	Second Hop	Number Of hops	Link Id	Sequence Number	Time	....
1	1	Null	1	2-1	...	...	
1	3	1	2	3-1	...	...	
1	8	1	2	8-1	56		
3	3	Null	1	3-2	...	...	
3	1	3	2	3-1	...	...	
3	4	3	2	4-2	...	...	
4	4	Null	1	4-1	...	...	
4	3	4	2	4-2	...	...	
5	1	5	2	5-1	...	...	
5	8	5	2	8-5	42		
6	1	6	2	6-1	...	...	
6	3	7	3	7-2	...	...	
7	3	7	2	7-1	...	...	
7	1	6	3	7-2	...	...	
8	8	Null	1	8-2	2	...	
8	1	8	2	8-1	2	...	
8	3	7	5	8-5	2	...	

Table 4.13: Routing Table of node 2 showing the deleted entries

```
01  If my address = Sender field in the Error Packet
02  {
03      The packet is discarded because the packet was generated by me
04  }
05  Else
06  {
07     If my address = destination field in the Error Packet
08     {
09         Delete any entry where the first hop is equal to the sender field in the
10         Error Packet (regardless of the link id).
11     }
12     else
13     {
14         Delete any entry where the link id is equal to the link_id field in the
15         Error Packet
16     }
17  }
```

Figure 4.16: Receiving an *Error Packet* Algorithm

Nodes that update their routing tables, should rebroadcast the *Error Packet*, whereas nodes that do not update their routing tables are not required to rebroadcast it. Tables (4.14a - 4.14g) show the routing tables of nodes 1, 3, 4, 5, 6, 7, and 8 after updating. Whereas, Table 4.13 shows the routing table of node 2 after discovering the link failure and updating its routing table.

The *Error Packet* is broadcast through the entire network as discussed in 4.3.2. So, a node may receive the same *Error Packet* several times. When a node receives an *Error Packet* at the first time, it updates its RT by deleting the entries where the link id is equal to the link id in the *Error Packet*. When the same node receives the same *Error Packet* from another neighbour, it will not find an entry that has the same destination and same link id any more. In this case the node simply discards the *Error Packet*.

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
2	8	2	2	8-2	24
5	8	5	2	8-5	42
8	8	Null	1	8-1	2
8	2	8	2	8-2	2
8	5	8	2	8-5	2
..	..	..	..	..	..

(a) Routing table of node 1

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
5	2	8	3	8-5	42
8	1	8	2	8-1	2
8	2	8	2	8-2	2
8	7	6	4	8-5	2
..	..	..	..	..	..
..	..	..	..	..	..

(b) Routing Table of node 3

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
5	2	8	3	8-5	42
8	2	8	2	8-2	2
8	3	1	3	8-1	2
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..

(c) Routing Table of node 4

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
1	8	1	2	8-1	56
2	8	2	2	8-2	24
8	8	Null	1	8-5	2
8	1	8	2	8-1	2
8	6	7	5	8-2	2
..	..	..	..	..	..

(d) Routing Table of node 5

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
2	5	8	3	8-2	24
8	1	8	2	8-1	2
8	5	8	2	8-5	2
8	7	3	4	8-2	2
..	..	..	..	..	..
..	..	..	..	..	..

(e) Routing Table of node 6

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
..	..	..	..	..	..
8	3	2	3	8-2	2
8	6	5	3	8-5	2
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..

(f) Routing Table of node 7

Ds	Fs	Sh	Nh	Ln	Sn
..	..	..	..	..	..
1	1	Null	1	8-1	56
2	2	Null	1	8-2	24
5	5	Null	1	8-5	42
..	..	..	..	..	..
..	..	..	..	..	..
..	..	..	..	..	..

(g) Routing Table of node 8

Table 4.14: Routing Tables of the nodes with deleted entries

4.4.4 Forwarding Data

To illustrate this phase, we describe two cases: the first case where the source node selects an active route, and the second phase where the source selects a route with a broken link. Let's assume that node 7 needs to send data to node 8. Table 4.11g shows that node 7 has 2 routes with the same metric to node 8 given by the following entries; (8 - 3 - 2 - 3 - 80002 and 8 - 6 - 5 - 3 - 80005) in its routing table, and has to select the best route.

- If node 7 selects the second route which is specified by the entry 8 - 6 - 5 - 3 - 80005. This means that it has to forward the packet to node 6 that acts as a first hop, and includes node 5 as a second hop in the data packet's header. As an

intermediate node, node 6 receives the data packet and locates a route to node 8 through node 5. This route is specified by the entry 8 - 5 - 8 - 2 - 80005 in Table 4.11f. Then, node 6 modifies the second hop field in the data packet's header to the second hop in that entry which is 8, and forwards the packet to node 5. Also, as an intermediate node, when node 5 receives the data packet, it locates the route to node 8 through node 8 (direct route). In this case, node 5 sends the packet to node 8 without modifying the data packet's header because the destination is a one hop neighbour.

- If node 7 selects the first route which is specified by the entry 8 - 3 - 2 - 3 - 80002 to send data to node 8. So, it includes node 2 as a second hop in the data packet's header and sends the packet to node 3. When node 3 receives the packet, it locates a route to node 8 through node 2. Node 3 should use the route specified by the entry 8 - 2 - 8 - 2 - 80002 (Table 4.11c), modify the second hop field of the data packet's header to be node 8, and forward the packet to node 2. When node 2 receives the packet, it discovers that node 8 is unreachable (link failure). In this case, node 2 should locate an alternative route to node 8 in its routing table. Table 4.11b shows that node 2 has two alternative routes to node 8 specified by the entries 8 - 1 - 8 - 2 - 80001 and 8 - 3 - 7 - 5 - 80005, and it has to select the best one which is specified by the entry 8 - 1 - 8 - 2 - 80001. Next, it initiates a *Failure Packet* and unicasts it to node 7 (Source node) to stop sending data to node 8 using the route with the link id 8002.

4.5 How to build Disjoint Paths

Disjoint paths have the desirable property that they are more likely to fail independently. There are two types of disjoint paths: *node disjoint* and *link disjoint*. Node disjoint paths do not have any nodes in common, except for the source and the destination nodes. They are also known as totally disjoint paths. In contrast, link disjoint

paths do not have any common links, but may have common nodes. In a traditional distance vector protocol, a node only keeps track of the next hop (and distance via the next hop) for each path. This limited one hop information is insufficient for a node to ascertain whether two paths obtained from two distinct neighbours are indeed disjoint.

MDSDV0 guarantees that alternate paths are node-disjoint. In section 5.5, we provide a rigorous argument that MDSDV constructs node disjoint paths. This section describes how MDSDV0 builds node disjoint multiple paths, by considering the example in Figure 4.3 where node 8 receives *Full Dumps* from its neighbours. As a part of the received information, it receives 8 entries for node 3 as shown in table 4.15. The first 3 entries are received from node 1, the following 3 entries are received from node 2, and the last 2 entries are received from node 5.

The protocol utilizes the algorithm in Figure 4.5 to create its routing table. The created entries that belong to node 3 are listed in table 4.16. Next, the new node (node 8) invokes the algorithm in Figure 4.6 (Filtering) to delete undesired routes. The steps for filtering the routing table are described as follows:

Destination	First Hop	Second Hop	Number Of hops	Link Id	Sequence Number
3	3	Null	1	3-1	...
3	2	3	2	3-2	...
3	6	7	3	7-1	...
3	3	Null	1	3-2	...
3	1	3	2	3-1	...
3	4	3	2	4-2	...
3	1	3	2	3-1	...
3	6	7	3	7-1	...

Table 4.15: Routing information received by node 8 from its neighbours

- At the beginning, the node compares entry 1 with entry 2. Both entries have the same destination, the same first hop, and a different number of hops. Thus, the protocol chooses to keep entry 1, and deletes entry 2 because entry 1 has a lower number of hops (line 03).

Destination	First Hop	Second Hop	Number Of hops	Link Id	Sequence Number
3	1	3	2	3-1	...
3	1	2	3	3-2	...
3	1	6	4	7-1	...
3	2	3	2	3-2	...
3	2	1	3	3-1	...
3	2	4	3	4-2	...
3	5	1	3	3-1	...
3	5	6	4	7-1	...

Table 4.16: Created entries by node 8 regarding destination node 3

- Comparing entries 3 and 1. Both entries have the same destination, the same first hop, and a different number of hops. Thus, the protocol chooses to keep entry 1, and deletes entry 3 because entry 1 has a lesser number of hops (line 03).
- Comparing entry 4 and entry 1. The node keeps both routes because they are disjoint.
- Comparing entries 5, 1, and 4. Entry 5 and entry 4 have the same destination, the same first hop, and a different number of hops. So, entry 5 is deleted because it has a larger number of hops (line 03).
- Comparing entries 6, 1, and 4. Entries 4, 6, have the same destination, the same first hop, and a different number of hops. So, entry 6 is deleted because it has a larger number of hops (line 03).
- Comparing entries 7, 1, and 4. Entry 7 is deleted because the second hop in entry 7 is the same as the first hop in entry 1 and the second hop is not the destination (line 07).
- Comparing entries 8, 1, and 4. The node keeps all routes because they are disjoint.

After filtering its routing table, node 8 gets 3 disjoint paths to node 3 as shown in table 4.17 and Figure 4.17.

Destination	First Hop	Second Hop	Number Of hops	Link Id	Sequence Number
3	1	3	2	3-1	...
3	2	3	2	3-2	...
3	5	6	4	7-1	...

Table 4.17: Disjoint Paths to node 3 generated by node 8

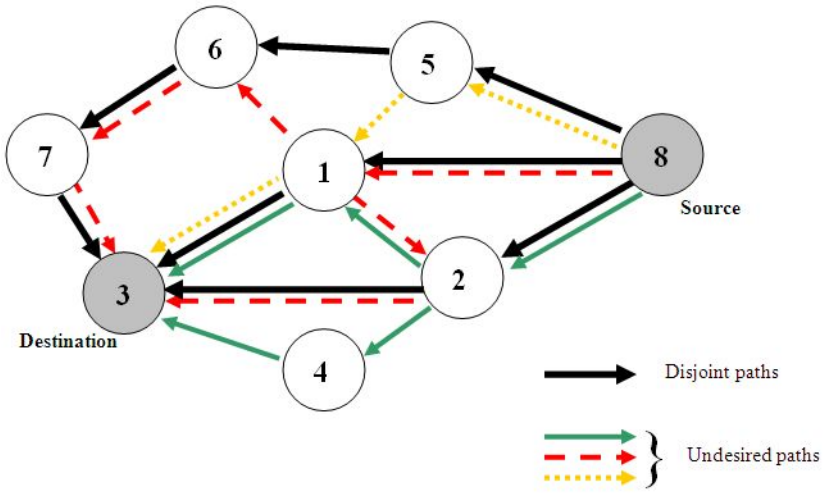

Figure 4.17: Node 8 generates 3 Disjoint Paths to node 3

From figure 4.17, we can find that node 8 gets 3 disjoint paths to node 3 which are:

8 - 1 - 3

8 - 2 - 3

8 - 5 - 6 - 7 - 3

In the other hand it deletes the undesired routes which are:

8 - 1 - 2 - 3 (red colour)

8 - 1 - 6 - 7 - 3 (red colour)

8 - 2 - 1 - 3 (green colour)

8 - 2 - 4 - 3 (green colour)

8 - 5 - 1 - 3 (yellow colour)

4.6 Limitations

The down side of MDSDV0 is that there is a huge control overhead for maintaining the routing table especially for large and dynamic networks. This overburdens the network with control packets rather than data packets. The main source of control packets is from the *Update packets* and *Error Packets* that are broadcast to the entire network. When a node receives an *Update Packet*, it needs to modify the packet and rebroadcast it to its neighbours if its routing table is updated. Also, when a node receives an *Error Packet*, it has to rebroadcast it to its neighbours if its routing table is updated. As a result, both of the packets are propagated many times consuming scarce resources such as bandwidth and power, and leading to potential network congestion.

Four simulation experiments illustrate the control overhead produced by MDSDV0 . We consider 30, 50, 70, and 100 node networks roaming in a rectangular area of 670m X 670m over a 100 sec period. The pause time and speed are set to 0 sec and 20 m/sec respectively. Tables 4.18 and 4.19 show the results obtained by the experiments. Table 4.18 shows that the Packet Delivery Fraction (PDF) in row three is low due to the huge number of control packets. Table 4.19 shows the number of each type of control packet produced by the protocol. It is clear that a large number of control packets are propagated to deliver less than 58% of the transmitted data packets. Also, we can see that the number of *Update packets* and *Error Packets* increases as the number of nodes increases. This is because these two packets are sent to the entire network.

Thus, some techniques should be added to reduce the number of control packets.

Seq. No.	Performance metric	Network size			
		30 nodes	50 nodes	70 nodes	100 nodes
1	Sent packets	4002	4613	4605	4577
2	Received packets	2239	2653	2367	1907
3	PDF	55.95	57.51	51.40	41.67
4	NRL	0.91	2.64	5.98	13.48
5	Average e-e delay(ms)	22.34	81.52	493.61	74.55
6	No. of dropped data (packets)	1714	1879	2174	2590
7	No. of dropped data (bytes)	912088	999948	1156808	1378080
8	Average Throughput[kbps]	94.14	111.56	99.50	80.17

Table 4.18: MDSDV0 performance results

Seq. No.	Control packet type	Network size			
		30 nodes	50 nodes	70 nodes	100 nodes
1	Hello packets	4	6	6	3
2	Available packets	352	588	823	1187
3	Full_Dump packets	47	85	124	234
4	Update packets	1044	4075	8985	16215
5	Error packets	569	2221	4197	8051
6	Failure packets	27	39	30	20
7	Total number of control packets	2043	7014	14165	25710

Table 4.19: MDSDV0 control packets

Chapter 5 presents the design of a revised protocol (MDSDV) with modifications to improve the PDF and minimize the control overhead. Appendix A contains a performance comparison between MDSDV0 and MDSDV which shows that MDSDV0 has a significantly high control overhead for both dynamic and static networks, and that MDSDV makes dramatic improvements in delivering data and reducing the control overhead.

Chapter 5

Final MDSDV Design

This chapter describes the design of a revised version of MDSDV0. We call the modified version as MDSDV. Section 5.1 outlines the differences between MDSDV0 and MDSDV. Section 5.2 describes the three tables that are maintained by MDSDV. The control packets are listed and described in section 5.3 . Section 5.4 specifies the mechanisms of MDSDV as pseudo code. A rigorous argument that MDSDV maintains node disjoint paths is presented in section 5.5. Finally, Section 5.6 presents the functionality testing of MDSDV.

5.1 Introduction

Nodes in ad hoc networks are distinguished by their limited resources such as bandwidth, energy, and memory as well as mobility. Nodes in ad hoc networks are free to move over a certain area. Because of this movement, the network topology may frequently change. This means that we need a routing protocol that quickly adapts to topology changes.

MDSDV0 suffers from sending a large number of control packets as discussed in

section 4.6. Therefore, in this chapter we use another technique to reduce this number while maintaining the flexibility and reliability of the protocol.

Before describing MDSDV in detail, we give an overview of how it is different from MDSDV0.

1. MDSDV0 maintains two tables: Routing table and Neighbours table, whereas MDSDV maintains three table: Routing table, Neighbours table, and Queue table.

2. In MDSDV0 nodes use two announcement packets: *Hello Message* indicating that it is a new node (has no neighbours) and *Available Message* which is broadcast periodically to tell the other neighbours that it is still available in the network, whereas MDSDV nodes only use *Hello Message* which is broadcast only when the node has no neighbours.

3. In MDSDV0, nodes use an *Update Packet* which is generated and propagated by a new node after creating its routing table. Nodes modify and broadcast the *Update Packet* if their routing table is updated. On the other hand, MDSDV nodes use two types of update packets: *Full Dump* and *Update Packet* . The *Full Dump* includes the best route for each destination, whereas the *Update Packet* includes all routes. The *Full Dump* is unicast only to a new neighbour, and the *Update Packet* is only broadcast to current neighbours. More details on the control packets of MDSDV is given in section 5.3

4. The *Error Packet* in MDSDV0 is sent to the entire network, whereas in MDSDV the *Error Packet* is only broadcast to current neighbours.

5.2 Tables

In MDSDV, each node maintains three tables Neighbours Table (NT), Routing Table (RT) and Queue Table (QT).

- ***Neighbours Table (NT):*** Instead of the *flag* field in MDSDV0 , the *TimeOut* field is used to identify the time for which this node can be considered as a neighbour. If no packets are received from the neighbour before this time has expired, the neighbour will be considered as an unreachable node and the entry will be deleted. The field *TimeOut* is updated whenever packets are received from a neighbour node. Figure 5.1 shows the structure of a NT's entry.

Field name	Description
Neighbour id	Address of the neighbour node
Link-id	An identifier generated for the new routes
TimeOut	Within this time, the node is considered as a neighbour

Table 5.1: Neighbours table structure (NT)

- ***Queue Table (QT):*** When a node has a data packet to send or forward and has no route to the desired destination, it uses this table to queue the packet. When a node gets a new path to any destination, this table is checked if it contains a data belongs to this destination. The structure of Queue table is shown in Figure 5.2

Field name	Description
Destination id	Address of the unreachable node
data	Data packets that the node is unable to forward

Table 5.2: Queue table structure (QT)

- ***Routing Table (RT):*** In MDSDV, we exclude the *queue* field from the routing table and included in a separate table called *Queue table* (QT). This is to make it easier for the node to check if it has data packets that have not been delivered. The *TimeOut* field is removed from the routing table and included in the Neighbours Table (NT). We add a new field called *TimeToLive*, which is used to delete stale routes. Figure 5.3 shows the structure of RT's entry.

Field name	Description
Destination	Address of the destination node
Next hop	The first hop to the destination
Second hop	The second hop to the destination
Number of hops	Number of hops to the destination
Link-id	An identifier generated for the new routes
Sequence number	A sequence number generated by the destination to distinguish stale routes
Changed at	The time that the path has been obtained or updated
TimeToLive	The time that the path should be deleted

Table 5.3: Routing table structure (RT) entry

5.3 MDSDV Packets

Some of the control packets have been modified and the following is a description of the control packets used by MDSDV.

1. ***Hello Message (HM):*** Periodically each node increments its sequence number and checks its Neighbours Table (NT). If the node's NT is empty (no entries), the node generates and broadcasts a *Hello Message* containing the new sequence number. The packet is used instead of both *Hello Message* and *Available Message* in MDSDV0.

2. ***Update Packet (UP):*** Periodically each node in the network checks its Neighbours Table (NT). If it contains at least one entry, the node increments its sequence number, generates and broadcasts an *Update Packet* including all routes in its routing table. Figure 5.4 shows the structure of the *Update Packet* which has the same structure as the *Full Dump*.

Destination	First hop	Second hop	Number of hops	Link Id Number	Sequence Number

Table 5.4: Structure of the *Update Packet (UP)* and *Full Dump (FD)* entries

3. ***Full Dump (FD):*** When a node receives a *Hello Message* or an *Update Packet* from a new neighbour (No entry belongs to the sender in Neighbours Table), it responds by unicasting a full dump of its routing table to the new node. The *Full Dump* includes only one route (the best) for each destination. The structure of the *Full Dump* entry is shown in figure 5.4. By receiving a *Full Dump*, a node may get full topology of the network if it is a new node, and may get more information if it is already a participating node. This depends on the information that the *Full Dump* sender has. The difference between a *Full Dump* and an *Update Packet* is the number of routes for each destination.

4. ***Error Packet (EP):*** This type of packet is propagated by any node that discovers a broken link. An *Error Packet* includes the *node id* that discovers the broken link, the *unreachable node id*, and the *link id* between both of them. Figure 5.5 shows the structure of the *Error Packet* .

   When a node receives an *Error Packet*, it deletes any entry from its routing table where the *link id* is the same as the *link id* that is included in this packet (same as MDSDV0). The only difference is that the received node does not need to rebroadcast the packet (to reduce number of control packets).

Packet Sender	Destination	Link-Id

Table 5.5: The structure of the *Error Packet*

5. ***Failure Packet (FP):*** When a node has a data packet that is ready to send, it selects the best route to the desired destination and includes the *second hop* of that route in the data packet's header. As the intermediate node receives a data packet, it forwards the packet to the node that is included in the data packet's header. If the intermediate node fails to forward the data packet to the specified node, it unicasts a *Failure Packet (FP)* to the source node. Thus, the source node stops transmitting data packets along this invalid route.

   The *Failure Packet* includes the *source node id* that is sending the data, the *first hop* which is the neighbour node that the source node used to send the data, and the *destination* which is the unreachable node. Figure 5.6 shows the structure of the *Failure Packet* .

Data Packet Sender	First hop	Destination

Table 5.6: The structure of the *Failure Packet*

Note: In addition to the difference in the structure of *Error Packet* and *Failure Packet*, the *Error Packet* is broadcast to the neighbours by the node that detects the link failure, whereas the *Failure Packet* is unicast only to the source node by the intermediate node when it fails to forward a data packet.

5.4 MDSDV Mechanism

5.4.1 Sending Data Packets

For sending a data packet, we use three additional fields in the data packet's header: *current_node*, *first_node*, and *forwarded_node*.

current_node: This field is used to store the node address that deals with the data packet (source node or intermediate node).

first_node: The source node uses this field to store the first hop of the selected path to send the data packet.

forwarded_node: This field is used to store the second hop of the selected path. It is used to force the intermediate node to forward the packet to the node that its address is stored in this field.

Of course, sending or forwarding a data can be done by a source node or an intermediate node. The following is a description of how the source and the intermediate nodes deal with a data packet. But before we start, we need to explain some terms that are used in different contexts.

Newest route is the route that has the highest sequence number.

Shortest route is the route that has the least number of hops.

Best route is the route that has the least number of hops and highest sequence number.

Path selection is always the one with the minimum number of hops. If two paths have the same number of hops, the one with highest sequence number is selected.

- When a source node has ready data to send, it searches for the best route to the destination.

  - If a route is not found, the node queues the packet in its Queue Table (QT) until it gets a route to that destination.
  - If a route is found, the node sets the additional fields of the packet's header as follows, and forwards the packet to the neighbour whose address is stored in the first hop field of the chosen entry.

    current_node = the source node address
    first_node = the first hop field of the chosen entry
    forwarded_node = the second hop field of the chosen entry

- When an intermediate node receives and needs to forward a data packet, it searches for a route to the destination through the node whose address is specified in the *forwarded_node* field. If a route is found, the node forwards the packet to the neighbour whose address is specified in the *first hop* field of the entry (it should be the same address as in the *forwarded_node* field of the data packet header) after modifying two fields of the packet's header as follows:

  current_node = the intermediate node address that is dealing with the packet
  forwarded_node = the second hop field of the chosen entry

  If a route is not found, the intermediate node generates and sends a *Failure Packet* to the source node to stop sending data through this link. Next, the intermediate node searches for an alternative route to send the data packet. The alternative route should not be through the node whose address is in the *current_node* field (this is to avoid fluctuation). If an alternative route is found, the intermediate node simply modifies the two fields of the data packet's header

and forwards the packet using the alternative route. Otherwise, the intermediate node queues the data packet in its ***QT*** until a new route to the desired destination is found.

For more illustration, we present the following example. Figure 5.1 shows a 17 node network, and node **S** needs to send data to node **D**. Node **S** has three routes for node **D**:

S - A - E - H - L - N - D
S - B - F - I - O - D
S - C - G - K - M - P - D

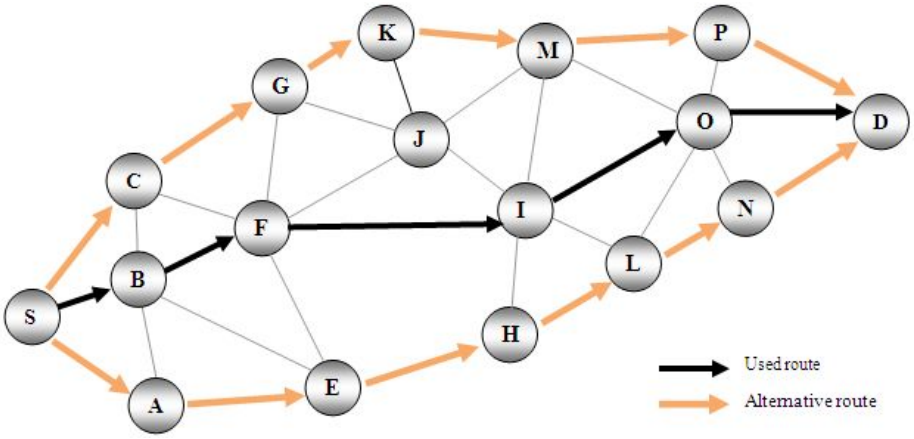

Figure 5.1: Node **S** is sending data through the best route to node **D**

The source node selects the path *S - B - F - I - O - D* because it is the shortest path (5 hops) to send the data. So, node **S** forwards the packet to node **B** after setting the three fields of the data packet's header as follows:

$$current\_node = S, \quad first\_node = B, \quad forwarded\_node = F$$

As node **B** receives the packet, it searches for the path to node **D** through node **F**. Next, it forwards the packet to node **F** after modifying two fields as follows:

$$current\_node = B, \quad forwarded\_node = I$$

Upon receiving the data packet, node **F** searches for the path to node **D** through node **I**, and forwards the packet to node **I** after modifying two fields as follows:

$$current\_node = F, \quad forwarded\_node = O$$

As node **I** receives the packet, it searches for the path to node **D** through node **O**, and forwards the packet to node **O** after modifying two fields as follows:

$$current\_node = I, \quad forwarded\_node = D$$

When node **O** receives the packet, it searches for the path to node **D** through node **D**. Node **O** does not need to modify these fields of the packet's header because the destination is one hop neighbour unless a route is not found.

Suppose that node **I** discovers that the link between itself and node **O** is broken when it tries to forward the data packet (Figure 5.2). In this case, node **I** should do the following:

- Choose one of the alternative routes (the best route) to send the data to node **D**.
- Generate and unicast a *Failure Packet* to the source node (node **S**) to stop using this link. The used route should be the best route, and not through the node whose address is in the *current_node* field (node **F**) to decrease number of collisions.

From figure 5.2, we can notice that node **I** has two alternative routes to node **D** with the same number of hops which are:

I - L - N - D (3 hops)
I - M - P - D (3 hops)

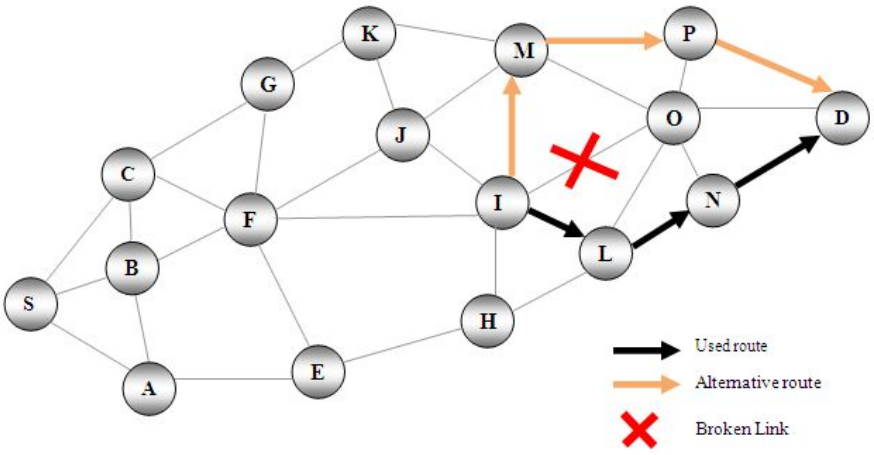

Figure 5.2: Node **I** is using an alternative route to forward data

We assume that the route *I - L - N - D* is the newest. Thus, it is chosen to send the data.

Next node **I** selects the best path to unicast a *Failure Packet* to node **S**. From figure 5.3, it is clear that node **I** has three routes to node **S** which are:

I - F - B - S	(3 hops)
I - H - E - A - S	(4 hops)
I - J - K - G - C - S	(5 hops)

Although the path *I - F - B - S* is the shortest (3 hops), node **I** does not use it because the data packet is received from node **F**, and node **I** has alternative paths that can be used to send the *Failure Packet*. In this case node **I** chooses the path *I - H - E - A - S* because it is the best route (4 hops) to unicast a *Failure Packet* to node **S**.

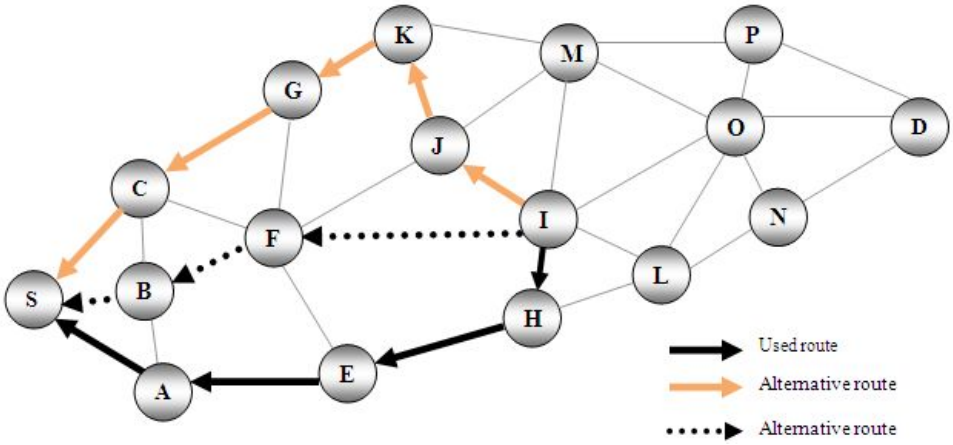

Figure 5.3: Node **I** is unicasting a *Failure Packet* to the source node (Node **S**)

5.4.2 Receiving Data Packets

When a node receives a data packet that is addressed to another destination, it searches for a route through the node that is specified in the *forwarded_node* field in the data packet header. If the route is found, the intermediate node modifies the header fields and forwards the packet to the specified node. Otherwise, the intermediate node does the following:

- Unicasts a *Failure Packet* to the source node asking it to stop sending data through this route anymore.
- Locate an alternative route to forward the received data packet to the desired destination.
  - If an alternative path is found, the intermediate node updates the packet

header fields (*current_node* and *forwarded_node*), and forwards the data packet to the first hop of the alternative path.

– If no alternative path is not found, the intermediate node queues the data packet in its queue until getting a valid route to the desired destination.

Figure 5.4 shows the process that the node follows when receiving a data packet.

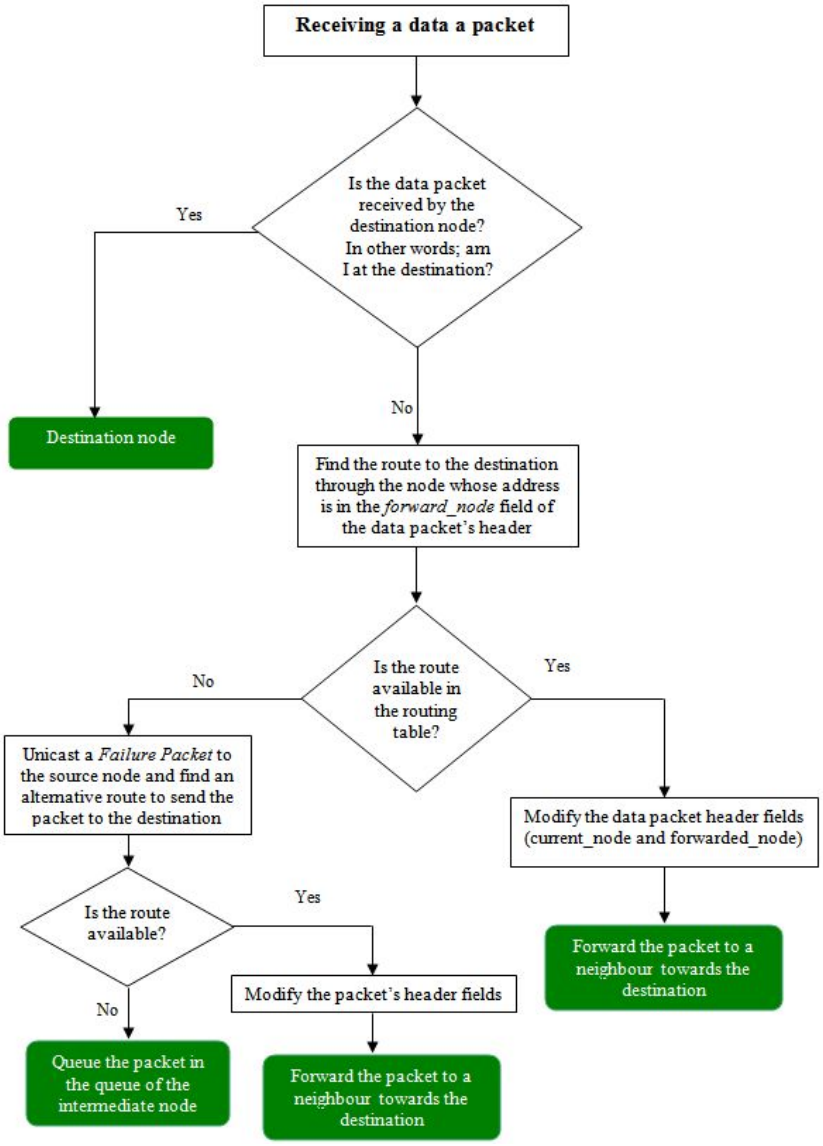

Figure 5.4: Receiving data packets Flowchart

5.4.3 Sending Control Packets

Periodically, each node in the entire network increments its sequence number and checks its neighbours table (NT). If a node has no neighbours (no entries in its NT), it broadcasts a *Hello Message (HM)* containing only the new sequence number. On the other hand, if the node found at least one entry in its NT, it broadcasts an *Update Packet (UP)* which includes the new sequence number and all the entries of the routing table.

Another control packet called *Full Dump (FD)* is used. When a node receives a *Hello Message* or an *Update Packet* from a new neighbour, it responds by unicasting a *Full Dump* of its routing table to the sender. *FD* includes the best route for each destination. By receiving a Full Dump, the receiver node may get enough information about the network.

Another two control packets; *Error Packet (EP)* and *Failure Packet (FP)* are sent by nodes. *EP* is generated and broadcast when a link failure is detected, whereas *FP* is generated and unicast to the source node when an intermediate node fails to forward a data packet to its neighbour.

5.4.4 Receiving Control Packets

When a node receives a control packet, it follows the process in figure 5.6. The node checks the packet's type and processes it as follows.

1. ***Receiving a Hello Message:*** When a node receives a *Hello Message*, it invokes the algorithm in figure 5.5 to deal with the *Hello Message*. The following is a description of the algorithm when node ***R*** receives a *Hello Message* from node ***S***.

- If there is no entry belonging to node ***S*** in ***R***'s NT table (i.e., new neighbour), node ***R*** adds a new entry including the address of node ***S***, Link id (***R-S***), and TimeOut (The neighbour node is considered as unreachable node when the time in the *TimeOut* field has expired). Next, it adds an entry as a direct route for node ***S*** in ***R***'s routing table, and unicasts a Full Dump (FD) to node ***S***. FD includes the best route for each destination.
- If an entry for node ***S*** is available in ***R***'s NT table (i.e., old neighbour), node ***R*** updates the *TimeOut* field of the entry that belongs to node ***S***.

```
01  The receiver node checks its Neighbours Table (NT).
02  If (No entry belongs to the Hello Message  sender in my NT)
03  {
04     Add an entry for the Hello Message sender in my NT.
05     Add an entry as a direct route for the Hello Message sender in my RT.
06     Unicast a Full Dump of my RT to the Hello Message sender.
07  }
08  else
09  {
10     Update the TimeOut field of the entry that belongs to the Hello Message
       sender in my NT.
11     Update the sequence number, Changed at, and TimeToLive fields of the entry that
       belongs to the Hello Message sender in my RT.
12  }
```

Figure 5.5: Receiving a *Hello Message* Algorithm

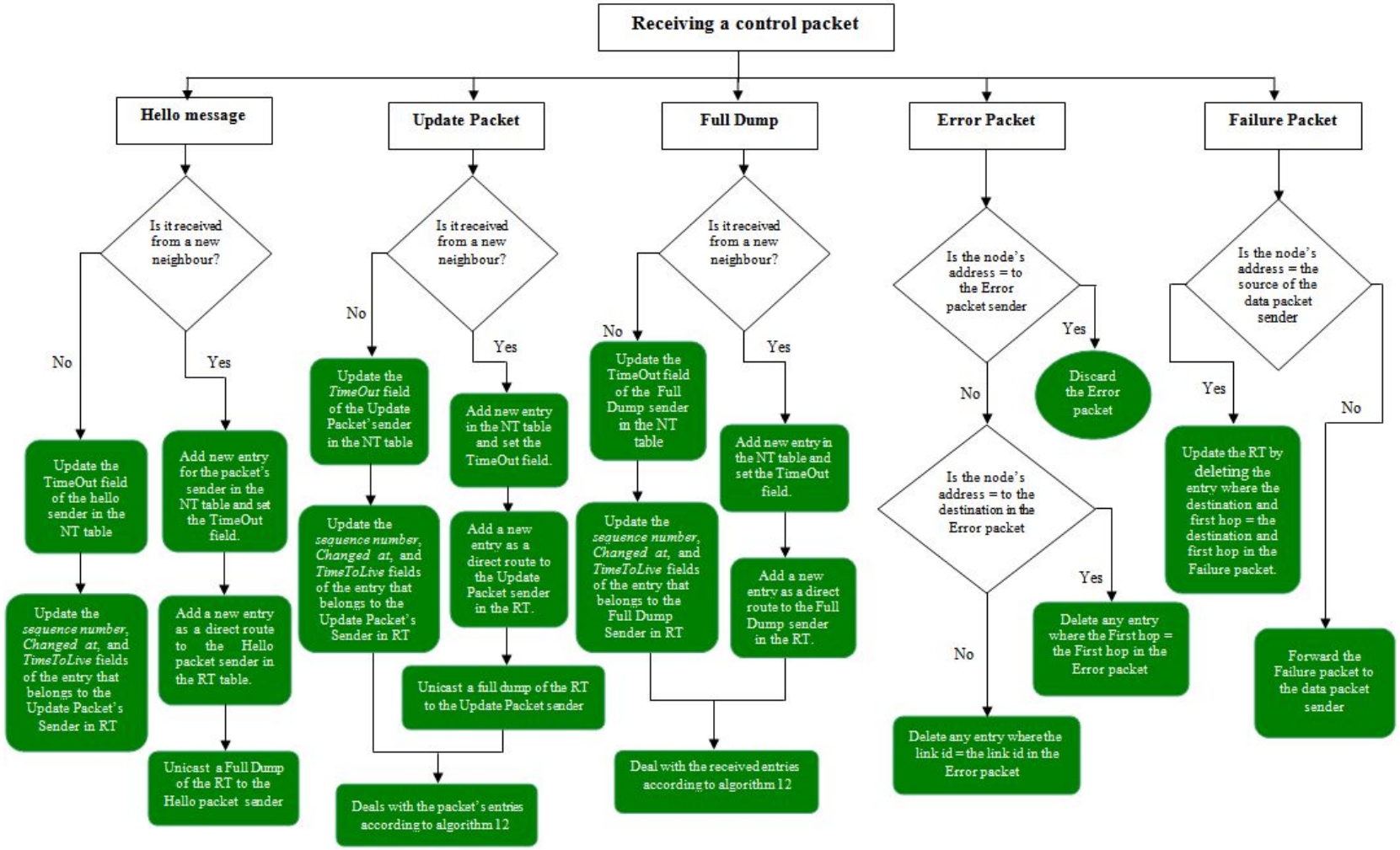

Figure 5.6: Receiving Control Packets Flowchart

2. ***Receiving an Update Packet:*** When a node receives an *Update Packet*, it invokes the algorithms in figures 5.7 and 5.8 respectively to process the packet.

```
01  The receiver node checks whether the Update Packet is received from a new or
    an old neighbour.

02  If (No entry belongs to the Update Packet sender in my NT)
03  {
04      Add a new entry for the Update Packet sender in my NT.
05      Add a new entry as a direct route for the Update Packet sender in my RT.
06      Unicast a Full Dump of my RT to the Update Packet sender.
07  }
08  else
09  {
10      Update the TimeOut field of the entry that belongs to the UP sender in my NT.
11      Update the sequence number, Changed at, and TimeToLive fields of the entry that
        belongs to the Update Packet sender in my RT.
12  }
13  Invoke the algorithm in figure 5.8 to deal with the entries of the UP.


%11      Update the Sequence Number field of the entry that belongs to UP sender.
```

Figure 5.7: Receiving an *Update Packet* Algorithm

As soon as node ***R*** receives an *Update Packet*, it invokes the algorithm in figure 5.7. The following is a description of the algorithm executed at node ***R*** to check whether the packet is received from a new or a known neighbour.

- If the *Update Packet* is received from a new neighbour, node ***R*** does the following:
  - Adds an entry for the *Update Packet* sender in its NT.
  - Adds a new entry as a direct route to the *Update Packet* sender in its RT.
  - Unicasts a *Full Dump* of its RT to the *Update Packet* sender containing the best route for each destination.

- If the *Update Packet* is received from an old neighbour, node ***R*** does the following:
  - Updates the *Timeout* field of the corresponding entry in its NT table.
  - Updates the *sequence number* of the corresponding entry in its RT table.

Next, node ***R*** invokes the algorithm in figure 5.8 to deal with each entry of the *Update Packet*.

The following is a description of the algorithm in figure 5.8 executed at node ***R*** to deal with each entry of the received packet (*Full Dump* or *Update Packet*). We use the following notation:

- my_Id: is the identifier of the node executing the algorithm.
- m.sender: is the packet sender ID.
- m.dst: is the destination field of the received entry.
- m.fh: is the first hop field of the received entry.
- m.sh: is the second hop field of the received entry.
- m.metric: is the number of hops field of the received entry.
- m.ln: is the link id field of the received entry.
- m.SeqNum: is the sequence number field of the received entry.
- DST: is the destination field of an entry in the routing table.
- FH: is the first hop field of an entry in the routing table.
- SH: is the second hop field of an entry in the routing table.
- Metric: is the number of hops field of an entry in the routing table.
- LN: is the link id field of an entry in the routing table.
- SeqNum: is the sequence number field of an entry in the routing table.

Node ***R*** discards the received entry if R's address is equal to m.dst, m.fh or m.sh. If not, node ***R*** modifies the received entry's fields where the *m.sh* field is set to *m.fh* field, the *m.fh* field is set to *m.sender* field, and *m.metric* field is incremented. Then, node ***R*** starts to compare the modified entry with the entries that are already in its routing table as follows:

- If no entries are available for this destination (m.dst), node ***R*** inserts the modified entry in its routing table.
- If the modified entry is not similar (m.fh $\neq$FH and m.sh $\neq$SH and m.ln $\neq$LN) to any entry in the routing table, node ***R*** inserts the modified entry in its routing table.
- If the modified entry is similar to more than one entry, node ***R*** discards the modified entry.
- If the modified entry is similar to only one entry, node ***R*** does one of the following:
  - Overwrites the modified entry if it includes a route that is better than the existing one.
  - Discards the modified entry if it includes a route that is worse than the existing one.

```
01 While (There are more entries in the received Packet)
02 {
03    The receiver node checks its address with the destination, first hop
      and second hop fields of the received entry

04    If (my_Id = m.dst or my_Id = m.fh or my_Id = m.sh)
05       discard the received entry.
06    else
07    {
08       modify the received entry's fields as follows:
09              m.sh = m.fh,  m.fh = m.sender, m.metric++

10       If there is no entry in my routing table to m.dst
11         Insert the modified entry in my routing table.
12       else
13       {
14          If(the modified entry is not similar to any entry in my RT)
15             Insert the modified entry in my routing table.
16          else
17          {
18             If (the modified entry is similar to only one entry in my RT)
19             {
20                 If (m.SeqNum > SeqNum OR (m.SeqNum = SeqNum and m.metric < Metric))
21                    Overwrite with the modified entry.
22                 else
23                    Discard the modified entry.
24             }
25             else
26                Discard the modified entry.
27          }
28       }
29    }
30 }
```

Figure 5.8: Processing a *Full Dump* and an *Update Packet* Algorithm

How do we consider that a route is better than another one? As in DSDV, the route with larger sequence numbers is considered better. But if two routes have the same sequence numbers, the one with smaller metric is considered better.

The following example illustrates how nodes deal with a received *Update Packet*.

Figure 5.9 presents a network of 8 nodes. Node 8 has two neighbours (node 1 and node 5) as shown in table 5.7.

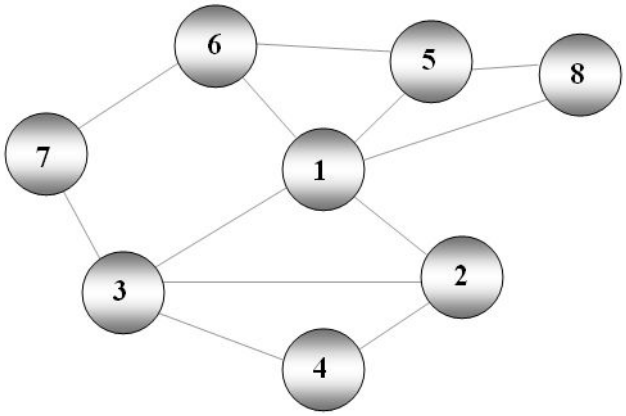

Figure 5.9: Ad Hoc network consists of 8 nodes

Neighbour ID	Link Id	TimeOut
1	8-1	167.3256
5	8-5	171.4325

Table 5.7: Neighbours Table of node 8

Each node periodically broadcasts a *Hello Message* or an *Update Packet* (depends on its NT table) as mentioned in section 5.3. Suppose that node 8 moves towards node 2, and let's assume that node 8 checked its NT table and found that there are two neighbours (node 1 and node 5) as shown in table 5.7. So, node 8 broadcasts an *Update Packet* as shown in figure 5.10. All neighbours (node 1, node 2, and node 5) receive the *Update Packet* and respond as follows:

Because node 1 and node 5 are old neighbours of node 8, they update the *Time-Out* field in the corresponding entry in their neighbours tables. Next, node 1 and node 5 invoke the algorithm in figure 5.8 to deal with the entries of the *Update Packet*.

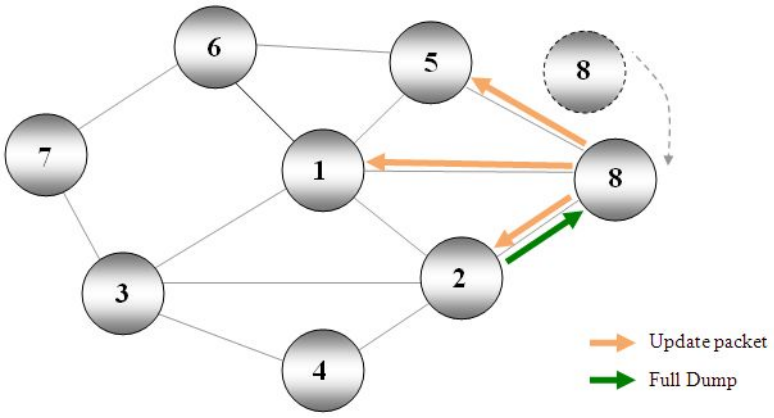

Figure 5.10: Node 8 is broadcasting an *Update Packet* and receiving a *Full Dump*

On the other hand, node 2 considers node 8 as a new neighbour. Thus, node 2 adds node 8 as a neighbour in its NT table and adds an entry as a direct route in its RT. Next, it unicasts a *Full Dump* of its routing table to node 8 as shown in figure 5.10. Then, it invokes the algorithm in figure 5.8 to deal with the entries of the *Update Packet*.

3. ***Receiving a Full Dump:*** When node ***R*** receives a *Full Dump* from node ***S***, it checks if node ***S*** is a new or old neighbour. If node ***S*** is a new node neighbour, node ***R*** adds a new entry for node ***S*** in its NT table and adds a new entry as a direct route for node ***S*** in its RT table. Otherwise, node ***R*** updates the *TimeOut* field of the corresponding entry in R's NT, and updates the *sequence number*, *changed_at*, and *TimeToLive* fields of the corresponding entry in R's RT. Next it invokes the algorithm in figure 5.8 to add, overwrite, or ignore the received entries.

4. ***Receiving an Error Packet:*** As described in chapter 4, any node that discovers a broken link should generate and broadcast an *Error Packet* containing the following fields:

   Packet Sender: the node's address that discovers the broken link.

Destination: the unreachable node address.

Link Id: the link id which is between the node that discovers the broken link and the unreachable node.

The following is a description of the algorithm in figure 5.11 executed at node ***R*** to deal with a received *Error Packet* . We use the following notation:

- my_Id: is the identifier of the node executing the algorithm.
- m.sender: is the *Packet Sender* field in the received packet.
- m.dst: is the *Destination* field in the received packet.
- m.ln: is the *Link Id* field in the received packet.
- FH: is the *first hop* field of an entry in the routing table.
- LN: is the *link id* field of an entry in the routing table.

```
01  The receiver node checks its address with the packet sender (m.sender) field
    in the Error packet.

02  If (my_Id = m.sender)
03  {
04     Discard the Error packet
05  }
06  else
07  {
08     If (my_Id = m.dst)
09     {
10        delete any entry in RT where FH = m.sender
11     }
12     else
13     {
14        delete any entry in RT where LN = m.ln.
15     }
16  }
```

Figure 5.11: Receiving an *Error Packet* Algorithm

When node ***R*** receives an *Error Packet*, it invokes the algorithm in figure 5.11 to do one of the following:

- Discard the *Error Packet* if its address is equal to the address in the *Packet Sender* (m.sender) field of the *Error Packet*. Or
- Delete any entry in its routing table where the first hop (FH) is equal to the packet sender (m.sender) field if R's address is equal to the address in the Destination (m.dst) field of the *Error Packet*. Or
- Delete any entry in its routing table where the link id (LN) is equal to link id (m.ln) of the *Error Packet*.

Our aim is to minimize the large number of control packets that MDSDV0 suffers from. So, one modification is that a receiving node does not need to re-broadcast an *Error Packet*.

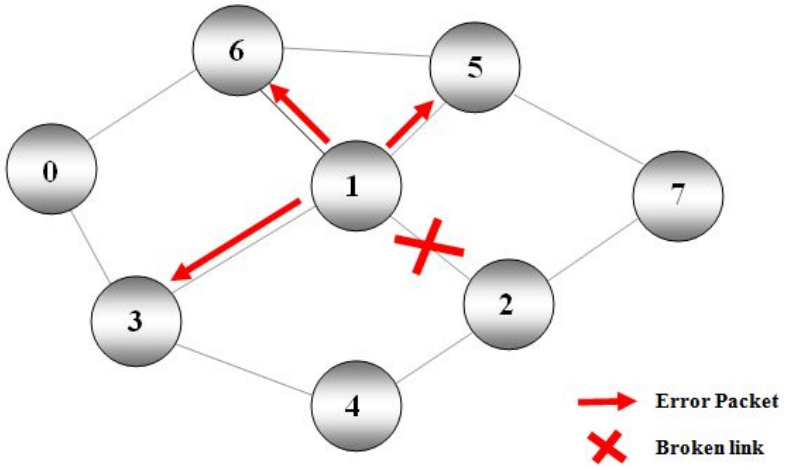

Figure 5.12: Node 1 discovers a broken link and broadcasts an *Error Packet*

For more illustration, we present an example describing the behaviour of each node when receiving an *Error Packet*.

Figure 5.12 presents a network with 8 mobile nodes. When node 1 discovers that the link between itself and node 2 is broken, it does the following:

- Considers node 2 as an unreachable node and deletes the corresponding entry from its NT as shown in Table 5.8.
- Deletes any entry from its routing table where node 2 acts as a first hop as shown in Figure 5.13.
- Generates and broadcasts an *Error Packet* as shown in Table 5.9

Neighbour ID	Link Id	TimeOut
2	1-2	151.3546
3	1-3	169.1256
5	1-5	170.2325
6	1-6	171.4357

Table 5.8: Neighbours table of node 1

dst	Fhop	Shop	metric	link no	change at	seq no.
0	3	0	2	3-0	97.91185681	26
0	6	0	2	6-0	98.37705970	26
1	1	null	0	1-1	0.00000000	34
2	2	null	1	1-2	101.72503623	26
2	3	4	3	4-2	97.91185681	24
2	5	7	3	7-2	101.63430245	24
3	2	4	3	4-3	101.72503623	26
3	3	null	1	1-3	97.91185681	28
3	6	0	3	3-0	98.37705970	26
4	2	4	2	4-2	101.72503623	30
4	3	4	2	4-3	97.91185681	30
5	2	7	3	7-5	101.72503623	28
5	5	null	1	5-1	101.63430245	30
5	6	5	2	6-5	98.37705970	28
6	3	0	3	6-0	97.91185681	26
6	5	6	2	5-6	101.63430245	28
6	6	null	1	1-6	106.30047081	30
7	2	7	2	7-2	101.72503623	30
7	5	7	2	7-5	101.63430245	30

Figure 5.13: Routing table of node 1 at the instance of discovering a broken link

Packet Sender	Destination	Link-Id
1	2	1-2

Table 5.9: Error Packet generated by node 1

All neighbours of node 1 (node 3, 5, and 6) will receive the *Error Packet*. As an example, we describe how node 3 deals with the received *Error Packet*. Because node 3 is not the *Error Packet sender* nor the *Destination*, it deletes any entry where the *link id* is equal to 1-2. So, node 3 deletes the following paths as shown in Figure 5.14:

3 - 1 - 2

3 - 4 - 2 - 1

dst	Fhop	Shop	metric	link no	change at	seq no.
0	0	null	1	3-0	105.91053447	28
0	1	6	3	6-0	104.43117758	26
1	0	6	3	6-1	105.91053447	32
1	1	null	1	3-1	113.45646550	36
1	4	2	3	1-2	105.31930399	32
2	1	2	2	1-2	104.43117758	26
2	4	2	2	4-2	105.31930399	26
3	3	null	0	3-3	0.00000000	30
4	1	2	3	4-2	104.43117758	30
4	4	null	1	4-3	105.31930399	32
5	0	6	3	6-5	105.91053447	28
5	1	5	2	5-1	104.43117758	30
5	4	2	4	7-5	105.31930399	28
6	0	6	2	6-0	105.91053447	28
6	1	6	2	1-6	104.43117758	28
7	1	5	3	7-5	104.43117758	30
7	4	2	3	7-2	105.31930399	30

Figure 5.14: Routing table of node 3 after dealing with the received *Error Packet*

From Figure 5.14, we can notice that node 3 deletes the entries where link id = 1-2 (in red). But it is unable to delete other paths to other destinations that use the link 1-2. For example, node 3 has the path 3 - 1 -2 - 4 (yellow colour). Of course the entry that include this path should be deleted as well, but node 3 can

not delete it because it does not meet any condition of the *Error packet*. For this reason, we use the Failure packet which will be described later, and we use the *TimeToLive* field to delete the route when its time is expired.

5. ***Receiving a Failure Packet:*** Sometimes, the source node uses a path with a broken link to send data. When an intermediate node receives a data packet for forwarding and fails to forward it to an adjacent node, it generates and unicasts a *Failure Packet* to the source. The *Failure Packet* contains the following fields:

   Data Packet Sender: the source node address that is sending the data.

   First Hop: the neighbour node address that the source node used to send the data.

   Destination: the destination node that the source node is communicating with.

   When a node receives a *Failure Packet*, it invokes the algorithm in Figure 5.15. The following is a description of the algorithm executed at node ***R*** to deal with the received *Failure Packet*. We use the following notation:

   - my_Id: is the identifier of the node executing the algorithm.
   - m.sender: is the address in the *Data Packet Sender* field of the *Failure Packet*
   - m.fh: is the address in the *First Hop* field of the *Failure Packet* .
   - m.dst: is the address in the *Destination* field of the *Failure Packet* .
   - FH: is the first hop field of an entry in the routing table.
   - DST: is the destination field of an entry in the routing table.

   When node ***R*** receives a *Failure Packet*, it invokes the algorithm in figure 5.15 to do one of the following:

- If ***R***'s address is equal to the address in the source field (i.e., node ***R*** is the data sender), it deletes the entry where the destination and first hop are the the same as the destination and first hop included in the Failure packet. Then, node ***R*** selects an alternative path to continue sending the data.
- Otherwise (i.e., node ***R*** is not the data sender), it forwards the *Failure Packet* towards the source node.

```
01  The receiver node checks its address with the Data Packet Sender (m.sender) field
    in the Failure Packet.

02  If (my_Id = m.sender)
03  {
04     Delete the entry where DST = m.dst and FH = m.fh
05  }
06  else
07  {
08     Forward the Failure Packet towards the source node.
09  }
```

Figure 5.15: Receiving *Failure Packet* Algorithm

For more illustration, we suppose that node 3 sends data using the path 3 - 1 - 2 - 4 (yellow colour) in Figure 5.14. When node 1 receives the data packet and needs to forward it to node 2, it finds that the link between itself and node 2 is broken. In this case, node 1 generates and unicasts a *Failure Packet* to node 3 (the data packet sender) as shown in Table 5.10. When node 3 receives the *Failure Packet*, it deletes the route for node 4 through node 1 (3 - 1 -2 - 4).

Data Packet Sender	First Hop	Destination
3	**1**	**4**

Table 5.10: Failure Packet generated by node 1

5.5 Disjoint Path Rigorous Argument

Lemma 1:

The *First hop* and *Second hop* of every path in the Routing table (RT) are distinct.

Supporting Argument:

Only *Full Dumps* and *Update Packets* add paths to the routing table, and both use the *Processing a Full Dump and an Update Packet algorithm* (figure 5.8). The conditions in line 11 and line 15 of the algorithm maintain this invariant.

Rigorous Argument that all MDSDV paths are disjoint

Definition 1:

Disjoint paths have a unique *first hop*, *second hop*, and *link id* in the routing table. The argument proceeds by induction over the MDSDV construction of the routing table.

Base case:

All first and second hop paths are disjoint. Follows directly from Lemma 1, and the fact that the Link id is uniquely determined by the final hop.

Induction step:

Induction Hypothesis: All paths in the routing table are disjoint without loss of generality. We consider adding a node n_i to the network with just 2 neighbours n_j and n_k

Argument by Contradiction:

Assume that for some destination (n_m) there is a common node (n_l) in the two new paths from n_i to n_m:

(n_i, n_j, ... n_l.....n_m) and (n_i,n_k, ..., n_l..... n_m)

From Lemma 1, the immediate successors to n_l will maintain only a single path to n_m in the receiving processing a *Full Dump* and an *Update Packet* algorithm (figure 5.8). Moreover as The Algorithm in figure 5.8 is deterministic every successor selects the same path with the same link id.

This contradicts the induction hypothesis.

5.6 Functionality Testing

In the absence of formal mathematical proof of the correctness of the protocol we need an extensive set of tests (verification) to confirm that the protocol meets the intended specifications, is fully functional, and works in a wide range of environments. One of the most common methods for doing this is functionality testing, where the new protocol results are checked to see if they meet expected results based on previously known results from other protocols already tested and verified. Providing a product, for example a routing protocol with bug-free or a minimum amount of issues, is also important and every developer's goal. Therefore, functionality testing helps to achieve such targets.
Once the implementation of MDSDV has been completely tested, and ported to run in NS2, its functionality is verified using the following scenario.

5.6.1 Scenario description

Figure 5.16 presents a simple 5-node wireless scenario that is used in the functionality testing of MDSDV. The topology consists of five mobile nodes which move about within an area whose boundary is defined as 600mX400m.
A mobile node consists of network components like Link Layer (LL), Interface Queue (IfQ), MAC layer, the wireless channel nodes transmit and receive signals from etc. At the beginning of the scenario, we define the type for each of these network components. Additionally, we define other parameters such as the type of antenna, the radio-propagation model, the type of ad-hoc routing protocol used by mobile nodes, etc. See comments in the code for a brief description of each variable defined (Line 4-16).
Next we configure and create mobile nodes (Line 29-45), and give them initial positions to start with (Line 47-61). As nodes are free to move, we produce some node movements in Lines 63-66. We setup traffic flow between node (0) and node (4).

Lines 68-76 set up a TCP connection betwen the two nodes with a TCP source on node (0). Next, we define stop time when the simulation ends (Lines 82-89), and finally start the simulation in (Line 90).

```
1   # wrls1.tcl
2   # A 5-node example for ad-hoc simulation with MDSDV

3   # Define options

4   set val(chan)           Channel/WirelessChannel    ;# channel type
5   set val(prop)           Propagation/TwoRayGround   ;# radio-propagation model
6   set val(netif)          Phy/WirelessPhy            ;# network interface type
7   set val(mac)            Mac/802_11                 ;# MAC type
8   set val(ifq)            Queue/DropTail/PriQueue    ;# interface queue type
9   set val(ll)             LL                         ;# link layer type
10  set val(ant)            Antenna/OmniAntenna        ;# antenna model
11  set val(ifqlen)         60                         ;# max packet in ifq
12  set val(nn)             5                          ;# number of mobilenodes
13  set val(rp)             MDSDV                      ;# routing protocol
14  set val(x)              600                        ;# X dimension of topography
15  set val(y)              400                        ;# Y dimension of topography
16  set val(stop)           150                        ;# time of simulation end

17  set ns   [new Simulator]
18  set tracefd       [open Example.tr w]
19  set windowVsTime2 [open win.tr w]
20  set namtrace      [open Example.nam w]

21  $ns trace-all $tracefd
22  $ns namtrace-all-wireless $namtrace $val(x) $val(y)
23  $ns use-newtrace

24  set topo  [new Topography] # set up topography object
25  $topo load_flatgrid $val(x) $val(y)

26  create-god $val(nn)

27  # Create nn mobilenodes [$val(nn)] and attach them to the channel....
28  # configure the nodes
29        $ns node-config -adhocRouting $val(rp) \
30                        -llType $val(ll) \
31                        -macType $val(mac) \
32                        -ifqType $val(ifq) \
33                        -ifqLen $val(ifqlen) \
34                        -antType $val(ant) \
35                        -propType $val(prop) \
36                        -phyType $val(netif) \
37                        -channelType $val(chan) \
38                        -topoInstance $topo \
39                        -agentTrace ON \
40                        -routerTrace ON \
41                        -macTrace OFF \
42                        -movementTrace ON

43  for {set i 0} {$i < $val(nn) } { incr i } {
44     set node_($i) [$ns node]
45  }

46  # Provide initial location of mobile nodes....
47  $node_(0) set X_ 540.0
48  $node_(0) set Y_ 350.0
49  $node_(0) set Z_ 0.0
```

```
50  $node_(1) set X_ 200.0
51  $node_(1) set Y_ 200.0
52  $node_(1) set Z_ 0.0

53  $node_(2) set X_ 300.0
54  $node_(2) set Y_ 320.0
55  $node_(2) set Z_ 0.0

56  $node_(3) set X_ -10.0
57  $node_(3) set Y_ 180.0
58  $node_(3) set Z_ 0.0

59  $node_(4) set X_ -150.0
60  $node_(4) set Y_ 60.0
61  $node_(4) set Z_ 0.0

62  ## Generation of movements
63  $ns at 2.20 "$node_(0) setdest 450.0 350.0 5.0"
64  $ns at 20.0 "$node_(4) setdest 400.0 80.0 5.0"
65  $ns at 50.20 "$node_(0) setdest 450.0 210.0 5.0"
66  $ns at 100.20 "$node_(2) setdest 100.0 310.0 5.0"

67  # Set a TCP connection between node_(0) and node_(4)
68  set tcp [new Agent/TCP/Newreno]
69  $tcp set class_ 2
70  set sink [new Agent/TCPSink]

71  $ns attach-agent $node_(0) $tcp
72  $ns attach-agent $node_(4) $sink
73  $ns connect $tcp $sink
74  set ftp [new Application/FTP]
75  $ftp attach-agent $tcp
76  $ns at 8.0 "$ftp start"                # start transmitting data

77  # Define node initial position in nam
78  for {set i 0} {$i < $val(nn)} { incr i } {
79  $ns initial_node_pos $node_($i) 25              # 30 defines the node size for nam
80  }

81  # Ending the simulation.......
82  $ns at $val(stop) "stop"
83  $ns at 150.01 "puts \"end simulation\" ; $ns halt"
84  proc stop {} {
85      global ns tracefd namtrace
86      $ns flush-trace
87      close $tracefd
88      close $namtrace
89  }

90  $ns run
```

Figure 5.16: Example1.tcl scenario

5.6.2 Scenario results

In this subsection, we present the reaction of MDSDV to the topology changes during the simulation time. In addition to sending data packets, we present how MDSDV sends and receives control packets to create and update routing tables.

Initial state

At the beginning of this scenario, each participating node is placed in its initial position as shown in Figure 5.17. Each node starts the scenario by creating its routing table with only one entry belonging to itself as shown in Figure 5.18.

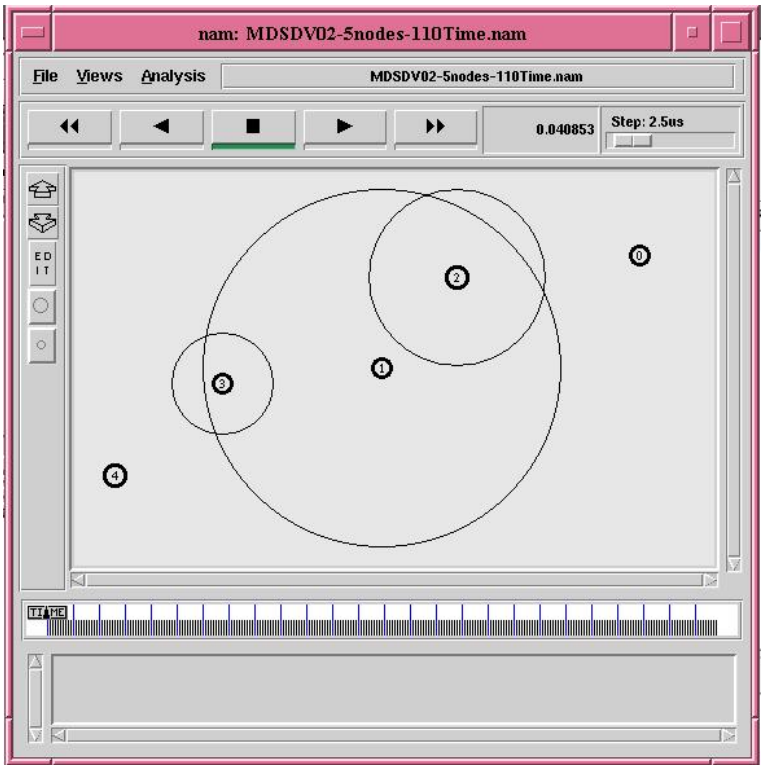

Figure 5.17: A simple network of five nodes

```
At 0  Routing table of node ( 0 )  consists of 1 entry ......
-------------------------------------------------------------------------------
 S_NO.  dst   hop    shop    metric    link no      change at     seq no.
-------------------------------------------------------------------------------
   1     0     0     Null       0          0        0.00000000       0
-------------------------------------------------------------------------------

At 0  Routing table of node ( 1 )  consists of 1 entry ......
-------------------------------------------------------------------------------
 S_NO.  dst   hop    shop    metric    link no      change at     seq no.
-------------------------------------------------------------------------------
   1     1     1     Null       0        10001      0.00000000       0
-------------------------------------------------------------------------------

At 0  Routing table of node ( 2 )  consists of 1 entry ......
-------------------------------------------------------------------------------
 S_NO.  dst   hop    shop    metric    link no      change at     seq no.
-------------------------------------------------------------------------------
   1     2     2     Null       0        20002      0.00000000       0
-------------------------------------------------------------------------------

At 0  Routing table of node ( 3 )  consists of 1 entry ......
-------------------------------------------------------------------------------
 S_NO.  dst   hop    shop    metric    link no      change at     seq no.
-------------------------------------------------------------------------------
   1     3     3     Null       0        30003      0.00000000       0
-------------------------------------------------------------------------------

At 0  Routing table of node ( 4 )  consists of 1 entry ......
-------------------------------------------------------------------------------
 S_NO.  dst   hop    shop    metric    link no      change at     seq no.
-------------------------------------------------------------------------------
   1     4     4     Null       0        40004      0.00000000       0
-------------------------------------------------------------------------------
```

Figure 5.18: Routing tables of all nodes in the network at the beginning of scenario

Sending and receiving *Hello messages* and *Update packets*

As mentioned in Section 5.3, periodically each node checks its Neighbours Table (NT). If the node's NT is empty (no entries), the node generates and broadcasts a Hello Message, otherwise, it generates and broadcasts an Update packet.

Figure 5.19 shows that node one checked its NT at 0.031539 sec, and found no entries (no neighbours). Thus, it broadcasts a *Hello messag*e. Both of node two and node three receive the *Hello message* and consider node one as a new neighbour node. Each of them adds an entry in its routing table as a direct route to node one. Tables 5.11 and 5.12 show routing tables of node two and node three respectively, after receiving the *Hello message* from node one.

At 0.037557 node two checks its NT and finds one entry belongs to node one. Therefore it broadcasts an *Update packet* as shown in Figure 5.19. Because node zero and node one are in the transmission range of node 2, they receive the *Update packet* and update their routing tables as shown in Table 5.13 and Table 5.14.

Similarly, when node three checks its NT and finds one entry belongs to node one, it broadcasts an *Update packet* at 0.039991 as shown in Figure 5.19. Node one and node four are in the transmission range of node three. Thus, they receive the *Update packet* and update their routing tables as shown in Tables 5.15 and Table 5.16.

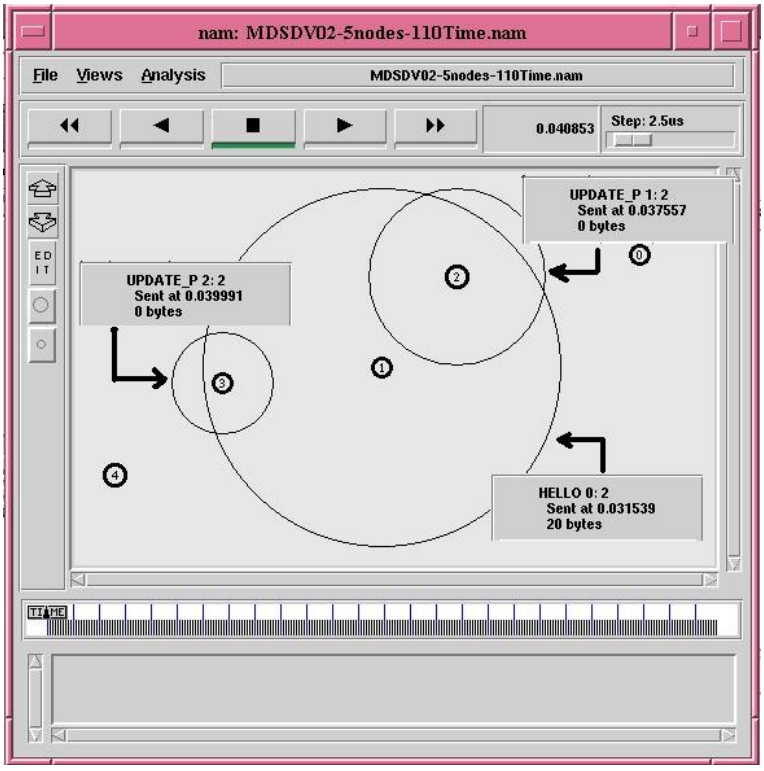

Figure 5.19: Broadcasting *Hello* and *Update* packets

At 0.037557 Routing table of node (2) consists of 2 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	1	1	Null	1	20001	0.03245546	4
2	2	2	Null	0	20002	0.00000000	4

Table 5.11: Routing Table of node 2 after receiving a Hello message from node 1

At 0.039991 Routing table of node (3) consists of 2 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	1	1	Null	1	30001	0.03245564	4
2	3	3	Null	0	30003	0.00000000	4

Table 5.12: Routing Table of node 3 after receiving a Hello message from node 1

At 0.179893 Routing table of node (0) consists of 3 entries after receiving an Update Packet from node 2 .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	0	0	0.00000000	0
2	1	2	1	2	20001	0.17989294	4
3	2	2	Null	1	2	0.17989294	4

Table 5.13: Routing Table of node 0 after receiving an update packet from node 2

At 0.179893 Routing table of node (1) consists of 2 entries after receiving an Update Packet from node 2 .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	1	1	Null	0	10001	0.00000000	4
2	2	2	Null	1	10002	0.17989265	4

Table 5.14: Routing Table of node 1 after receiving an update packet from node 2

At 0.889504 Routing table of node (1) consists of 3 entries after receiving an Update Packet from node 3 .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	1	1	Null	0	10001	0.00000000	4
2	2	2	Null	1	10002	0.17989265	4
3	3	3	Null	1	10003	0.88950414	4

Table 5.15: Routing Table of node 1 after receiving an update packet from node 3

At 0.889504 Routing table of node (4) consists of 3 entries after receiving an Update Packet from node 3 .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	1	3	1	2	30001	0.88950406	4
2	3	3	Null	1	40003	0.88950406	4
3	4	4	Null	0	40004	0.00000000	0

Table 5.16: Routing Table of node 4 after receiving an update packet from node 3

Sending and receiving a Full Dump

As a result of receiving an *Update packet* from a new neighbour (node zero), node four generates and broadcasts a *Full Dump* at 99.0743 sec which contains the best route for each destination.

Figure 5.17 shows that routing table of node four contains seven entries. Specifically, it contains 1 route for node zero, 2 routes for node one, 1 route for node two, 2 routes for node three, and 1 route belongs to itself. So, node four should select one route from the two routes that belong to node one, and one route from the two routes that belong to node three.

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	1	2	3	20000	94.05397148	22
2	1	1	Null	1	40001	94.05397148	26
3	1	3	1	2	30001	89.83882835	24
4	2	1	2	2	10002	94.05397148	24
5	3	1	3	2	10003	94.05397148	24
6	3	3	Null	1	40003	89.83882835	24
7	4	4	Null	0	40004	0.00000000	46

Table 5.17: Routing Table of node 4 contains 7 entries at the time of generating a Full Dump

Entry_NO.	dst	hop	shop	metric	link no	seq no.
(1)	0	1	2	3	20000	22
(2)	1	1	Null	1	40001	26
(3)	2	1	2	2	10002	24
(4)	3	3	Null	1	40003	24
(5)	4	4	Null	0	40004	46

Table 5.18: A Full Dump generated by node 4 contains five entries

Selecting the best route can be briefly described as follows. The node selects the route with the higher sequence number. If two routes have the same sequence number, the route with smaller number of hops is selected. Figure 5.17 shows that the RT of node

four contains two routes for node one (row 2 and row 3), and the route with higher sequence number (row 2) is selected. Meanwhile, the same figure shows that the RT contains two routes for node three (row 5 and row 6), and the route with smaller number of hops (row 6) is selected. Figure 5.18 shows the entries of Full Dump that is generated by node four.

When node 0 receives the Full Dump, the routing information (Figure 5.18) is compared with the information that is already available at the routing table (Figure 5.19) to update the routing table.

As a result of receiving this Full Dump, node 0 creates one extra route for each destination in the network (Yellow colour in Figure 5.20). For example, a new route to node three through node four is obtained (row 7).

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	0	0	0.00000000	26
2	1	2	1	2	20001	94.65602520	26
3	2	2	Null	1	2	94.65602520	26
4	3	2	1	3	10003	94.65602520	24
5	4	2	1	250	30004	53.61783722	22
6	4	2	1	3	10004	94.65602520	44

Table 5.19: Routing Table of node 0 before dealing with the Full Dump

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	0	0	0.00000000	26
2	1	2	1	2	20001	94.65602520	26
3	1	4	1	2	40001	99.19833523	26
4	2	2	Null	1	2	94.65602520	26
5	2	4	1	3	10002	99.19833523	24
6	3	2	1	3	10003	94.65602520	24
7	3	4	3	2	40003	99.19833523	24
8	4	2	1	250	30004	53.61783722	22
9	4	2	1	3	10004	94.65602520	44
10	4	4	Null	1	4	99.19833523	46

Table 5.20: Routing Table of node 0 after dealing with the Full Dump

Broadcasting an Error packet

In MANETs, movement of nodes cause broken links. Any node discovers a broken link should broadcast an Error packet. To describe the broken links, we present the following scenario. Line **$ns at 100.20 "$node_(2) setdest 100.0 310.0 5.0"** produce the movements of node two. It means at time 100.20 sec, node two starts to move towards the destination (x=100,y=310) at a speed of 5m/s. As a result, node two and node four become out of each other's transmission range after some time.

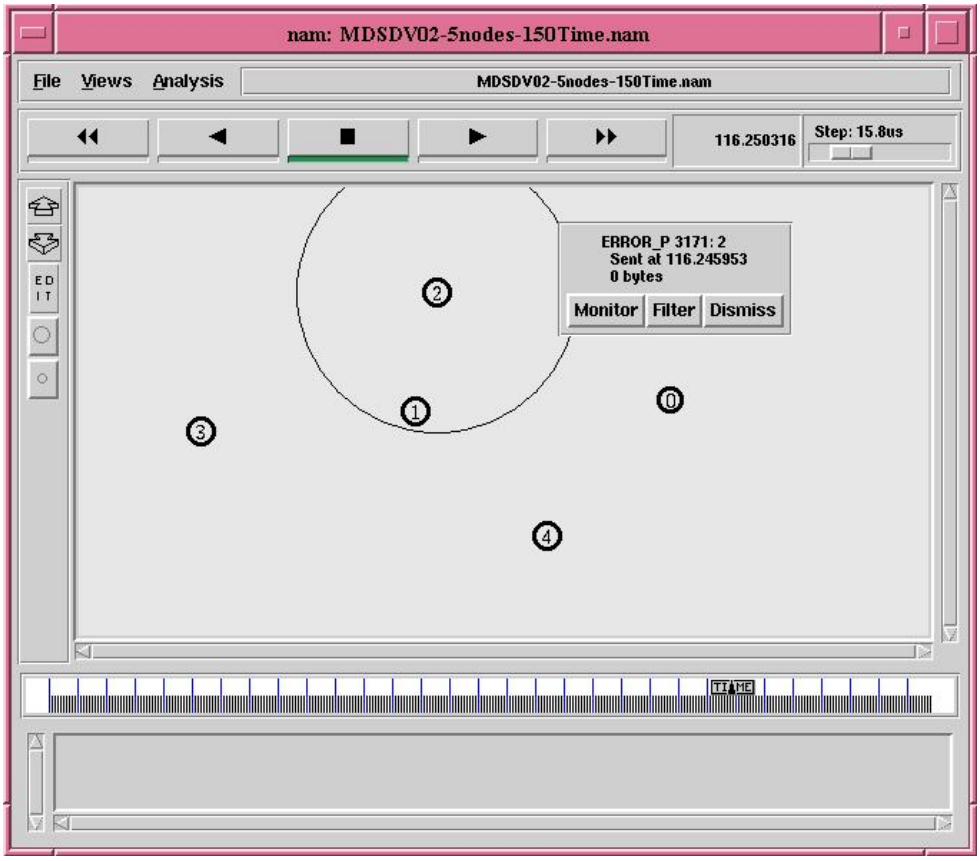

Figure 5.20: Node 2 is bradcasting an *Error packet* at 116.245953 sec

If a node does not receive any packet from its neighbour for a certain time, the node considers the neighbour as unreachable node. Figure 5.20 shows that node two broadcasts an *Error packet* at time 116.245953 sec due to the broken link between itself and node four. Routing table of node two before discovering the broken link is shown in

Figure 5.21. As Node two detects that the link between itself and node four is broken, it assigns ∞ metric for each route where node four acts as a first hop as shown in Figure 5.22. Next, it broadcasts an *Error packet* containing three fields as shown in Figure 5.23. Any node receives the *Error packet* deletes any route that has the same link Id included in the *Error packet*.

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	4	0	2	40000	100.51936627	26
2	1	1	Null	1	20001	112.61354796	30
3	1	4	1	2	40001	100.51936627	26
4	2	2	Null	0	20002	0.00000000	30
5	3	1	3	2	10003	112.61354796	28
6	3	4	3	2	40003	100.51936627	24
7	4	1	4	2	10004	112.61354796	48
8	4	4	Null	1	20004	100.51936627	48

Table 5.21: RT of node 2 before broadcasting an Error packet at 116.245953 sec

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	4	0	∞	40000	116.24595281	0
2	1	1	Null	1	20001	112.61354796	30
3	1	4	1	∞	40001	116.24595281	0
4	2	2	Null	0	20002	0.00000000	30
5	3	1	3	2	10003	112.61354796	28
6	3	4	3	∞	40003	116.24595281	0
7	4	1	4	2	10004	112.61354796	48
8	4	4	Null	∞	20004	116.24595281	0

Table 5.22: RT of node 2 after broadcasting an Error packet at 116.245953 sec

Packet Sender	Destination	Link-Id
2	4	20004

Table 5.23: An *Error Packet* has been sent by node 2

Forwarding data packets

Sending or forwarding a data can be done by a source node or an intermediate node. When a node has a ready data to send, it searches for a route to the destination. If only one route is found, the node uses that route to send data. If more than one route is available in the routing table, the best route is used to send data. If no route is available, the node queues the packet in the interface queue. When a node gets a route to a new destination, it checks the interface queue to see if there are queued packets belonging to the new destination.

In this scenario Lines 68-75 determine the source and destination nodes to send and receive data packets. We assume that node four (Destination) receives any incoming TCP traffic. Therefore, it has a TCP sink agent attached to it. The other node (node zero) has an FTP agent connected to its TCP agent, simulating the FTP traffic source. The FTP traffic (data sending) is started at time 8.0 sec (**$ns at 8.0 "$ftp start"**). However, at this time no route is available at the source to the destination node as shown in Table 5.24. Hence, node zero should queue data packets in the interface queue until a route for node four becomes available.

At 8.0 Routing table of node (0) consists of 3 entries ......

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	0	0	0.00000000	4
2	1	2	1	2	20001	0.88400496	4
3	2	2	Null	1	2	0.88400496	4

Table 5.24: Routing table of no 0 at time 8.0 sec

At 10.4097 node 0 received an Update packet contains 4 entries. The entries are:

Entry_NO.	dst	hop	shop	metric	link no	seq no.
(1)	0	0	Null	1	20000	6
(2)	1	1	Null	1	20001	6
(3)	3	1	3	2	10003	4
(4)	4	1	3	3	30004	4

Table 5.25: At 10.4097 Node 0 receives an update packet from node 2 contains 4 entries

At time 10.4097 sec , node zero receives an *Update packet* (Table 5.25) from node 2 containing 4 entries. Table 5.26 shows that node zero updates its routing table by adding 2 new routes (one for node three and one for node four). As soon as it gets a route for node four, node zero starts to forward the queued packets.

In this scenario, we describe the forwarding of packet no 5 from source (node zero) to destination (node four).

- At 10.4097 node zero selects a route in the fifth entry (4 hops) in Table 5.26 and forwards the packet to node two.
- At 10.4146 node two selects a route in the fifth entry (3 hops) in Table 5.27 and forwards the packet to node one.
- At 10.4198 node one selects a route in the fifth entry (2 hops) in Table 5.28 and forwards the packet to node three.
- At 10.4247 node three selects a route in the fifth entry (1 hop) in Table 5.29 and forwards the packet to node four.

Transmitting data packets from node zero to node four is shown in Figure 5.21.

At 10.4097 Routing table of node (0) consists of 5 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	0	0	0.00000000	6
2	1	2	1	2	20001	10.40968830	6
3	2	2	Null	1	2	10.40968830	6
4	3	2	1	3	10003	10.40968830	4
5	4	2	1	4	30004	10.40968830	4

Table 5.26: Routing table of node 0 after updating

At 10.4146 Routing table of node (2) consists of 5 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	1	20000	8.12430147	6
2	1	1	Null	1	20001	8.09628605	6
3	2	2	Null	0	20002	0.00000000	6
4	3	1	3	2	10003	8.09628605	4
5	4	1	3	3	30004	8.09628605	4

Table 5.27: Routing table of node 2 at the time of forwarding packet number 5

At 10.4198 Routing table of node (1) consists of 5 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	2	0	2	20000	10.40968815	6
2	1	1	Null	0	10001	0.00000000	6
3	2	2	Null	1	10002	10.40968815	6
4	3	3	Null	1	10003	10.40438403	6
5	4	3	4	2	30004	10.40438403	6

Table 5.28: Routing table of node 1 at the time of forwarding packet number 5

At 10.4247 Routing table of node (3) consists of 5 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	1	2	3	20000	8.09628624	4
2	1	1	Null	1	30001	8.09628624	6
3	2	1	2	2	10002	8.09628624	4
4	3	3	Null	0	30003	0.00000000	6
5	4	4	Null	1	30004	10.28001459	6

Table 5.29: Routing table of node 3 at the time of forwarding packet number 5

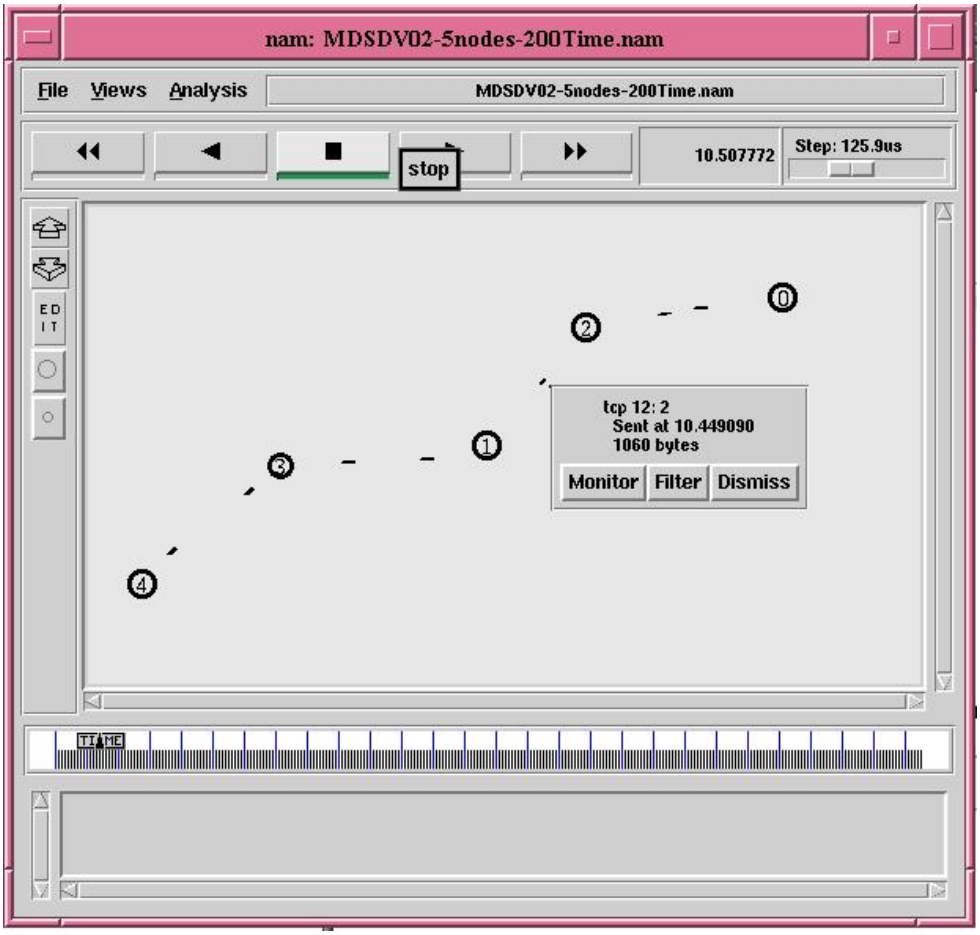

Figure 5.21: Node 0 transmits data packets to node 4

Due to the movement of nodes, routing tables may frequently updated. At time 69.4509 node zero received an Update packet (Table 5.31) from node two. Table 5.30 shows the routing table of node zero at the time of receiving the packet. As a result, node zero updates its routing table (row 2, 3, 4, and 5 in Table 5.32. From Table 5.32 we can see that the route to node four becomes in 3 hops. Node zero continue sending the data through this route as shown in figure 5.22.

At 69.4509 Routing table of node (0) consists of 5 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	0	0	0.00000000	18
2	1	2	1	2	20001	60.09480843	16
3	2	2	Null	1	2	60.09480843	18
4	3	2	1	3	10003	60.09480843	14
5	4	2	1	4	30004	60.09480843	28

Table 5.30: Routing table of node 0 at 69.4509 sec before dealing with the update packet

At 69.4509 node 0 received an Update packet contains 4 entries. The entries are:

Entry_NO.	dst	hop	shop	metric	link no	seq no.
(1)	0	0	Null	1	20000	18
(2)	1	1	Null	1	20001	18
(3)	3	1	3	2	10003	16
(4)	4	1	4	2	10004	32

Table 5.31: An update packet contains 4 entries received by node 0 from node 2 at 9.4509 s

At 69.4509 Routing table of node (0) consists of 5 entries .....

S_NO.	dst	hop	shop	metric	link no	change at	seq no.
1	0	0	Null	0	0	0.00000000	18
2	1	2	1	2	20001	69.45091791	18
3	2	2	Null	1	2	69.45091791	20
4	3	2	1	3	10003	69.45091791	16
5	4	2	1	3	10004	69.45091791	32

Table 5.32: Routing table of node 0 at 69.4509 sec after dealing with the update packet

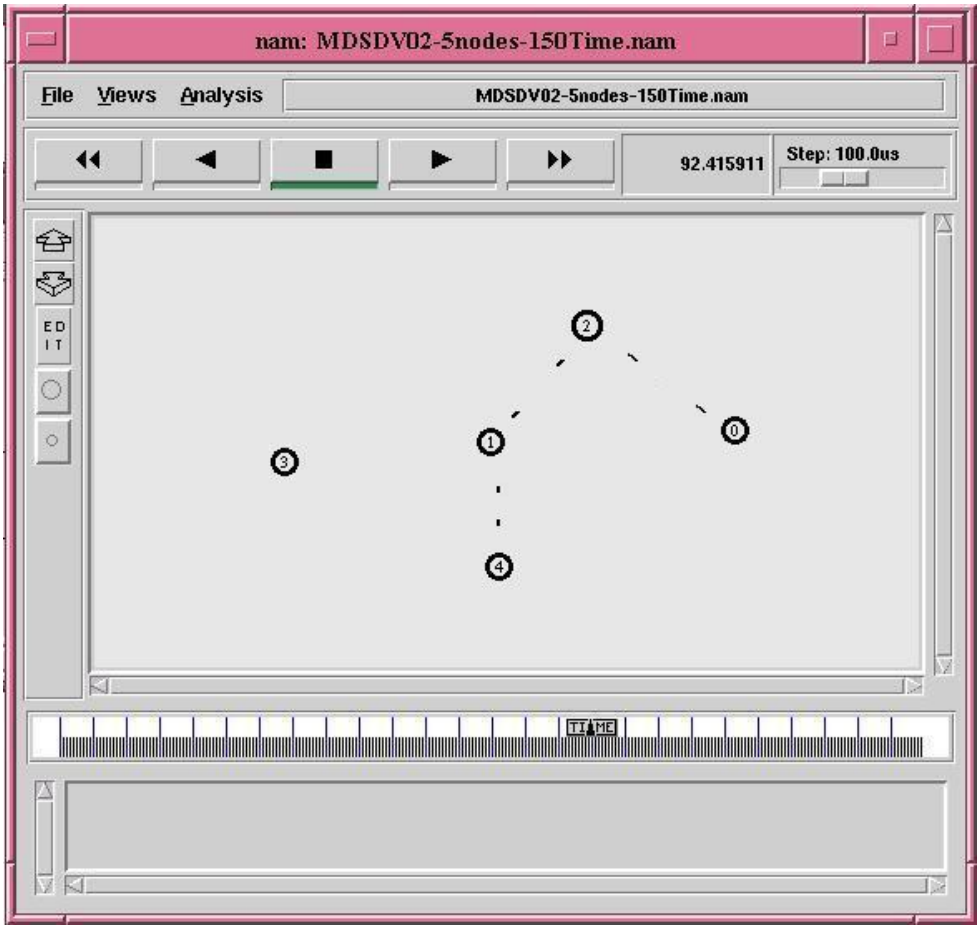

Figure 5.22: Node 0 transmits data packets to node 4 in 3 hops

5.7 Summary

Due to the large number of control packets transmitted by the preliminary version (MDSDV0), we present a revised version in this chapter to reduce the number of control packets and improve the performance in terms of the packet delivery ratio. The final version is referred to as MDSDV.

The main source of control packets is from the *Update packets* and *Error Packets* that are broadcast to the entire network. Thus, MDSDV uses a different mechanism to deal with these two packets. Instead of rebroadcasting Error and Update packets, MDSDV broadcasts them only to its one hop neighbours. In addition, the *Full dump* is unicast only to the new neighbours. When a node receives a control packet from a new neighbour, it responds by unicasting its routing table to that neighbour.
MDSDV only uses a *Hello Message* instead of the *Hello Message* and *Available message* that are used in MDSDV0. A node broadcasts a *Hello Message* only when it has no neighbours (this rarely occurs).
Moreover, MDSDV uses another table called a Queue table to queue packets if a route to the destination is not available.

Appendix A presents the performance comparison between the preliminary version and final version of MDSDV. The results show that MDSDV makes dramatic improvements in delivering data and reducing the control overhead.

Chapter 6

MDSDV Control Overhead

This chapter investigates the overheads, i.e., the control packets generated by MDSDV. Each routing protocol uses a number of control packets and has its own strategy to build routes. MDSDV uses five types of control packets to maintain its routes as discussed in section 5.3. In this chapter, we investigate the control packets that are generated and used by MDSDV. Specifically, we investigate all *control packets* in subsection 6.2.1, *Full Dumps* in subsection 6.2.2, *Update Packets* in subsection 6.2.3, *Error Packets* in subsection 6.2.4, *Hello Messages* in subsection 6.2.5, and *Failure Packets* in subsection 6.2.6. We conducted all our simulation experiments using the Network Simulator NS-2 [33] (version 2.30).

Although most researchers use the number of control packets to measure the overhead, there are other metrics that can be used to measure overhead (e.g., number of control bytes, memory overhead, energy overhead). The total number of control bytes transmitted is one of the metrics. It includes not only the bytes in the routing control packets, but also the bytes in the header of the data packets [73]. The memory overhead can be described as the size in bits of all the data structures used by the routing protocol [98]. The energy overhead is the energy level associated with each transmission and the power spent by receivers. All these measurements show similar behaviours to the number of control packets, so we use that metric.

While this chapter focuses solely on MDSDV overheads, chapter 7 and 8 also report on overheads. Chapter 7 compares the Normalized Routing Load (NRL) of MDSDV and DSDV in a number of scenarios, and Chapter 8 compares the NRL of MDSDV, AODV, and DSR.

6.1 Simulation Environment

Our simulation environment uses similar traffic and mobility models to [26][27][60]. The evaluation is based on the simulation of 50 nodes forming a network over a 670 x 670 square meter area. Nodes move according to the widely used random waypoint model [11][15][63]. In this model, each node begins the simulation by remaining stationary for pause time seconds. It then selects a random destination in the 670m x 670m space and moves to that destination at a speed distributed uniformly between 0 and some maximum speed. Upon reaching the destination, the node pauses again for pause time seconds, selects another destination, and proceeds there as previously described, repeating this behaviour for the duration of the simulation which is 200 seconds. The Distributed Coordination Function (DCF) of IEEE 802.11 [24] for wireless LANs is used as the MAC layer protocol. We fix the number of nodes at 50 nodes where each node has a 250 meter propagation radius. Meanwhile, we varied the pause time and speed of nodes to illustrate the impact of mobility and speed on the number of control packets generated by MDSDV. We run our simulations varying the pause times from 0, 50, 100, 150 and 200 simulated seconds obtaining a range of scenarios that span continuously moving nodes to static ones. We varied the maximum node speed among values 1, 5, 10, 15, 20 and 25 m/s.

The traffic is generated by 10 Constant Bit Rate (CBR) sources spreading the traffic among all nodes. The sending rate was set to 4 packets per second, and the data packet's size was set to 512 bytes. Each data point represents an average of thirty runs with identical traffic models, but different randomly generated mobility scenarios.

Results are based on simulation of 30 runs, and the error bars represent the 95% confidence interval of the mean. Table 6.1 lists the parameters used for the simulations.

Parameter	Value
Simulator	NS-2
Simulation time	200 seconds
Area of the network	670m x 670m
Number of nodes	50 nodes
MAC layer	IEEE 802.11
Transmission range	250 m
Pause time	0 , 50, 100, 150, and 200 seconds
Maximum speed of nodes	1, 5, 10, 15, 20, and 25 m/s.
Mobility model	Random waypoint
Traffic type	CBR (UDP)
Number of data sources	10 Sources
Packet size	512 byte
Transmission rate	4 packets/second
Bandwidth	2 Mb/s
Link failures models	Link failure detection method of MAC layer and TimeOut of beacon packet
Number of runs per data point	30

Table 6.1: Simulation parameters used to evaluate the control packets generated by MDSDV

6.2 Simulation Results

6.2.1 Control Packets

In this subsection, we analyze the total control packets that are generated and transmitted by MDSDV. From Figure 6.1 and Table B.1, we observe that the number of control packets transmitted in static networks (pause time 200) at all speeds is very similar, whereas it increases as the mobility increases.

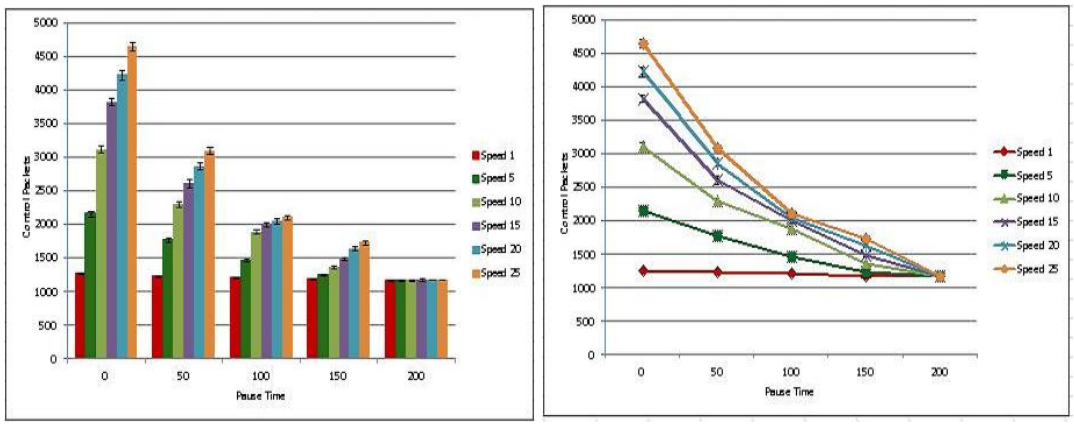

Figure 6.1: Control Packets as a function of Pause Time

Also, the other parameter of mobility (speed of nodes) has an impact on the number of control packets, where the number of control packets increases as the speed increases. From the first row of table B.1, we see that in highly dynamic networks (Pause time 0), MDSDV transmits at low speed (1 m/s) only about 27% of control packets that are transmitted at high speed (25 m/s). This is because nodes move slowly at low speed. As a result, topology changes happen rarely. In other words, nodes do not discover new neighbours frequently and hence do not need to unicast Full Dumps frequently. Also nodes do not discover broken links frequently and hence do not need to broadcast Error Packets frequently. Thus, the number of Full Dumps and Error Packets is reduced. It is interesting that the number of control packets transmitted at

high mobility (Pause time 0) with low speed (1 m/s) is similar to the number of control packets transmitted at low mobility (pause time 200) with high speed (25 m/s).

Moreover, we observe that the 4653 control packets transmitted in high mobility (pause time 0) at high speed (25 m/s) is four times greater than the 1274 control packets transmitted in high mobility (pause time 0) at low speed (1 m/s).

6.2.2 *Full Dumps*

Figure 6.2 and Table B.2 show the number of Full Dumps unicasted during the simulation. When any node receives any type of control packet from a new neighbour, it responds by unicasting a *Full Dump* to that neighbour as described in Chapter 5. We found that the number of Full Dumps is very low and similar in medium mobility (pause time 100 sec) and low mobility (pause time 200 sec) at all speeds, whereas it increases as the mobility increases and the speed increases.

This is because Full Dumps are unicast only when discovering new neighbours. This happens rarely at low mobility and at low speeds, and happens frequently at high mobility.

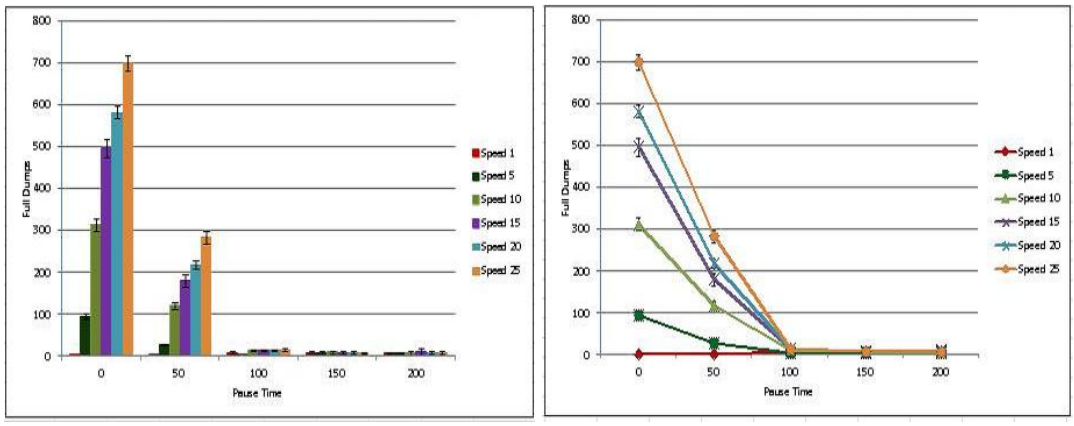

Figure 6.2: Full Dumps as a function of Pause Time

6.2.3 Update Packets

Figure 6.3 and Table B.3 show that neither the mobility nor speed of nodes has an impact on the number of Update Packets. This is because Update Packets in MDSDV are time-triggered only, i.e., there are no event-triggered updates. As a result, the number of Update Packets will be very similar in both dynamic and static networks at all speeds.

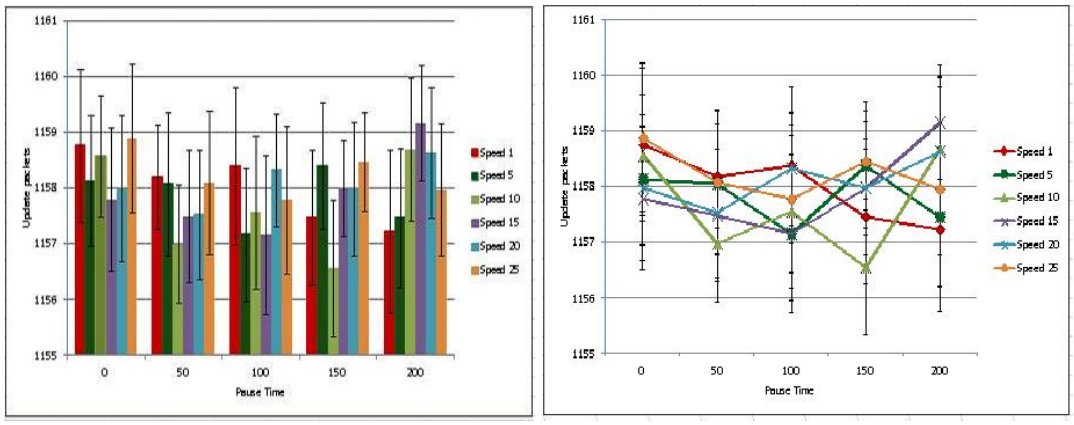

Figure 6.3: Update Packets as a function of Pause Time

6.2.4 Error Packets

Figure 6.4 and Table B.4 show the number of Error Packets that are broadcast during the simulation. As shown in Figure 6.4, number of Error Packets increases as the mobility increases. Also, the number of Error Packets increases as the speed increases. This is because Error Packets are broadcast when broken links are discovered, and the probability of links breaking increases as the mobility increases.

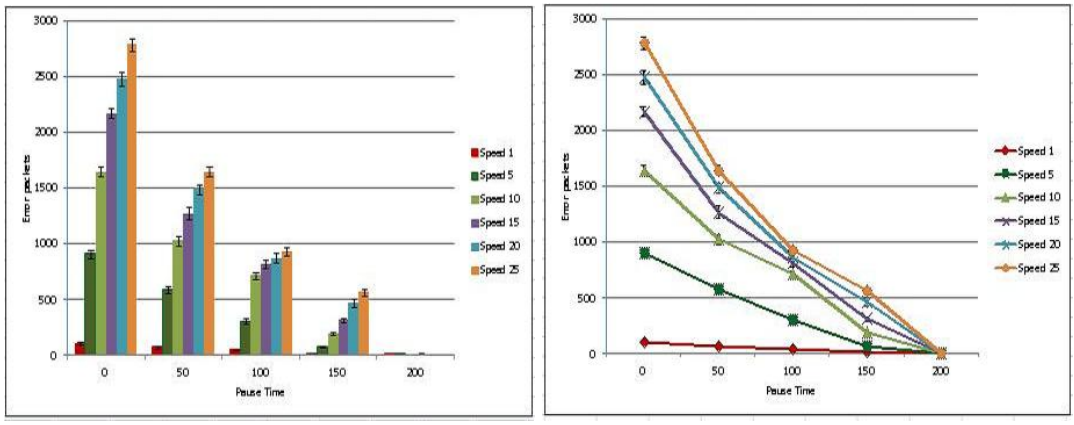

Figure 6.4: Error Packets as a function of Pause Time

6.2.5 Hello Messages

Figure 6.5 and Table B.5 present the number of Hello Messages that are broadcast during the simulation. In general, as shown in Table B.5, we can see that the number of Hello Messages is very small and very similar at all speeds. This is because periodically the node checks its Neighbours Table (NT) to decide whether to broadcast a *Hello Message* or an *Update Packet*, and it broadcasts a *Hello Message* only when it has no neighbours (its NT is empty). Broadcasting Hello Messages usually occurs at the beginning of the simulation time. The NT stays empty until the node receives a control packet from a neighbour node.

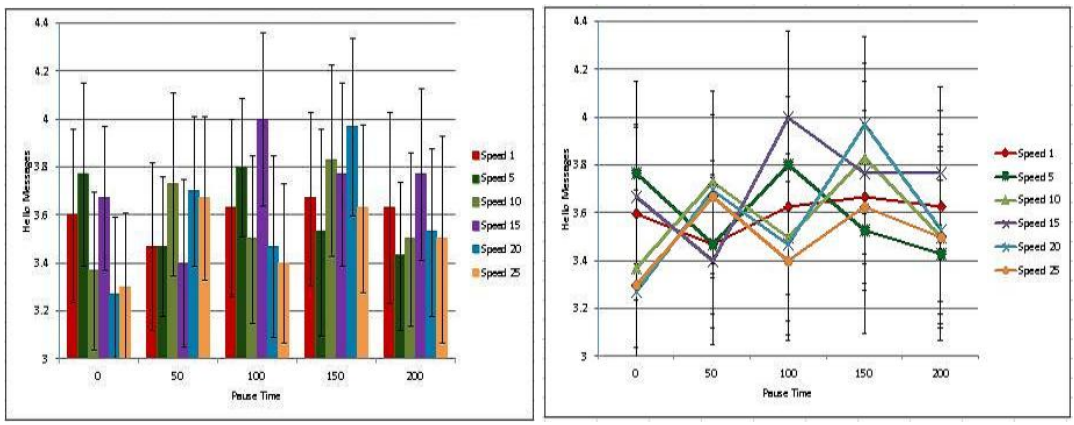

Figure 6.5: Hello Messages as a function of Pause Time

6.2.6 Failure Packets

Figure 6.6 and Table B.6 show the number of Failure Packets that are sent to the source nodes during the simulation. When an intermediate node fails to forward a data packet through the route that has been specified by the source node, it unicasts a *Failure Packet* to the source, as mentioned in section (5.3). As can be seen from Table B.6, the number of Failure Packets is very small at all speeds because the source node usually uses the shortest and newest route to send its data packets.

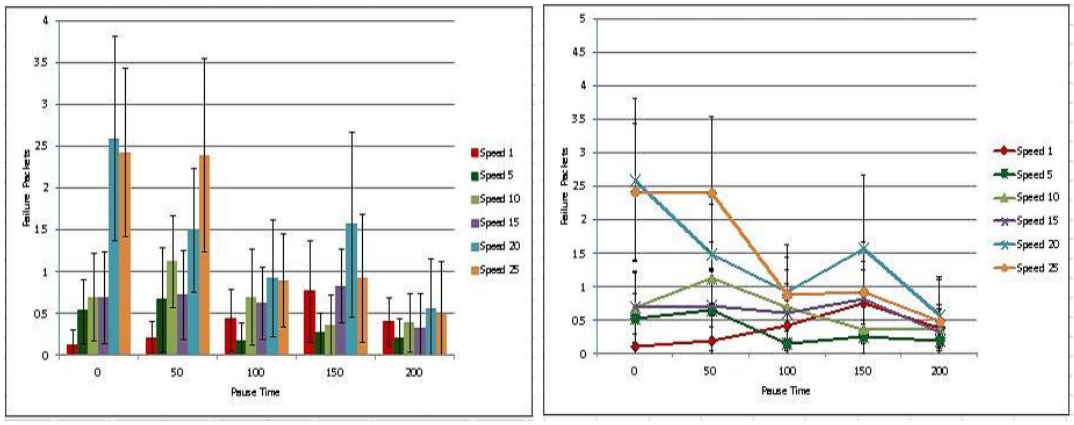

Figure 6.6: Failure Packets as a function of Pause Time

6.3 Summary

We have investigated the control packets generated by MDSDV. We conclude that when the mobility and speed increases, only the number of *Full Dumps* and *Error Packets* increase. This is due to the probability of discovering new neighbours and discovering broken links increases as the mobility and speed increases. On the other hand, the mobility and speed have no impact on the number of *Update Packets* because this kind of packet is broadcast periodically. Also the number of *Failure Packets* is very low because MDSDV usually uses the shortest and newest route. Finally, the number of *Hello messages* is very low because this message is broadcast only when the NT of a node is empty (no records), and the NT is updated whenever any control packet is received.

Chapter 7

Performance Comparison with DSDV

DSDV is a widely used single path routing protocol, and MDSDV adds multiple routes to it, as outlined in Chapter 4. One of our goals was to improve the performance of DSDV. In this chapter, our aim is to demonstrate the improvement of MDSDV over DSDV in terms of a number of metrics.

To obtain fair comparisons among the routing methods, each run of the simulator accepts as input a scenario file that describes the exact motion and the exact sequence of packets originated by each mobile node, together with the time at which each motion change or packet origination occurs. We pregenerate a number of scenario files with varying movement patterns and communication patterns, and then run the routing protocols against each of these scenario files. This chapter is organized as follows: Section 7.1 presents the performance evaluation methodology. Section 7.2 presents the comparison results of our simulations. Finally, Section 7.3 summarises the simulation results.

7.1 Methodology

In this chapter, we use similar traffic and mobility models to [26][27][60]. Nodes in the simulation move according to the random waypoint model [11][15][63], where each node remains stationary for pause time seconds before selecting and moving to a new randomly chosen destination.

We used 10 CBR (Continuous Bit-Rate) flows with 4 packets per second. The source-destination pairs are distributed randomly over the network. Only 512 byte data packets are used as a data packet's size, and the MAC layer protocol is IEEE 802.11. All traffic sessions are established at random times and they stay active until the end of the simulation time.

We have chosen to vary three factors (Network size, Pause Time, and Speed of nodes) to study their impact on the behaviour of the protocols. Thus, our simulation is conducted using three different experiments to compare MDSDV with DSDV.

1. In the first experiment, we tested the impact of network size on the performance of MDSDV and DSDV by varying the number of nodes. 20, 30, 40, 50, 60, 70, 80, 90, and 100 node networks are used.

2. In the second experiment, we ran our simulations with movement patterns generated for 5 different pause times: 0 (Dynamic network), 50, 100, 150, and 200 (static network) seconds.

3. We performed a third experiment to study the effect of the velocity of the nodes on the protocol's performance. We used six different maximum speeds of node movement: 1, 5, 10, 15, 20, and 25 m/s.

Each data point represents an average of 30 runs with identical traffic models, but different randomly generated mobility scenarios. We include error bars on the graphs

which represent 95% confidence interval of the mean. The three experiments use the same simulation parameters that are listed in Table 6.1 with some differences. The differences are listed in tables 7.1, 7.2, and 7.3.

7.1.1 Performance Evaluation Metrics

Several performance metrics are used for evaluation such as Packet Delivery Fraction, Average End-to-End Delay, throughput, Total Packets Received, Normalized Routing Load, Normalized MAC Load, Control Packet Overhead, Packet Loss Percentage, and Route Discovery Frequency. In this chapter, our evaluation is based on four key performance metrics: *Packet Delivery Fraction (PDF)*, *Average End-to-End Delay*, *Normalized Routing Load (NRL)*, and *Data Packets Dropped*. The first three metrics are very widely used [37][97][108]. The first two are the most important metrics for best-effort traffic [12][116]. However, these performance metrics are not completely independent. For example, a shorter delay may not necessarily imply a higher packet delivery fraction, because delay is only measured on the successfully delivered packets. On the other hand, the lower packet delivery fraction and the longer delay may be the reasons for the larger overhead [12].

- **Packet Delivery Fraction (PDF):** This measurement shows the ratio between the number of packets originated by the CBR sources and the number of packets successfully received by the CBR sinks at their target destinations. The PDF shows how a protocol successfully delivers packets from source to destination. The higher PDF give us the better results. It characterizes both the completeness and correctness of the routing protocol. This metric is calculated by dividing the number of packets received by destinations over the number of packets originated from sources.

$$\text{Packet Delivery Fraction (PDF)} = \frac{\sum \text{packets received by destinations}}{\sum \text{packets sent by sources}} \text{ x } 100$$

- **Average End-to-End Delay:** It is the average elapsed time to deliver a packet from source to destination. This metric includes all possible delays caused by queueing at the interface queue, propagation and transfer times, and retransmission delays at the MAC (Medium Access Control) layer.

$$\text{Average End-to-End Delay} = \frac{\sum (\text{packet received time - packet sent time })}{\sum \text{packets received by destinations}}$$

- **Normalized Routing Load (NRL):** It is the number of routing packets transmitted per data packet delivered to the destination. It is calculated by dividing the number of transmitted control packets over the number of data packets received by the destination. This metric is important because it measures the scalability of a protocol, the degree to which it will function in congested or low-bandwidth environments, and its efficiency in terms of consuming node battery power.

$$\text{Normalized Routing Load (NRL)} = \frac{\sum \text{Transmitted Routing Packets}}{\sum \text{packets received by destinations}}$$

- **Data Packets Dropped:** This metric is a measure of data lost by the protocol and includes the data that the source or intermediate nodes drop during the simulation time.

7.2 Simulation Results

7.2.1 Network Size (Varying Number of Nodes)

In this experiment, the comparison of the MDSDV and DSDV routing protocols is performed by varying the network size (varying number of nodes). The number of nodes is set to 20, 30, 40, 50, 60, 70, 80, 90, and 100 nodes. We run our simulations with movement patterns generated for 2 different pause times: 0 and 200 of simulated seconds (0 as a dynamic network and 200 as a static network), whereas we limit the maximum speed of a node to 20 m/s which is a high speed for an ad hoc network, compared to traffic speeds inside a city [102]. Table 7.1 shows the simulation parameters that differ from the baseline parameters given in Table 6.1.

Parameter	Value
Number of nodes	20, 30, 40, 50, 60, 70, 80, 90, and 100 nodes
Pause time	0, 200 seconds
Max. speed of nodes	20 m/s

Table 7.1: Parameters used in the first experiment to compare MDSDV with DSDV

- Packet Delivery Fraction (PDF)

  Figures 7.1 and 7.2 compare MDSDV and DSDV on the basis of their Packet Delivery Fraction (PDF) as a function of Network Size in both a dynamic network and a static network respectively. The figures show that there is a significant difference in the performance of the protocols especially in dynamic networks. MDSDV improves the performance of DSDV by between 27% and 31% in dynamic networks (Figure 7.1), whereas the improvement in performance is between 2% and 3% in static networks (Figure 7.2).

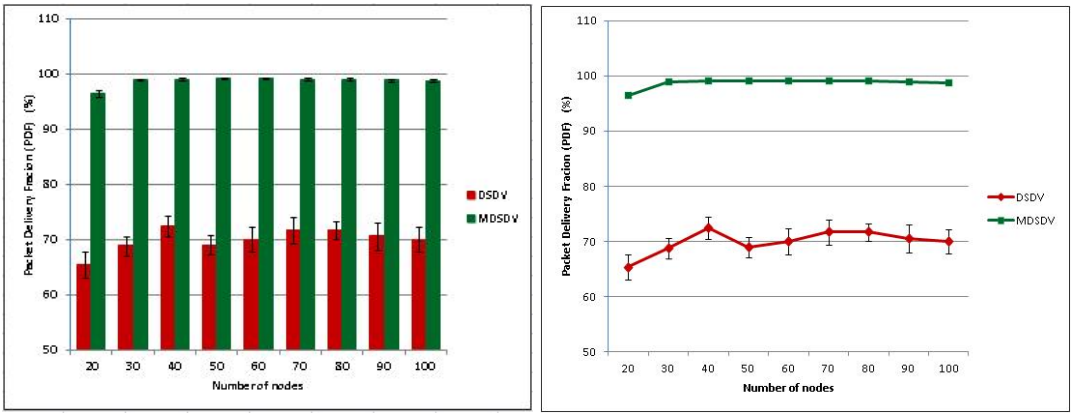

Figure 7.1: PDF vs Number of Nodes (Pause Time 0 sec)

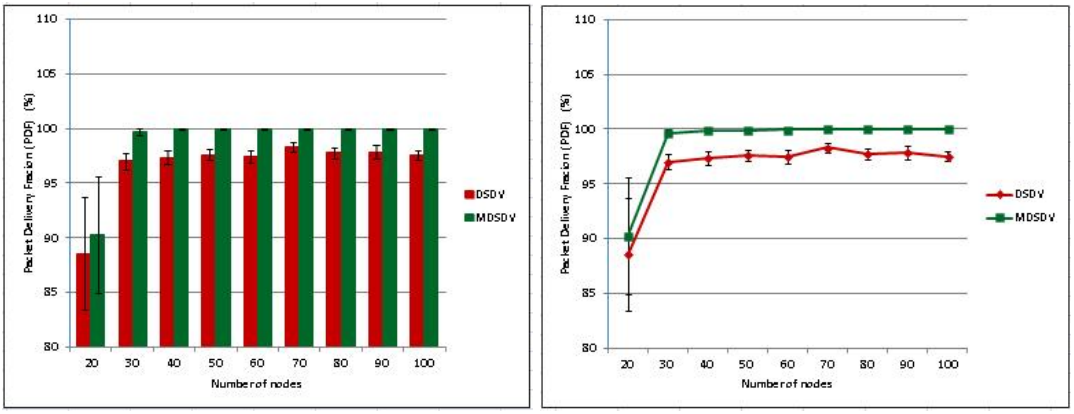

Figure 7.2: PDF vs Number of Nodes (Pause Time 200 sec)

In general, MDSDV improved the performance of DSDV in both dynamic and static networks. The reason for the low PDF of DSDV is due to the fact that it uses the stale routes in the case of broken links [52][70]. In DSDV the existence of a stale route implies that there is a valid route to the destination. In DSDV, the node has to wait until it receives the next update message originated by the destination node to update its routing table entries. On the other hand, MDSDV always uses the newest and shortest route of the alternative routes available in the case of link failure.

- Average End-to-End Delay

  The *Average End-to-End Delays* are shown in Figure 7.3 and Figure 7.4 for dynamic and static networks respectively. From the two figures we can con-

clude that the difference between MDSDV's delay and DSDV's delay is not statistically significant in all cases (except for networks with 100 nodes). In these networks MDSDV exhibits more delay (90%) than DSDV in the dynamic environment (Figure 7.3), and exhibits less delay (84%) in static environment (Figure 7.4).

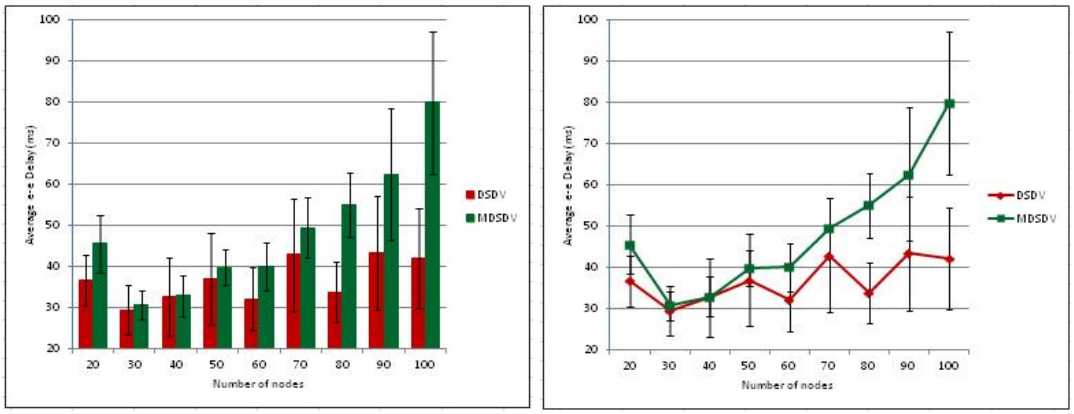

Figure 7.3: Average End to End Delay vs Number of Nodes (Pause Time 0 sec)

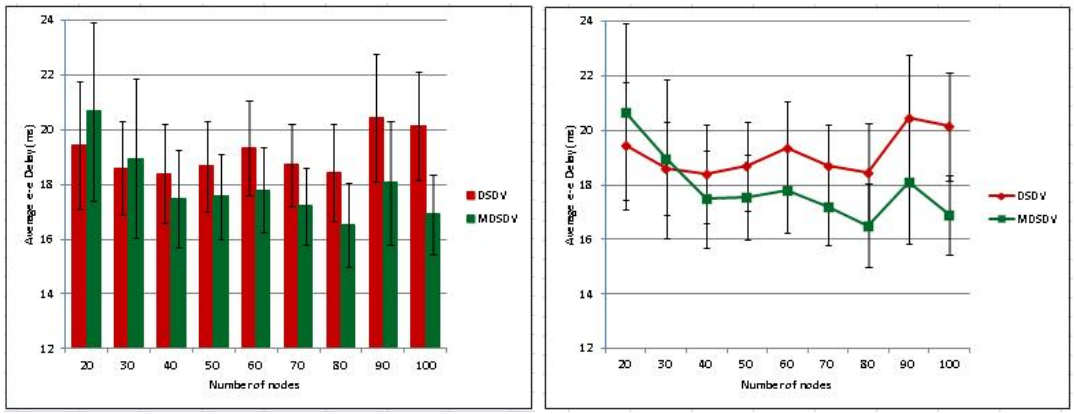

Figure 7.4: Average End to End Delay vs Number of Nodes (Pause Time 200 sec)

- Normalized Routing Load (NRL)

  Figures 7.5 and 7.6 plot the *Normalized Routing Load (NRL)* obtained in our first simulation experiment for dynamic network and static network. The main observation is that MDSDV has a greater NRL than DSDV in dynamic networks, and has a lower NRL in static networks. The figures show that the NRL

grows linearly with the network size for both MDSDV and DSDV. We also found that the difference in NRL increases as the network size increases. Specifically, compared to DSDV, MDSDV increased the overhead by between 43% and 150% in dynamic networks (Figure 7.5), whereas it reduced the overhead by between 1% and 36% in static networks (Figure 7.6).

The main reason for the increase of overhead in dynamic networks using MDSDV is that the unicasting of Full Dumps and the broadcasting of Error Packets may happen frequently. Note that in dynamic networks nodes become neighbours and go out of transmission range of each other frequently.

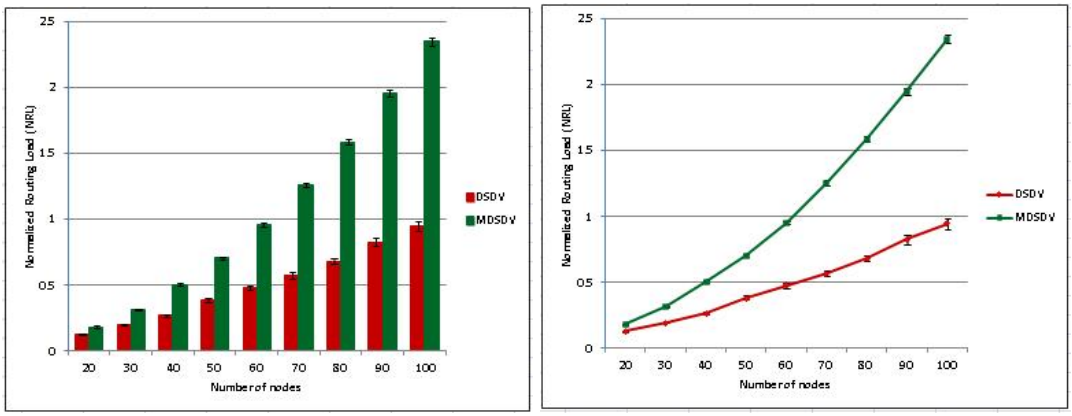

Figure 7.5: NRL vs Number of Nodes (Pause Time 0 sec)

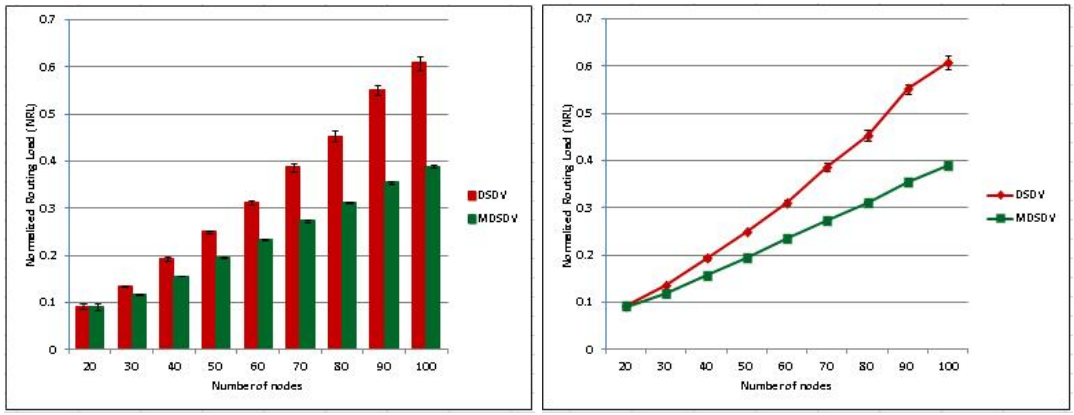

Figure 7.6: NRL vs Number of Nodes (Pause Time 200 sec)

- Data Packets Dropped

  Figures 7.7 and 7.8 show a comparison between the MDSDV and DSDV routing protocols in terms of the *Data Packets Dropped* by each protocol. We can be confident that MDSDV drops less data packets than DSDV in all cases. The only exceptions is 20 node static network where the two protocols drop similar data packets. This is because of the few neighbours that can provide multiple paths for each destination.

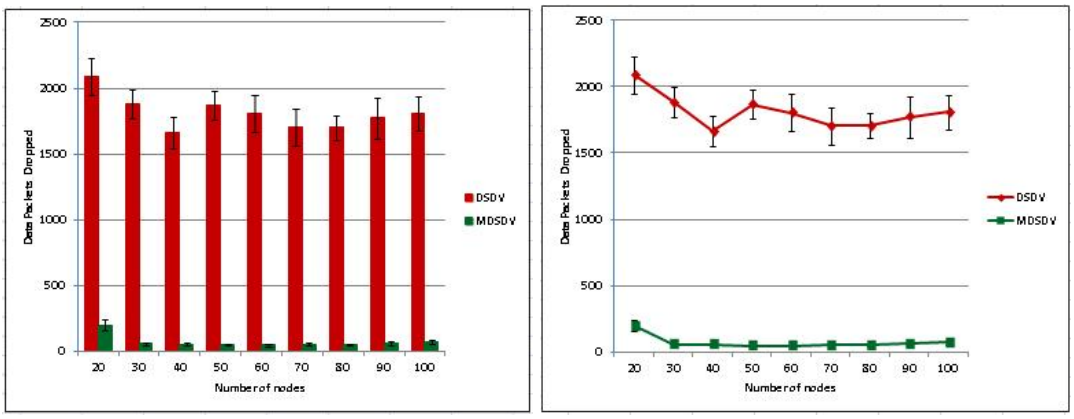

Figure 7.7: Data Dropped vs Number of Nodes (Pause Time 0 sec)

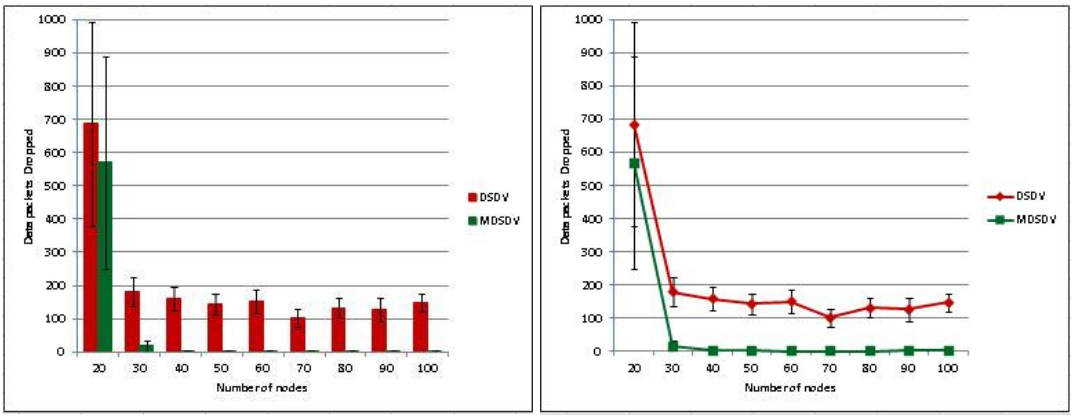

Figure 7.8: Data Dropped vs Number of Nodes (Pause Time 200 sec)

In a dynamic environment (Figure 7.7) we observe that MDSDV dropped 10% of the data packets dropped by DSDV in 20 node networks. Whereas, it dropped about 4% of the data packets dropped by DSDV in networks with more than

20 nodes. On the other hand, in a static environment (Figure 7.8), MDSDV dropped about 83% of data packets dropped by DSDV in 20 node networks. Whereas, it dropped between 1% and 10% of data packets dropped by DSDV in networks with more than 20 nodes.

Our experiments show that the number of data packets dropped by MDSDV is very similar in all networks except a network of 20 nodes (i.e, one with a low node density).

This is because in low node density networks, the probability of getting multiple paths is low. In both cases (dynamic and static networks), the simulation results show that compared with DSDV, MDSDV can adapt more quickly to frequent topology changes in MANETs, and reduce the number of dropped data packets due to link breakage. This is because DSDV waits for a period of time to get new information, and if no route is available and a node plans to transmit data, the node has to queue the packets. Alternatively, the packets will be dropped if the queue is full. In contrast, packet drops are fewer with MDSDV as alternate routes may be used in response to link failures.

7.2.2 Mobility (Varying Pause Time)

This experiment simulates 50 mobile nodes forming an ad hoc network, moving over a rectangular 670x670 flat space. To study the impact of mobility on performance, we choose to vary the pause time. The pause time is varied as 0 (high mobility), 50, 100, 150, and 200 seconds (no mobility). Two maximum speeds 1 m/sec (low speed) and 20 m/sec (high speed) are used. Table 7.2 shows the simulation parameters that differ from the baseline parameters given in Table 6.1. The same four metrics of the previous subsection are measured and compared.

Parameter	Value
Number of nodes	50 nodes
Pause time	0, 50, 100, 150, and 200 seconds
Max. speed of nodes	1 and 20 m/s

Table 7.2: Parameters used in the second experiment to compare MDSDV with DSDV

- Packet Delivery Fraction (PDF)

  Figures 7.9 and 7.10 show the packet delivery fractions of MDSDV and DSDV as a function of pause time in a low speed network and a high speed network respectively. We can confidently conclude that MDSDV outperforms DSDV in all cases specially in dynamic environment.

  The figures show that MDSDV improves the performance of DSDV by between 2% and 5% in low speed networks (Figure 7.9), and between 2% and 30% in high speed networks (Figure 7.10). Moreover, we found that the difference in performance increases as the pause time decreases. This is because as the mobility becomes low, nodes become more stationary which leads to more stable paths from source to destination nodes. MDSDV improved the PDF since it uses an alternative path to destination when a broken link occurs.

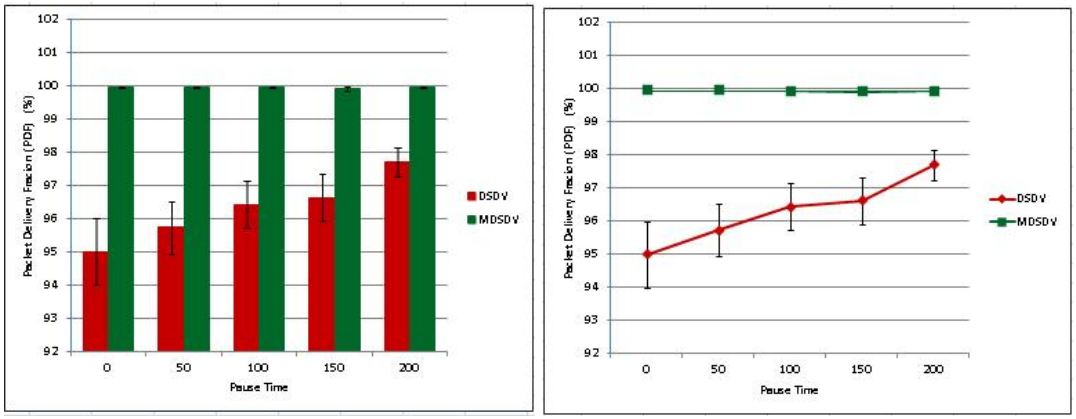

Figure 7.9: PDF vs Pause Time (Low speed: 1 m/sec)

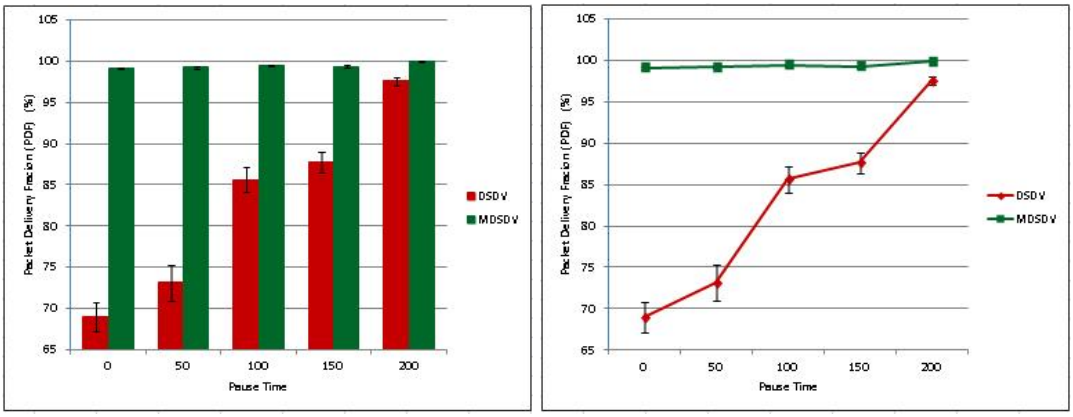

Figure 7.10: PDF vs Pause Time (High speed: 20 m/sec)

- Average End-to-End Delay

  Figures 7.11 and 7.12 show the average End-to-End delays in both low speed and high speed networks respectively. From the figures we can confidently conclude that the difference between MDSDV's delay and DSDV's delay is not statistically significant in all cases

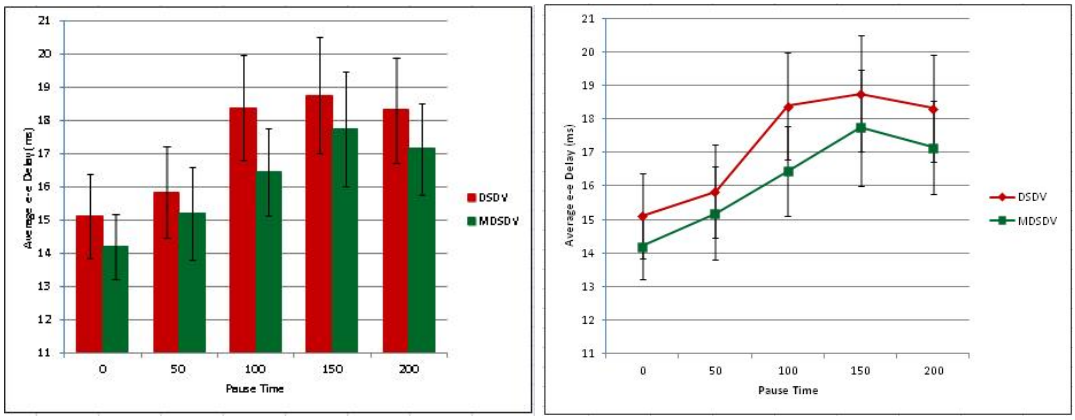

Figure 7.11: Average End-to-End Delay vs Pause Time (Low speed: 1 m/sec)

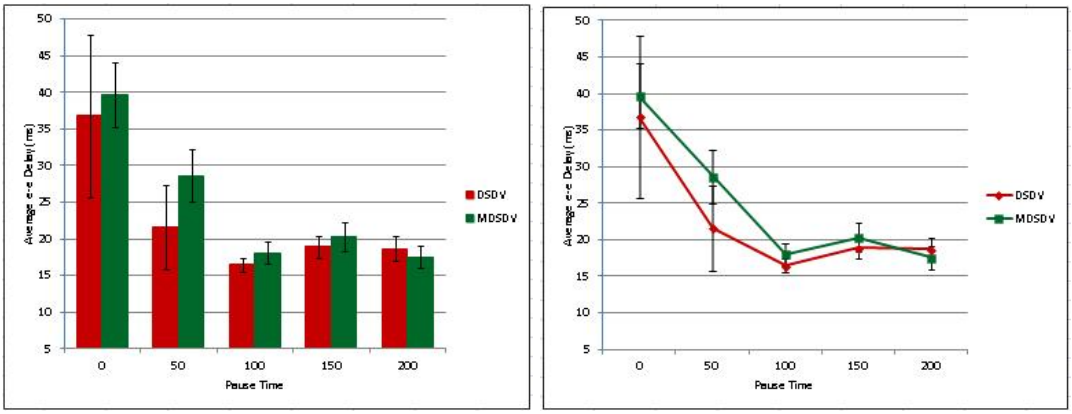

Figure 7.12: Average End-to-End Delay vs Pause Time (High speed: 20 m/sec)

- Normalized Routing Load (NRL)

  Figure 7.13 shows that in a low speed network (speed = 1 m/s) at all pause times, MDSDV has an improvement over DSDV in terms of *Normalized Routing Load* of between 19% and 23%.

  On the other hand, Figure 7.14 shows that in a high speed network (speed = 20 m/s), MDSDV and DSDV seem to compete with each other. Both protocols have regions where they outperform the other protocol, and neither protocol is uniformly better. Specifically, MDSDV has less routing overhead than DSDV in low mobility networks (pause time is greater than 100 sec) by between 13% and 22%, whereas it has higher routing overhead in high mobilty networks (pause time is less than 150 sec) by between 6% and 83%.

  MDSDV is worse than DSDV in the case of high mobility and high speed due to the large number of Full Dumps and Error Packets that are sent by MDSDV, because nodes frequently become neighbours and go out of the range of each other.

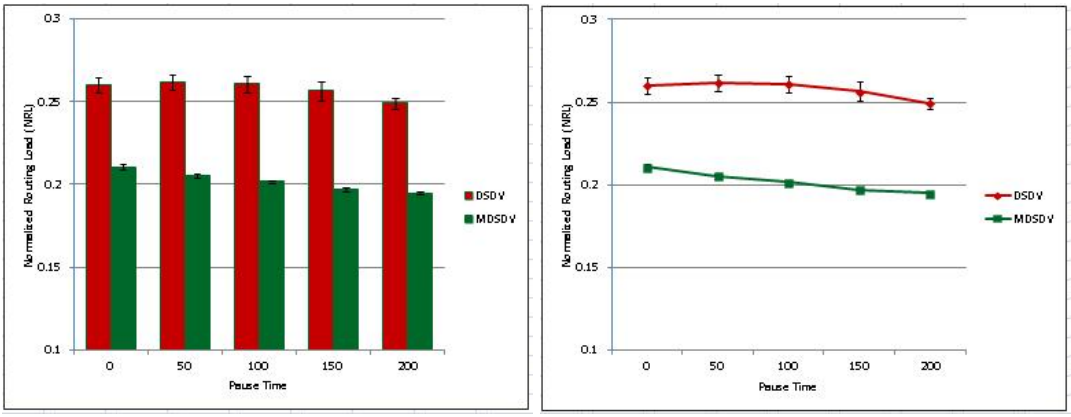

Figure 7.13: NRL vs Pause Time (Low speed: 1 m/sec)

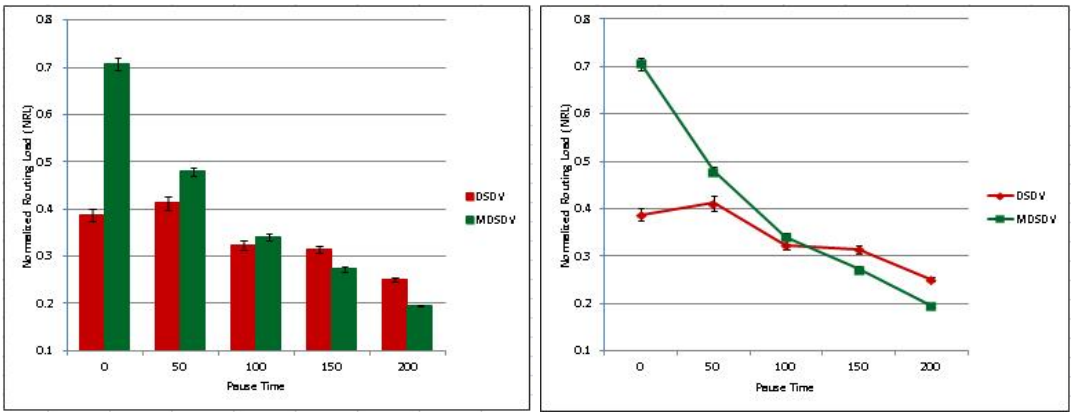

Figure 7.14: NRL vs Pause Time (High speed: 20 m/sec)

- Data Packets Dropped

  Figures 7.15 and 7.16 show the *Data Packets Dropped* in terms of Pause time for both low speed and high speed networks.

  From the figures, we observe that in a high speed network (speed = 20 m/sec), the mobility has a very slight impact on the performance of MDSDV in terms of *Data Packets Dropped*, whereas it has no impact in low speed networks (speed = 1 m/sec). On the other hand, DSDV is impacted by the mobility in both networks (low and high speed networks). The number of data packets dropped by DSDV dramatically increases as the mobility increases.

From the figures, we can confidently conclude that there is a significant difference between MDSDV and DSDV in terms of *Data Packets Dropped*. Specifically, the difference grows from a factor of 68 to a factor of 150 in low speed networks (Figure 7.15), whereas it grows from a factor of 22 to a factor of 38 in high speed networks (Figure 7.16).

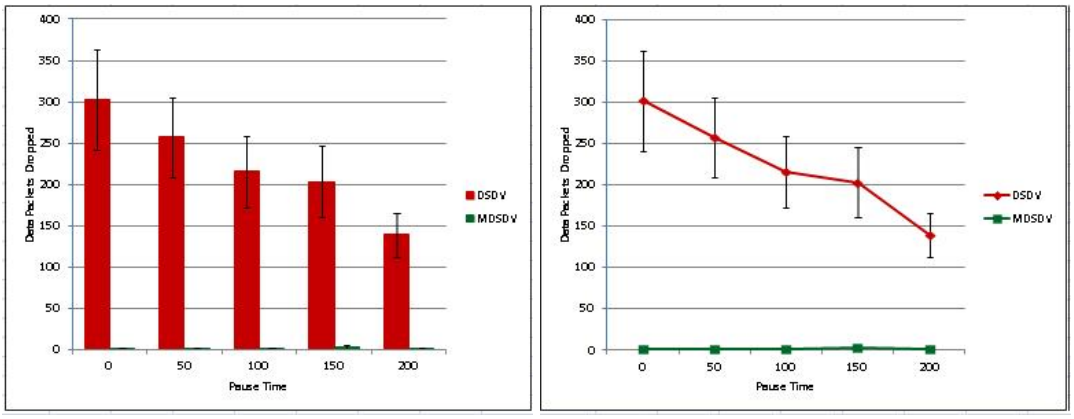

Figure 7.15: Data Dropped vs Pause Time (Low speed: 1 m/sec)

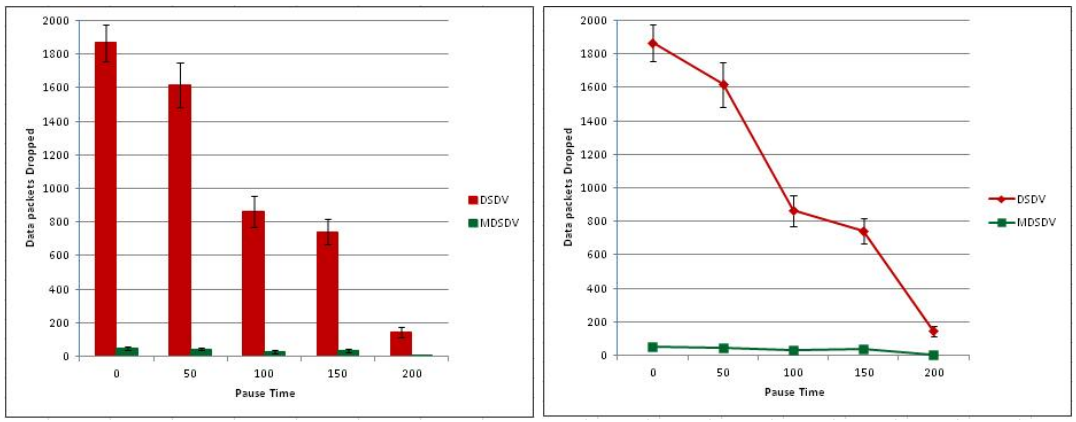

Figure 7.16: Data Dropped vs Pause Time (High speed: 20 m/sec)

7.2.3 Mobility (Varying Speed of Nodes)

This experiment studies the behaviour of MDSDV and DSDV in the case of changing mobility by varying the maximum speed of nodes. Mobility is increased by increasing

the speed of nodes. In this experiment, we use 6 different speeds; 1 m/sec (low speed), 5, 10, 15, 20, and 25 m/sec (high speed). We ran our simulations with movement patterns generated for 2 different pause times: 0 and 200 seconds. A pause time of 0 seconds corresponds to continuous motion, and a pause time of 200 (the length of the simulation) corresponds to no motion. Simulations last for 200 seconds and a 50 node network with terrain area (670m x 670m) is used for this experiment. Table 7.3 shows the simulation parameters that differ from the baseline parameters given in Table 6.1. The same four metrics of the previous subsection are measured and compared.

Parameter	Value
Number of nodes	50 nodes
Pause time	0 and 200 seconds
Max. speed of nodes	1, 5, 10, 15, 20, and 25 m/s

Table 7.3: Parameters used in the third experiment to compare MDSDV with DSDV

- Packet Delivery Fraction (PDF)

  The comparative results of *Packet Delivery Fraction* for dynamic networks and static networks are shown in figure 7.17 and figure 7.18 respectively.

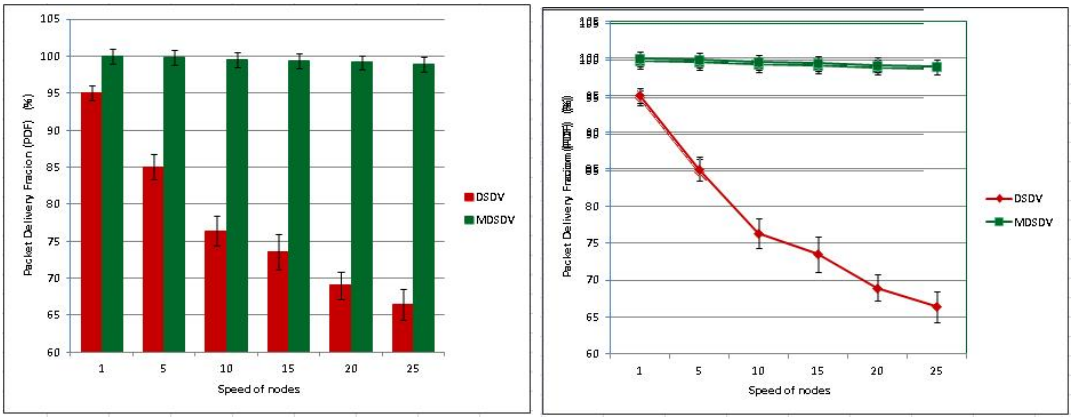

Figure 7.17: PDF vs Speed of Nodes (Dynamic network: Pause Time = 0 sec)

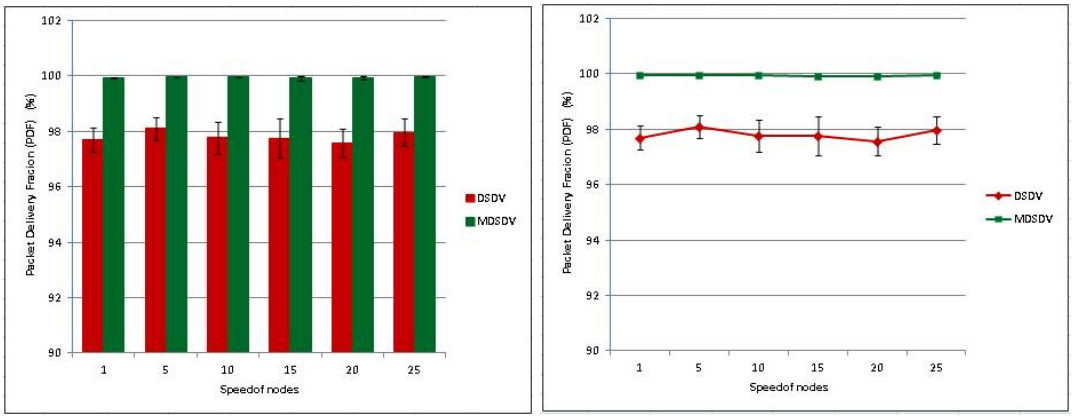

Figure 7.18: PDF vs Speed of Nodes (Static network: Pause Time = 200 sec)

An interesting observation is that the performance of MDSDV stabilized at nearly 100% in both networks (dynamic and static networks), whereas the performance of DSDV dramatically dropped in the dynamic network when the speed of nodes increases.

Based on Figure 7.17, the performance of DSDV dropped from 95% to 66%. The poor performance of DSDV could be caused by the frequent update control packets as the speed of nodes increases. Although, MDSDV improves the performs of DSDV by only about 2% at all speeds in static networks (Figure 7.18), the improvement increases up to 32% in dynamic networks (Figure 7.17). The difference in performance increases as the speed increases.

From the figures, we deduce that the mechanism of backup routes used by MDSDV is still helpful to the performance of data packet delivery in high mobility even though the number of control packets used rises.

- Average End-to-End Delay

  Figure 7.19 and Figure 7.20 plot the *Average End-to-End Delays* for dynamic and static networks as a function of speed. The two figures show that MDSDV and DSDV provide similar delay and the difference between MDSDV's delay and DSDV's delay is not statistically significant in all cases.

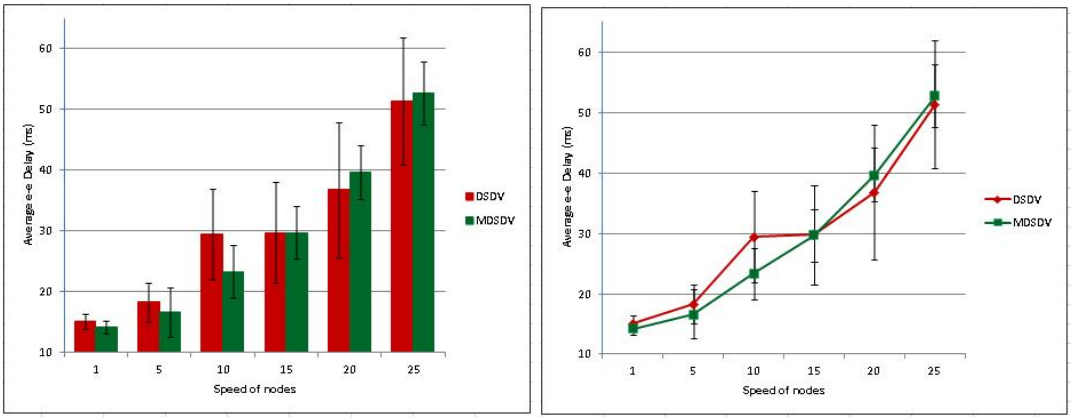

Figure 7.19: Average End to End Delay vs Speed of Nodes (Dynamic network)

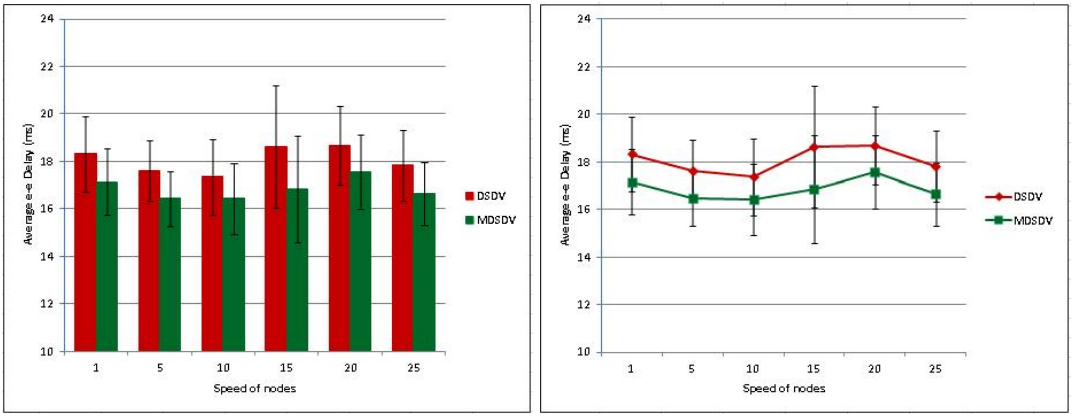

Figure 7.20: Average End to End Delay vs Speed of Nodes (Static network)

- Normalized Routing Load (NRL)

  The simulated results of *Normalized Routing Load* of DSDV and MDSDV are shown in figures 7.21 and 7.22 for dynamic and static networks respectively.

  Figure 7.21 shows that MDSDV demonstrates higher normalized routing load than DSDV in dynamic networks where the node speed is more than 1 m/sec. It has more routing load by between 21% and 100%. The major contribution to MDSDV's routing load overhead is from *Full Dumps* and *Error Packets*. The movement speed of nodes has an impact on their status. As the movement speed increases, the probability of meeting a new neighbour (discovering a new

neighbour) or of losing contact with an old neighbour (discovering a broken link) increases. As a result, the node responds by unicasting a Full Dump or responds by broadcasting an Error Packet. The high MDSDV's routing load overhead is the price of its high delivery ratio.

In contrast, Figure 7.22 shows that MDSDV presents slightly lower routing load than DSDV in static networks. Compared to DSDV, MDSDV reduces the routing load by 22% in static networks.

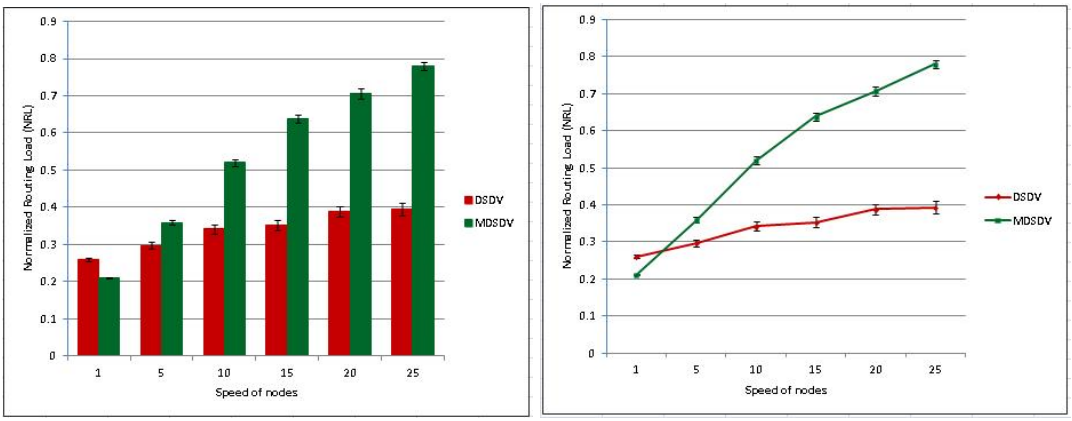

Figure 7.21: NRL vs Speed of Nodes (Dynamic network: Pause Time = 0 sec)

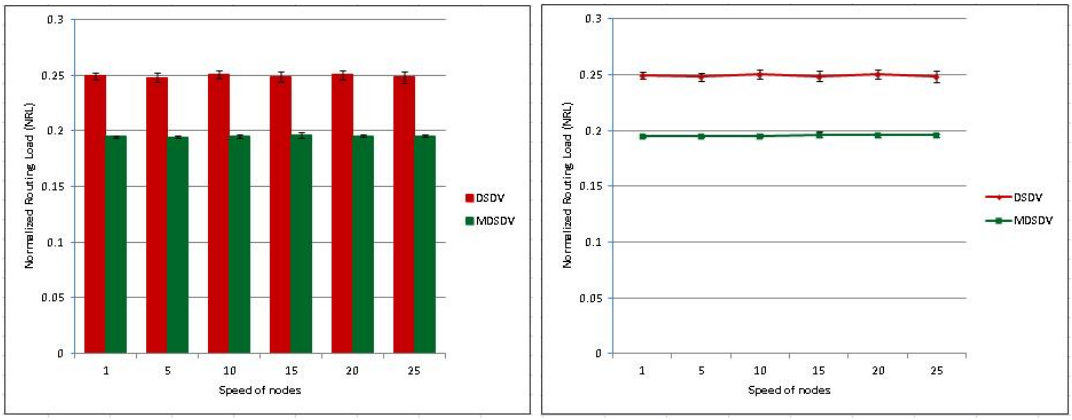

Figure 7.22: NRL vs Speed of Nodes (Static network: Pause Time = 200 sec)

- Data Packets Dropped

  Figures 7.23 and 7.24 present the *Data Packets Dropped* by DSDV and MDSDV in both dynamic and static networks. From the figures we can be confident to consider that MDSDV drops much fewer data packets than DSDV in all cases.

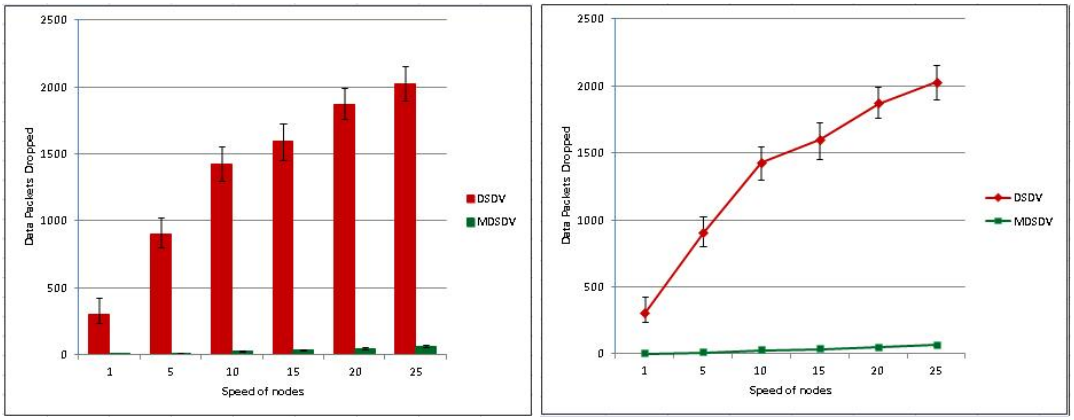

Figure 7.23: Data Dropped vs Speed of Nodes (Dynamic network: Pause Time 0 sec)

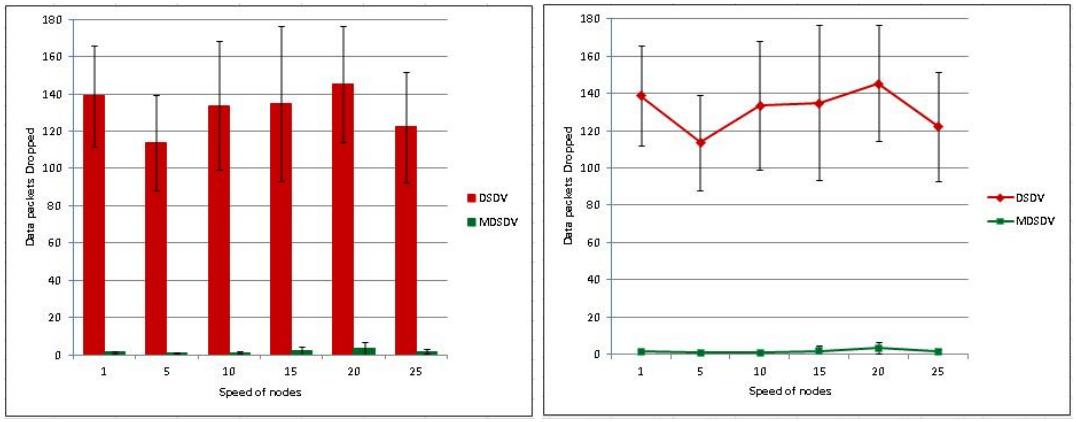

Figure 7.24: Data Dropped vs Speed of Nodes (Static network: Pause Time 200 sec)

Figure 7.23 shows that MDSDV drops much fewer data packets than DSDV in dynamic networks and the difference increases dramatically as the speed increases. An interesting observation is that MDSDV drops only between 0.7% and 3% of data packets dropped by DSDV in both environments (dynamic and static networks).

This is because DSDV has only a single path to each destination, and if a path to the desired destination does not exist or a broken one is found, the packets are dropped instead of queued [45]. In contrast, using MDSDV, an active node which is forwarding data packets commonly maintains several fresh alternative paths. If the used path fails, the data packet can be forwarded through another path instead of being dropped.

7.3 Summary

The objective of this chapter was to provide a quantitative comparison of the DSDV and MDSDV routing protocols. The performance comparison uses four metrics: *Packet Delivery Fraction*, *Average End-to-End Delay*, *Normalized Routing Load*, and *Data Packets Dropped* in three different scenarios: changing number of nodes, changing pause time, and changing speed of nodes. The results show that MDSDV routing protocol is effective especially in situations where nodes move frequently. The key observations are as follows.

1. *Packet Delivery Fraction*: Our results show that MDSDV is more robust and it improves the performance of DSDV in all of the simulated scenarios. The difference in performance increases as the mobility increases (Figures 7.9, 7.10, and 7.17). Also, it is observed that MDSDV is a stable protocol that delivers more than 99% of data packets in all cases (except for a 20 node network) (Figure 7.2). It is difficult for a small network (e.g., 20 nodes) to demonstrate availability of multiple paths, since there are few nodes that offer alternative routes. Figures 7.17 and 7.18 show that the performance has slightly improved by 2% in static networks, whereas it has considerably improved by between 5% and 32% in dynamic networks (depends on the speed). Moreover the speed of the nodes has a little impact on the performance of MDSDV. However, the performance of DSDV dramatically decreases in dynamic networks as the speed increases (Figure 7.17).

2. The results show that MDSDV and DSDV have a similar delay in almost all cases. The only exception is in 30 node networks where MDSDV produces more delay by %90 in dynamic environment (Figure 7.3), and produces less delay by %16 in static environment (Figure 7.4).

3. From our results, we found that in a high mobility environment (Figure 7.21), MDSDV demonstrates significantly higher routing load than DSDV. In contrast, it provides a lower routing load than DSDV in low mobility environment (Figure 7.13 and 7.22). The high routing load of MDSDV in a high mobility environment is due to the extra routing packets that are broadcast. Specifically, the number of *Full Dumps* and *Error Packets* increases as the mobility increases, as a result of discovering new neighbours and discovering link failures. The high MDSDVs routing load overhead is the price of its high delivery ratio

4. MDSDV drops significantly less data packets than DSDV in all cases. From the figures 7.15, 7.16, 7.23, and 7.24 we observed that the mobility and speed have a little impact on the performance of MDSDV in terms of *Data Packets Dropped*. In contrast, the mobility and speed have a significant impact on the performance of DSDV. The number of data packets dropped by DSDV increases as the mobility and/or speed increase (Figures 7.15, 7.16, and 7.23).
   MDSDV dropped less data packets because an alternative route can always be used by the source and the intermediate nodes in response to link failures. However, no such alternative path is available for DSDV and thus packets are dropped until a new route can be found.

Our results indicate that the performance of MDSDV protocol is certainly superior to standard DSDV. MDSDV has several novel aspects in that increase the Packet Delivery Fraction, reduce the number of Data Packets Dropped, reduce the control overhead in a low mobility environment, and achieve multiple node-disjoint routing paths. It is evident from our simulation results that MDSDV outperforms DSDV

because multiple node-disjoint routing paths provide robustness to mobility. The simulation results show that MDSDV increases the Packet Delivery Fraction and reduces the number of Data Packets Dropped with little increased overhead in MANETs with high mobility.

There are key aspects of the MDSDV design that contribute to its good performance compared to DSDV. Firstly, MDSDV uses multiple node-disjoint paths instead of a single path. Secondly, the Update Packet is time-triggered only. Thirdly, the Full Dumps are unicast only when discovering new neighbours. Fourthly, the Error Packets are not rebroadcast. Finally, the routes are always updated by Update Packets and Full Dumps which make the routes fresh.

Chapter 8

Performance Comparison with AODV and DSR

In this chapter, our overall goal is to compare the performance of MDSDV with two well known reactive routing protocols: AODV and DSR. We have chosen these two reactive protocols as AODV is one of the most popular and widely researched on-demand ad hoc protocols [55]. and DSR is a protocol in which a node may learn and cache multiple routes to any destination [47], and hence an alternative path can be used in case of a link failure. We use similar traffic and mobility models to [49]. The rest of this chapter is organized as follows: Section 8.1 presents the performance evaluation methodology. Section 8.2 presents the comparison results of our simulations. Finally, the simulation results are summarized in section 8.3.

8.1 Methodology

In order to make fair comparisons between the protocols, it is necessary to challenge the protocols with identical loads and environmental conditions. Four experiments are reported in this chapter. For each experiment, we pre-generated a number of

scenario files that describe the exact motion of each node and the exact sequence of packets originated by each node. In the first experiment (Section 8.2.1), we varied the number of source-destination pairs to change the offered load in the network. The second experiment (Section 8.2.2) shows the impact of using different network sizes on the behaviour of the protocols by varying the number of nodes. Varying the pause time in the third experiment (Section 8.2.3) shows some of the impact of mobility on the behaviour of the four protocols. Finally, the fourth experiment (Section 8.2.4) manifests the impact of the speed on the performance of the protocols

We simulated a number of mobile nodes forming an ad hoc network, moving about over a square (670m x 670m) flat space for 100 seconds of simulated time. A square space is chosen to allow nodes to move more freely with equal node density [116]. Nodes in the simulation move according to the random waypoint model [11][15][63], where each node starts its journey from a random location to a random destination at a speed distributed uniformly between 0 and some maximum speed. Upon reaching the destination, the node pauses for pause time seconds, selects another destination, and proceeds there as previously described, repeating this behaviour for the duration of the simulation (100 seconds).
The communication model used in our simulations is constant bit rate (CBR) traffic. The size of the packet is 512 bytes. All connections were started at certain times and continued until the end of the simulation time of 100 seconds.

Each data point represents an average of 30 runs with identical traffic models, but different randomly generated mobility scenarios. We include error bars which indicate 95% confidence that the actual mean is within the range of said interval. In certain cases, the confidence intervals are small enough that they are obscured by the symbol itself. The four experiments use the simulation parameters given in Table 6.1 with some differences. The differences are listed in tables 8.1, 8.2, 8.3, and 8.4.

8.1.1 Performance Evaluation Metrics

In this chapter, we present a performance comparison of MDSDV with the AODV and DSR routing protocols. The comparison is based on the same four metrics that are considered in Chapter 7: *Packet Delivery Fraction (PDF)*, *Average End-to-End delay*, *Normalized Routing Load (NRL)*, and *Data Packets Dropped*.

8.2 Simulation Results

This chapter reports on four experiments manifesting the impact of *Offered Load* (Section 8.2.1), *Network Size* (Section 8.2.2), *Pause Time* (Section 8.2.3), and *Speed of Nodes* (Section 8.2.4). The simulation results bring out several important characteristic differences in the three protocols. We categorize and discuss them in the following subsections.

8.2.1 Offered Load (Varying Number of Sources)

This experiment considers a MANET with 50 mobile nodes spread randomly over an area of 670m x 670m. Nodes move with a maximum speed of 20 meters/sec with two pause times: 0 and 100 seconds. In this experiment, we present an evaluation of the impact of changing the number of sources on the behaviour of the protocols. We vary the source-destination pairs (10, 20, 30, 40, and 50 traffic sources), while keeping the packet's size and the sending rate constant at 512 bytes and 4 packets per second respectively. The source-destination pairs are spread randomly over the network, and the number of sources is varied to change the offered load in the network. Simulations are run for 100 simulated seconds. Each data point represents an average of thirty runs with different randomly generated mobility scenarios. Table 8.1 shows the simulation parameters that differ from the baseline parameters given in Table 6.1.

Parameter	Value
Simulation time	100 seconds
Number of nodes	50 nodes
Pause time	0, 100 seconds
Max. speed of nodes	20 m/s
Number of sources	10, 20, 30, 40, and 50 sources

Table 8.1: Parameters used in the first experiment to compare MDSDV with AODV and DSR

- Packet Delivery Fraction (PDF)

  Figures 8.1 and 8.2 show the *Packet Delivery Fraction* of MDSDV, AODV and DSR for both dynamic and static networks respectively, as a function of network load (number of sources).

  The simulation analysis for the figures show that the PDF of the three protocols are similar for 10 and 20 sources in both dynamic and static networks. For networks with 30, 40, and 50 sources, figure 8.1 shows that MDSDV has slightly better PDF than AODV and DSR in dynamic networks, whereas figure 8.2) shows that MDSDV outperforms AODV and DSR in static networks and the difference in performance increases as the number of sources increases. MDSDV outperforms AODV by up to 12% and outperforms DSR by up to 9%. This is because both of the reactive protocols require more paths to send data when the number of sources increases. As a result, extra routing packets (RREQ and RREP) are broadcast which may create packet collisions. Whereas, using MDSDV, a node expects to have at least one path ready for each destination in its routing table.

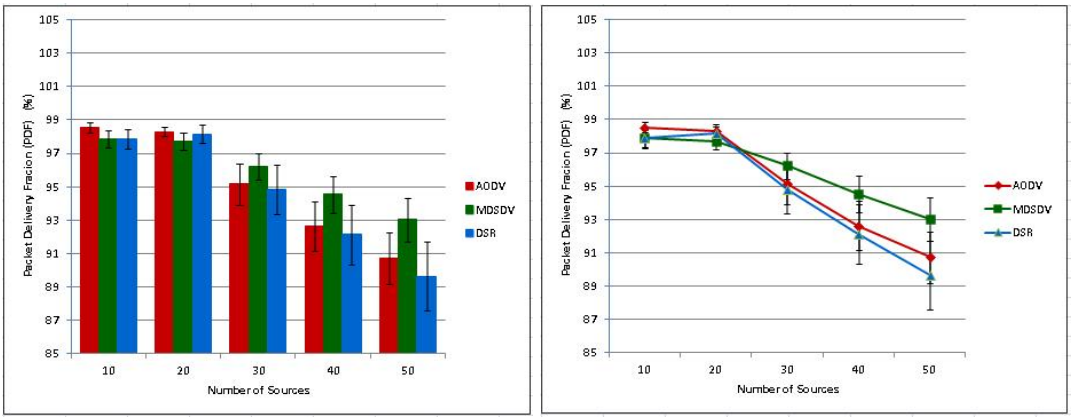

Figure 8.1: PDF vs Number of Sources (Dynamic network)

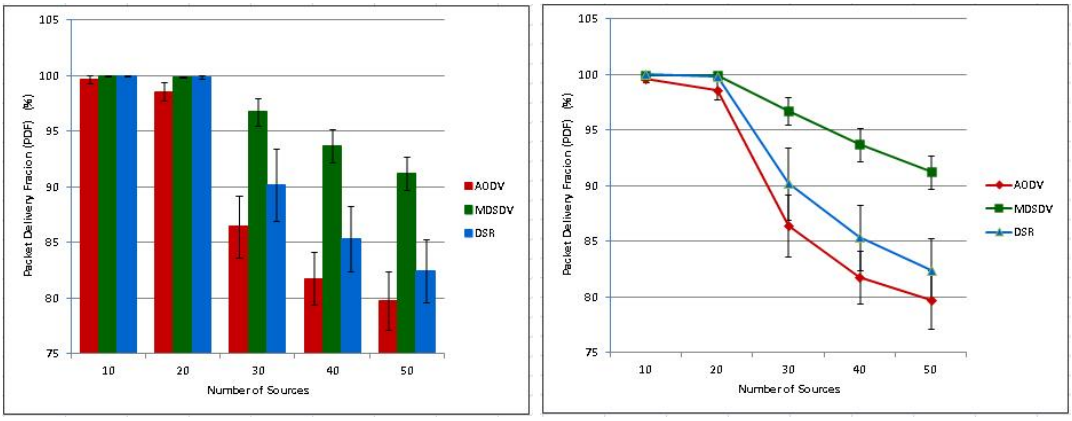

Figure 8.2: PDF vs Number of Sources (Static network)

- Average End-to-End Delay

  The *Average End-to-End Delays* of MDSDV, AODV, and DSR are shown in figures 8.3 and 8.4 for dynamic and static networks respectively. The main observation is that DSR exhibits the highest delay with 30, 40, and 50 sources in both cases (dynamic and static networks). The figures show that MDSDV, AODV, and DSR have similar delays for 10 and 20 sources in dynamic networks (Figure 8.3).

  The delay of DSR is comparable to MDSDV at small traffic loads (10 and 20 sources), but with the increase in network load (30, 40, and 50 sources), delay in DSR is much higher than MDSDV. Compared to DSR, MDSDV reduces the delay by between 13% and 67% in static networks and between 17% and 58%

in dynamic networks. From the figures we can see that MDSDV and AODV have very similar delay in dynamic networks. Moreover, we can see that both protocols have similar delay in static networks with 30, 40, and 50 sources, whereas MDSDV offers a significant reduction in delay by up to 66% in static networks with 20 and 30 sources.

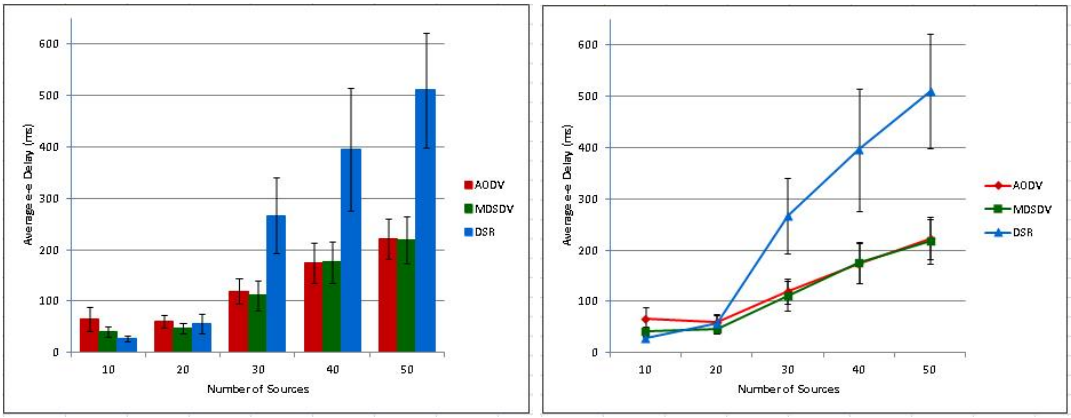

Figure 8.3: Average End to End Delay vs Number of Sources (Dynamic network)

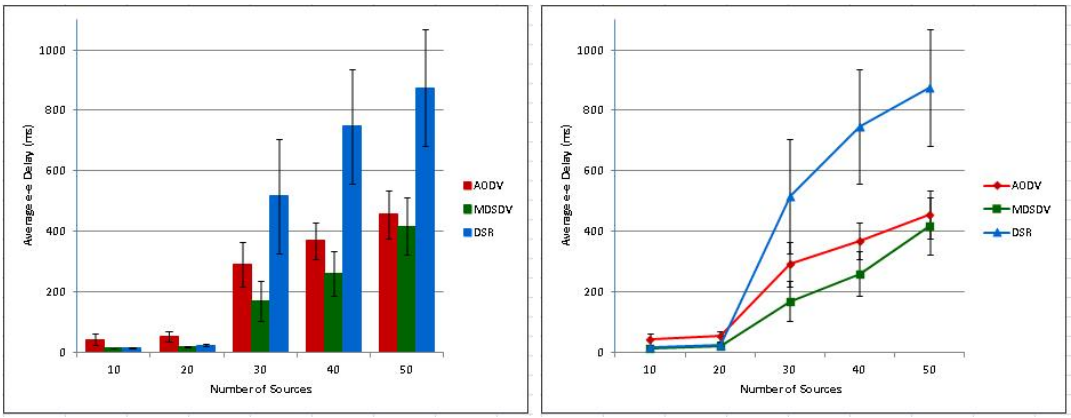

Figure 8.4: Average End to End Delay vs Number of Sources (Static network)

- Normalized Routing Load (NRL)

  Figures 8.5 and 8.6 show the *Normalized Routing Load* for both dynamic and static networks as a function of the number of sources. The results show that AODV demonstrates significantly the highest routing load which increases as

the number of sources increases. We expected this, since AODV is an on-demand routing protocol and as the number of sources increases, more routing packets have to be transmitted in order for routes to more destinations to be maintained [16]. Compared to AODV, MDSDV decreases the routing load by between 18% and 72% in dynamic networks (Figure 8.5) and between 6% and 93% in static networks (Figure 8.6).

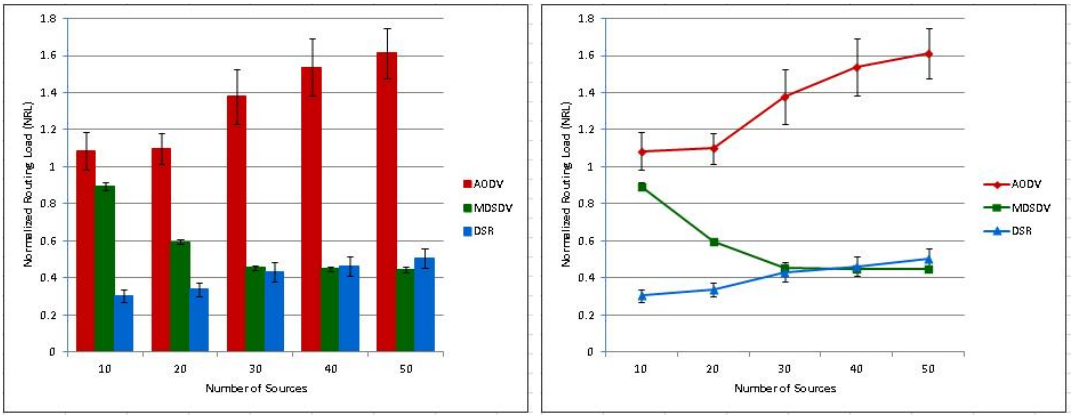

Figure 8.5: NRL vs Number of Sources (Dynamic network)

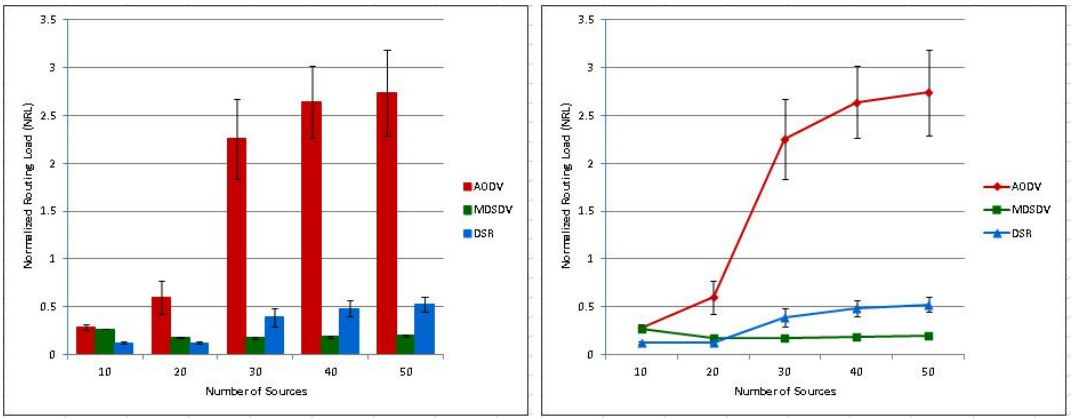

Figure 8.6: NRL vs Number of Sources (Static network)

On the other hand, MDSDV and DSR seem to be competitive with each other. In dynamic networks, Figure 8.5 shows that MDSDV produces more overhead (up to 196%) for small traffic loads (10 and 20 sources), and produces similar overhead for large traffic loads (30, 40, and 50 sources). However, in static

networks (Figure 8.6), MDSDV produces more overhead by between 43% and 119% for small traffic loads, and produces less overhead by between 53% and 61% for large traffic loads.

The other observation is that the MDSDV does not produce more overhead as number of sources increases. This is expected as MDSDV is a proactive routing protocol and the routes to all destinations in the network are created even if they are not needed.

- Data Packets dropped

  Figures 8.7 and 8.8 show the number of data packets dropped by the three protocols as a function of the number of sources.

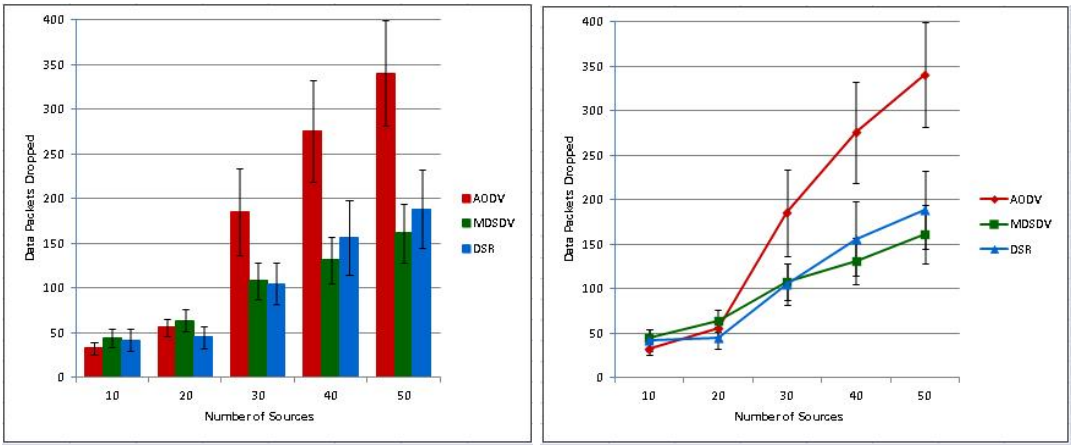

Figure 8.7: Data Dropped vs Number of Sources (Dynamic network)

For small traffic loads (10, 20 sources), the three protocols drop similar number of data packets in both networks (dynamic and static networks). In contrast, for large traffic loads (30, 40, and 50 sources), the figures show that AODV drops significantly more data packets. For large traffic loads, MDSDV dropped between 47% and 58% in dynamic networks (Figures 8.7), and drops between 12% and 22% in static networks (Figures 8.8) of data packets dropped by AODV.

MDSDV and DSR dropped similar data packets in dynamic networks, whereas MDSDV dropped between 38% and 50% of data packets dropped by DSR in static networks.

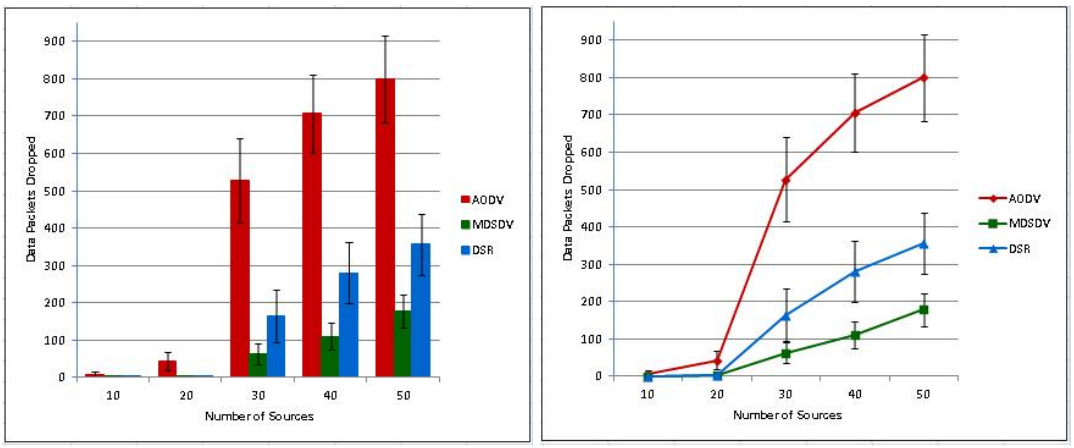

Figure 8.8: Data Dropped vs Number of Sources (Static network)

The number of sources has relatively less impact on the performance of MDSDV. This is because MDSDV is a multipath proactive routing protocol and hence an alternative path is mostly available that can be used immediately to forward a data packet instead of dropping it. In contrast, AODV is a single path routing protocol and has to flood a route request to find a new route to forward a data packet. This may cause the dropping of some data packets when waiting a long time before finding a route to the destination. DSR may have several routes for a certain destination in its cache. Using one of the stale routes by DSR may lead to the dropping of some data packets.

8.2.2 Network Size (Varying Number of Nodes)

This set of experiments varies the number of nodes to show the impact of network size on the performance of MDSDV, AODV, and DSR. The simulations are performed for 30, 40, 50, 60, 70, 80, 90, and 100 nodes. Two pause times are used: 0 seconds correspond to a dynamic network and 100 seconds corresponds to a static network.

The results are collected at maximum speed of 20 m/s. We use 30 CBR traffic sources. Table 8.2 shows the simulation parameters that differ from the baseline parameters given in Table 6.1.

Parameter	Value
Simulation time	100 seconds
Number of nodes	30, 40, 50, 60, 70, 80, 90, and 100 nodes
Pause time	0, 100 seconds
Max. speed of nodes	20 m/s
Number of sources	30 sources

Table 8.2: Parameters used in the second experiment to compare MDSDV with AODV and DSR

- Packet Delivery Fraction (PDF)

  The first results show the *Packet Delivery Fraction* for the three protocols for dynamic (Figure 8.9) and static (Figure 8.10) networks.

  Figure 8.9 shows that MDSDV and AODV deliver similar data packets in dynamic networks with less than 80 nodes. Meanwhile, MDSDV and DSR deliver similar data packets in networks with less than 70 node. The same figure shows that MDSDV outperforms AODV and DSR in the other networks by up to 5% and 12% respectively. Figure 8.10 shows a trend for MDSDV to deliver a higher fraction of packets in static networks. And we can be confident that the PDF is higher in networks with more than 30 nodes. Specifically, MDSDV delivers more data packets than AODV by between 6% and 23.5%, and delivers more data packets than DSR by between 6% and 16%.

  Interestingly, in static networks (Figure 8.10), the PDF of AODV and DSR drops by 21% and 14% respectively, whereas the PDF of MDSDV only drops by 3.5%. This is because in MDSDV, the control packets are not transmitted to

the entire network. Thus, the increase in the number of nodes does not increase the number of control packets. In contrast, in AODV and DSR, the number of routing packets is directly proportional to the number of nodes because the control packets are transmitted to the entire network.

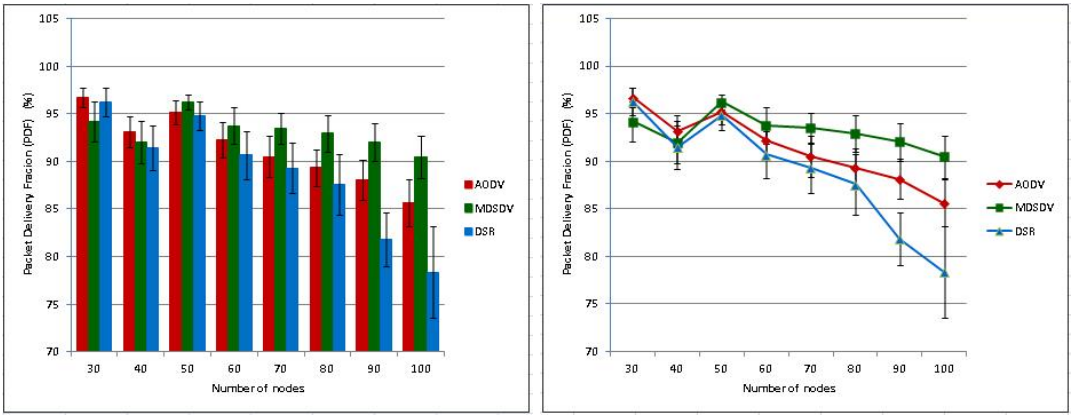

Figure 8.9: PDF vs Number of Nodes (Dynamic network)

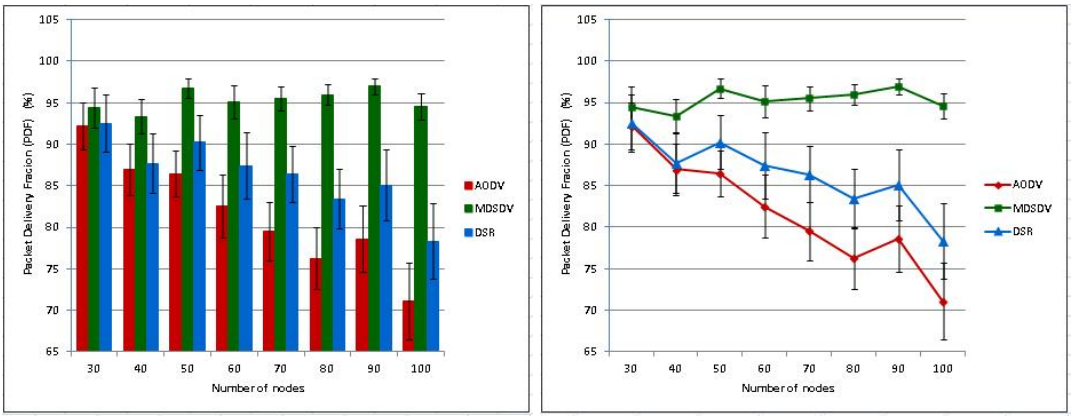

Figure 8.10: PDF vs Number of Nodes (Static network)

- Average End-to-End Delay

  Figures 8.11 and 8.12 plot the *Average End-to-End Delay* when changing the node density. The figures show that DSR has the highest delay in both dynamic and static networks.

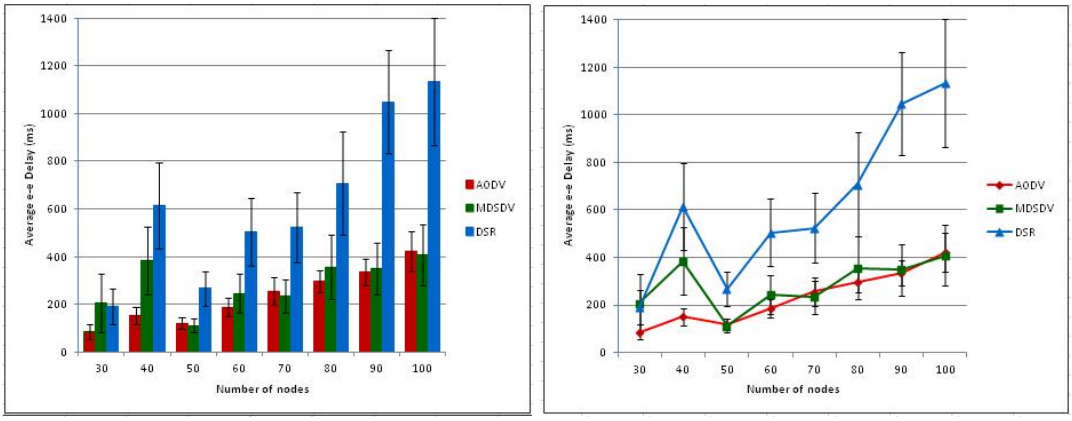

Figure 8.11: Average End-to-End Delay vs Number of Nodes (Dynamic network)

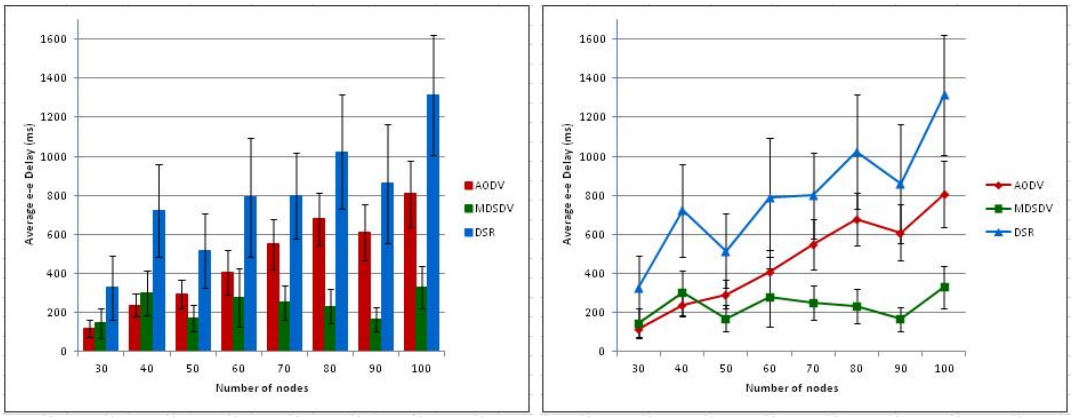

Figure 8.12: Average End-to-End Delay vs Number of Nodes (Static network)

In dynamic networks (Figure 8.11), although MDSDV and DSR provide similar delay in a 30 node network, MDSDV decreases the delay of DSR by between 37% and 67% in the other networks. On the other hand, Figure 8.11 shows that MDSDV incurs larger delays than AODV in networks with less than 50 nodes, whereas both protocols exhibit similar delay in networks with more than 40 nodes.

In static networks (Figure 8.12), MDSDV decreases the delay of DSR by 55% to 81%. Compared to AODV, Figure 8.12 shows that MDSDV creates a higher delay in 30 and 40 node networks by around 25%. Whereas, MDSDV decreases the delay of AODV by 31% to 73% in the other networks.

- Normalized Routing Load (NRL)

  Figures 8.13 and 8.14 demonstrate the *Normalized Routing Load* for dynamic and static networks respectively.

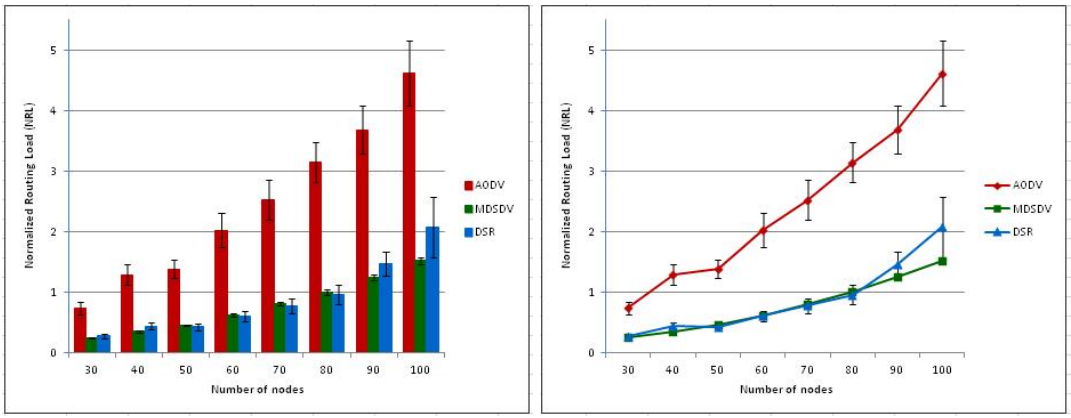

Figure 8.13: NRL vs Number of Nodes (Dynamic network

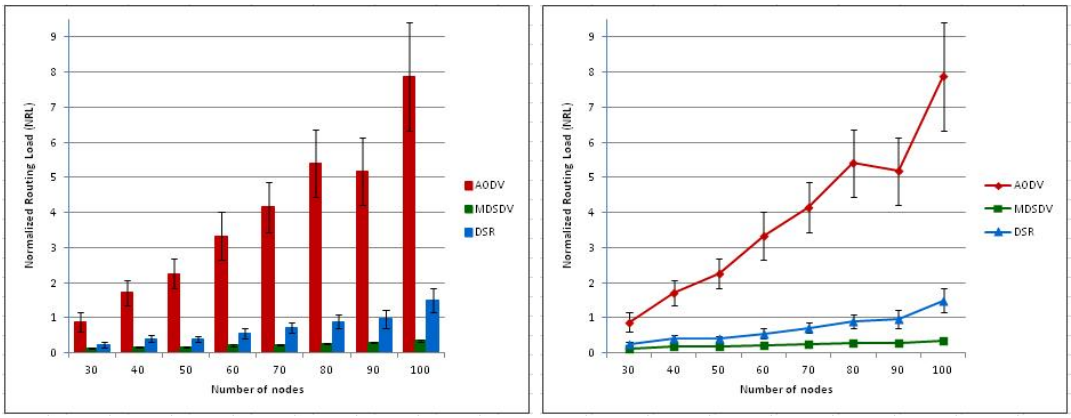

Figure 8.14: NRL vs Number of Nodes (Static network

In all cases, AODV produces significantly the highest routing load. Compared to AODV, MDSDV provides between 66% and 72% reduction in routing overhead in dynamic networks (Figure 8.13), whereas it provides between 85% and 95% reduction in static networks (Figure 8.14). Although MDSDV and DSR provide a similar routing load in dynamic networks, MDSDV provides between 42% and 75% reduction in routing overhead in static networks.

In static networks, as the node density is increased, MDSDV maintains the flattest curve when compared to the other two protocols. This shows that the number of retransmitting nodes do not significantly increase in MDSDV. We can be confident to conclude that MDSDV produces the lowest routing load in static networks.

- Data Packets Dropped

  Figure 8.15 and 8.16 show the number of data packets that are dropped by the three protocols, and show that AODV drops more data than the other two protocols.

  In dynamic networks (Figure 8.15), AODV and MDSDV drop similar data packets in 30 and 40 nodes networks. Whereas MDSDV dropped between 42% and 54% of data packets that are dropped by AODV in the other networks. The same figure shows that MDSDV and DSR drop similar data packets in networks with 40, 50, 60, 70 and 80 nodes networks. However MDSDV drops more data than DSR in 30 nodes networks by 150%, it delivers less data packets in 90 and 100 nodes networks by 43% in both cases.

  On the other hand, in static networks (Figure 8.16) compared to AODV, MDSDV decreases the data packet drops by between 66% and 94% in networks with more than 30 nodes. Whereas both protocols drop similar data packet in 30 node network. Meanwhile, the figure show that both of MDSDV and DSR drop similar data packets in 30 and 40 node networks, whereas MDSDV dropd between 62% and 84% of data packets that are dropped by DSR in the other networks.

  Packet drops are less frequent with MDSDV as alternate routing table entries can often be assigned in response to link failures.

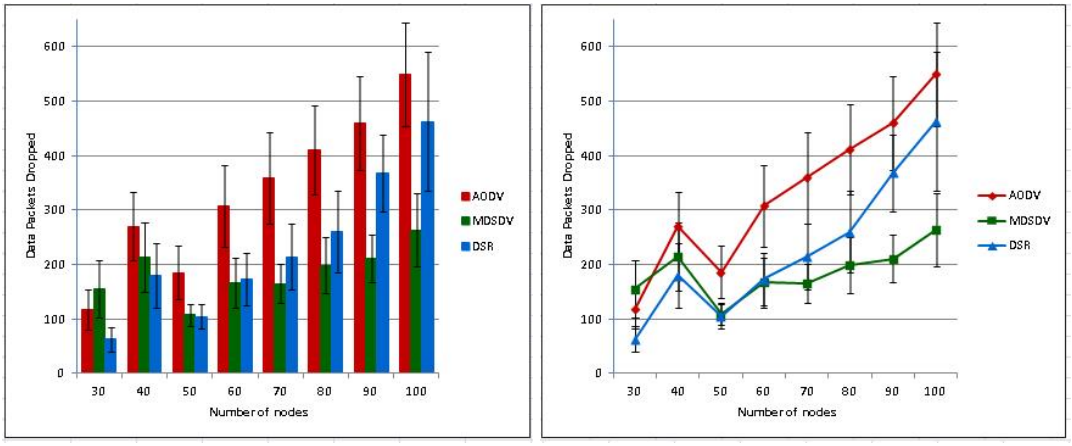

Figure 8.15: Data Dropped vs Number of Nodes (Dynamic network

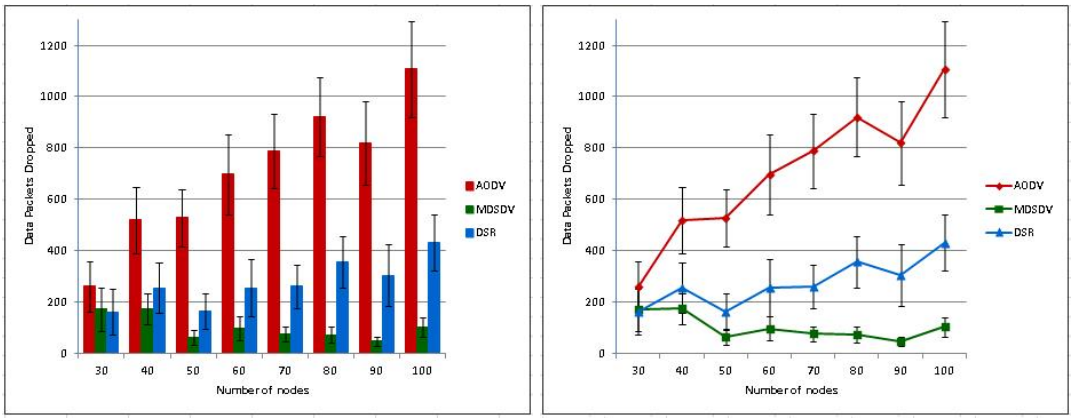

Figure 8.16: Data Dropped vs Number of Nodes (Static network

8.2.3 Mobility (Varying Pause Time)

This experiment simulates a 50 node network. The pause time is varied between 0 sec (dynamic network) and 100 sec (static network). Specifically, the following five pause time values are used in our simulations: 0, 25, 50, 75, and 100 seconds. Varying the pause time changes the frequency of node movement. Two node speeds are chosen: 1 m/sec as low speed and 20 m/s as high speed networks. The network consists of 30 CBR/UDP traffic sources sending 512 byte packets to chosen destinations at the rate of 4 packets/sec. The total simulation time is 100 seconds, and each data point in the

following figures is the average of 30 simulation runs. We include error bars which indicate 95% confidence that the actual mean is within the range of said interval. Table 8.3 shows the simulation parameters that differ from the baseline parameters given in Table 6.1.

Parameter	Value
Simulation time	100 seconds
Number of nodes	50 nodes
Pause time	0, 25, 50, 75, 100 seconds
Max. speed of nodes	1, 20 m/s
Number of sources	30 sources

Table 8.3: Parameters used in the third experiment to compare MDSDV with AODV and DSR

- Packet Delivery Fraction (PDF)

  Figures 8.17 and 8.18 show the *Packet Delivery Fractions (PDF)* for variations of the pause time for MDSDV, AODV, and DSR in both low speed and high speed networks.

  In low speed networks, figures 8.17 shows a trend for MDSDV to deliver a higher fraction of packets. And we can be confident that the PDF is higher for all pause time values. MDSDV achieves up to 10.5% higher PDF than AODV and up to 8.5% higher PDF than DSR. In high speed networks, figure 8.18 shows that MDSDV and AODV deliver similar data packets at pause times 0 sec and 25 sec (high mobility networks) environment, however MDSDV achieves up to 10% higher PDF in medium and low mobility (i.e., pause time is greater than 25 sec) environment. On the other hand, MDSDV outperforms DSR at pause time 100 sec (low mobility) environment (up to 7% higher), and both protocols deliver similar data packets at the other pause times.

The better performance of MDSDV is due to the prospect of being able to exploit fresh routes for many destinations

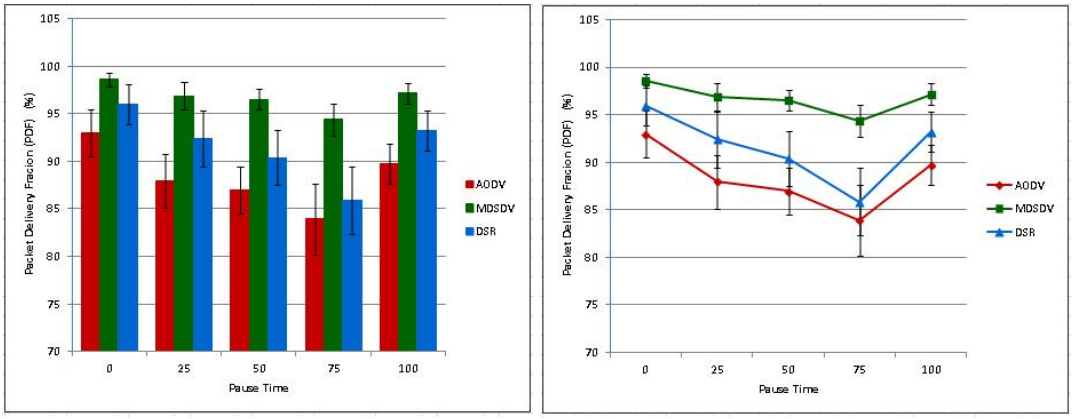

Figure 8.17: PDF vs Pause Time (Low Speed)

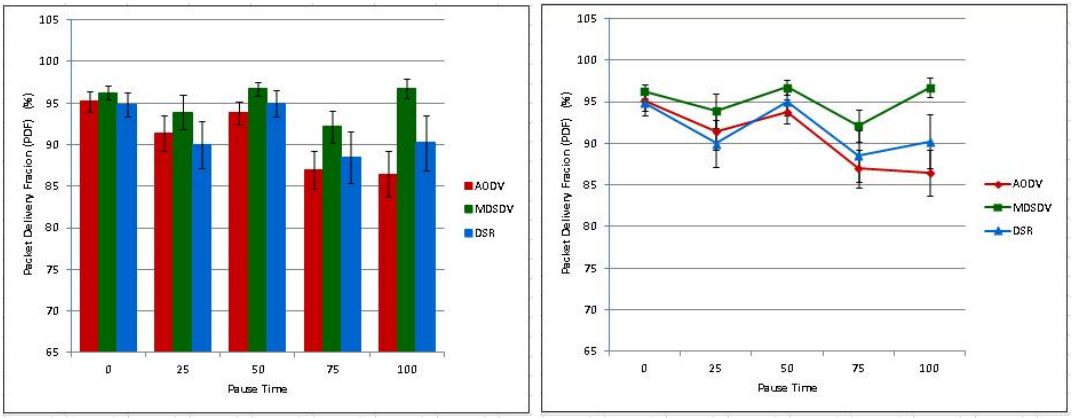

Figure 8.18: PDF vs Pause Time (High Speed)

- Average End-to-End Delay

  Figures 8.19 and 8.20 show the *Average End-to-End Delay* for MDSDV, AODV, and DSR in both low speed and high speed networks. The figures show that DSR demonstrates significantly the highest delay in both cases, follows by AODV and MDSDV. Compared to DSR, MDSDV decreases the delay by between 54% and 71% in low speed networks (Figure 8.19), and between 58%

and 67% in high speed networks (Figure 8.20).On the other hand, MDSDV and AODV produce similar delay in all cases (except for low speed networks at 0 sec pause time where MDSDV reduces the delay of AODV by 60%).

The main reasons for the large data packet delays in DSR are the lack of a mechanism to remove expired unused routes from its caches, together with the aggressive use of caching [16]. The route discovery process of AODV may also cause long delays, due to the number of control packets being transmitted. These delays result in packets waiting in the queues being dropped.

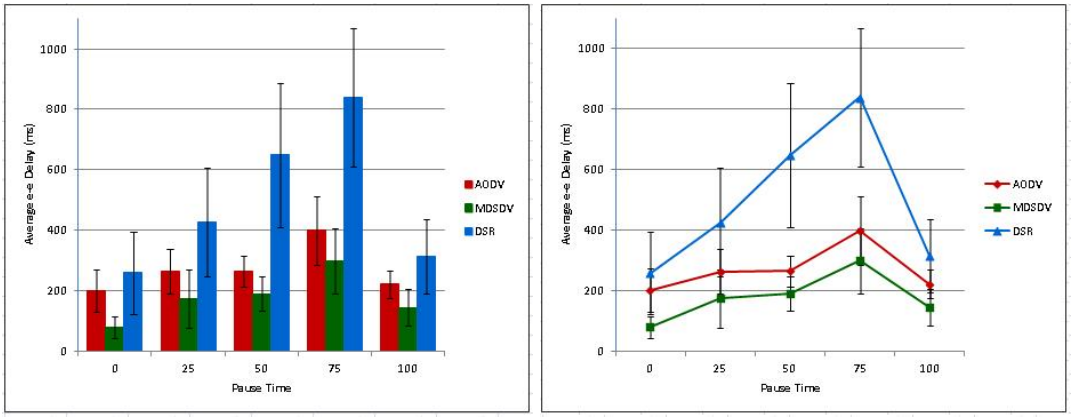

Figure 8.19: Average End to End Delay vs Pause Time (Low Speed)

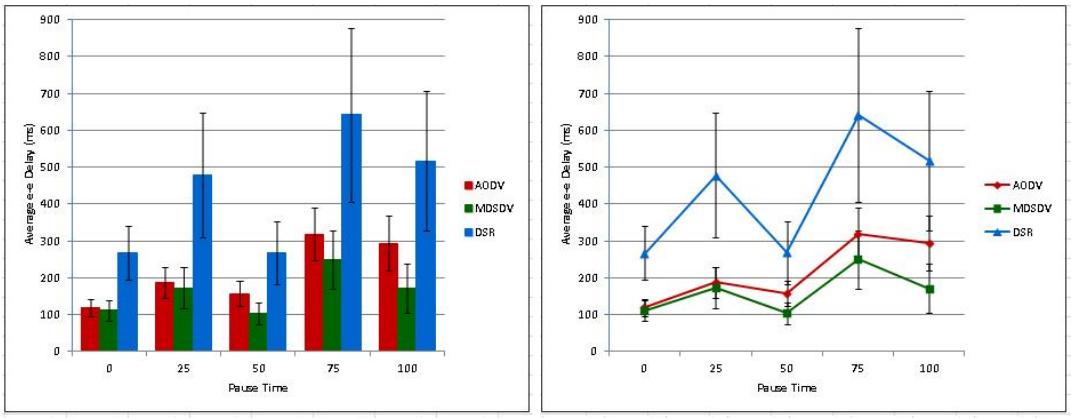

Figure 8.20: Average End to End Delay vs Pause Time (High Speed)

- Normalized Routing Load (NRL)

  The *Normalized Routing Load (NRL)* of MDSDV, AODV, and DSR is exhibited in figure 8.21 and figure 8.22. The figures show that AODV always demonstrates the highest routing load in both low and high speed networks.

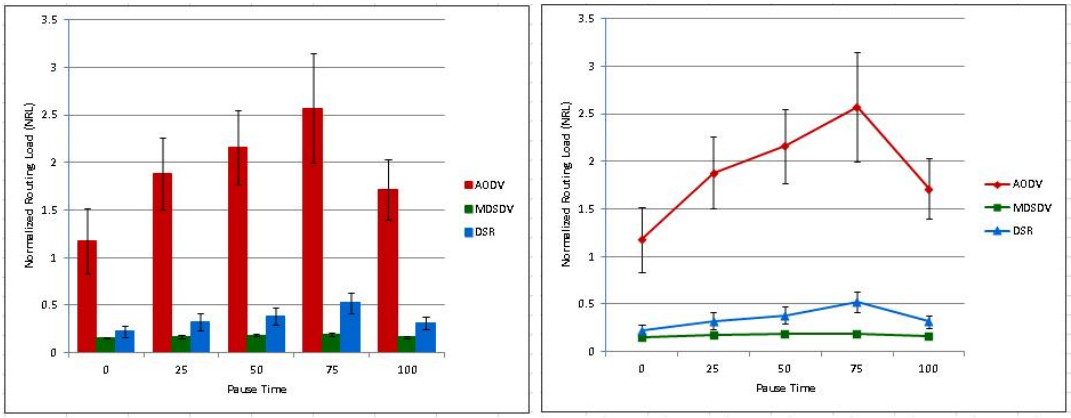

Figure 8.21: NRL vs Pause Time (Low Speed)

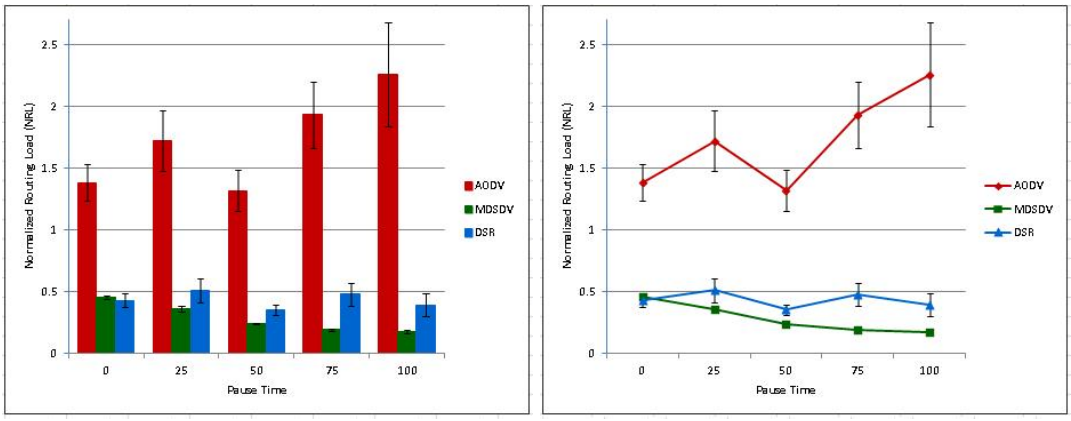

Figure 8.22: NRL vs Pause Time (High Speed)

Compared to AODV, MDSDV reduces the overhead by between 86% and 92% in low speed networks (Figure 8.21), and between 67% and 92% in high speed networks (Figure 8.22). This is because AODV is a single path protocol, and this high increase occurs because each of the route discoveries in AODV is typically propagated to every node in the network [56].

In comparison with DSR, MDSDV reduces the routing load in low speed networks by between 27% and 62% (Figure 8.21). On the other hand, in high speed networks, figure 8.22 shows that MDSDV and DSR demonstrate similar routing load at 0 pause time, whereas MDSDV reduces the routing load of DSR by between 27% and 58% at the other pause times.

- Data Packets Dropped

  The last metric investigated here is the *Data Packets Dropped* by MDSDV, AODV, and DSR (Figures 8.23 and 8.24). The figures show that AODV dropped more data packets than the other two protocols.

  MDSDV dropped between 10% and 20% of the data dropped by AODV in low speed networks (Figure 8.23) and dropped between 42% and 88% in high speed networks (Figure 8.24). On the other hand, although MDSDV and DSR drop similar number of data packets in many cases, MDSDV drops fewer data packets than DSR in some cases. In low speed networks (1 m/sec), figure 8.23 shows that MDSDV drops 62% and 55% of data packets dropped by DSR at 50 sec and 75 sec pause times. In high speed networks (20 m/sec), figure 8.24 shows that MDSDV drops 62% of data packets dropped by DSR at 100 sec pause time.

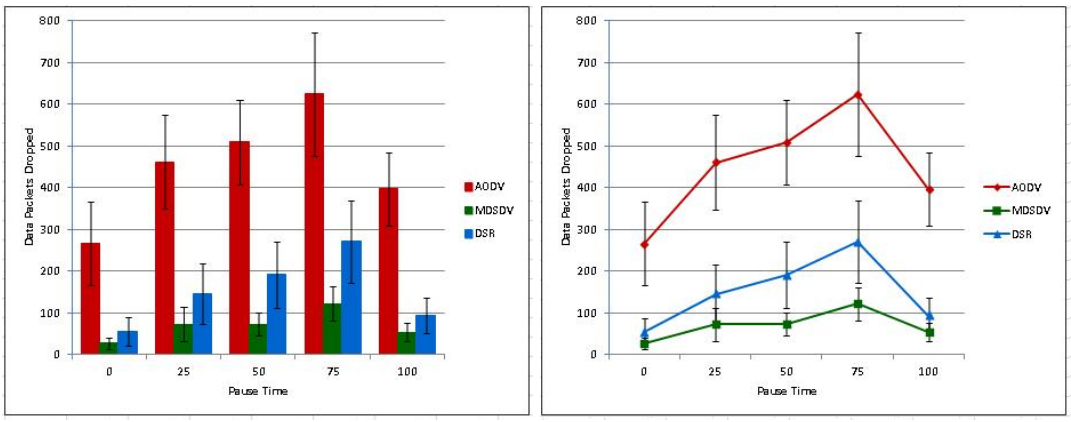

Figure 8.23: Data Dropped vs Pause Time (Low Speed)

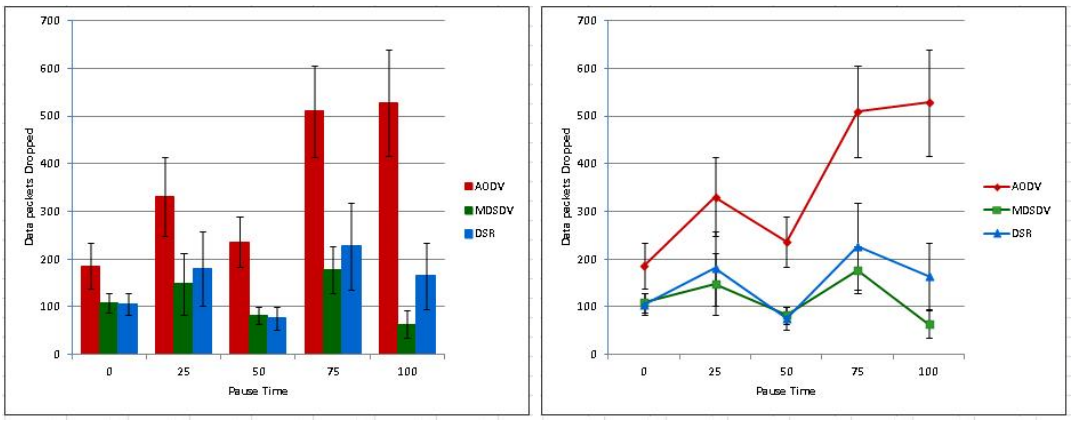

Figure 8.24: Data Dropped vs Pause Time (High Speed)

8.2.4 Mobility (Varying Speed of Nodes)

To explore how the protocols behave as the rate of topology change varies, we varied the mobility by varying the maximum speed of nodes, and evaluated all three protocols over scenario files using these movement speeds. In this section, we report on a 50 node network simulation. two pause times are used: 0 sec (dynamic network) and 100 sec (static network). Five maximum speeds are used in our simulations. Specifically, 1 m/s (low speed), 5, 10, 15, and 20 m/s (high speed) are used. Varying the speed of nodes changes the frequency of node movement. The network consists of 30 CBR/UDP traffic sources sending 512 byte packets to chosen destinations at the rate of 4 packets/sec. The total simulation time is 100 seconds, and each data point in the following figures is the average of 30 runs. Table 8.4 shows the simulation parameters that differ from the baseline parameters given in Table 6.1.

Parameter	Value
Simulation time	100 seconds
Number of nodes	50 nodes
Pause time	0, 100 seconds
Max. speed of nodes	1, 5, 10, 15 and 20 m/s
Number of sources	30 sources

Table 8.4: Parameters used in the fourth experiment to compare MDSDV with AODV and DSR

- Packet Delivery Fraction (PDF)

  Figures 8.25 and 8.26 show the *Packet Delivery Fraction (PDF)* when we vary the speed of nodes for the MDSDV, AODV, and DSR routing protocols in dynamic and static networks. In dynamic networks, figure 8.25 shows that MDSDV and DSR deliver similar data packets. However, MDSDV performs better than AODV by up to %6. The difference in performance increases as the speed of nodes decreases. In static networks, figure 8.26 shows that MDSDV outperforms AODV and DSR. Specifically, MDSDV delivers more data packets than AODV by between %7.5 and %10 and delivers more data packets than DSR by between %4 and %10.

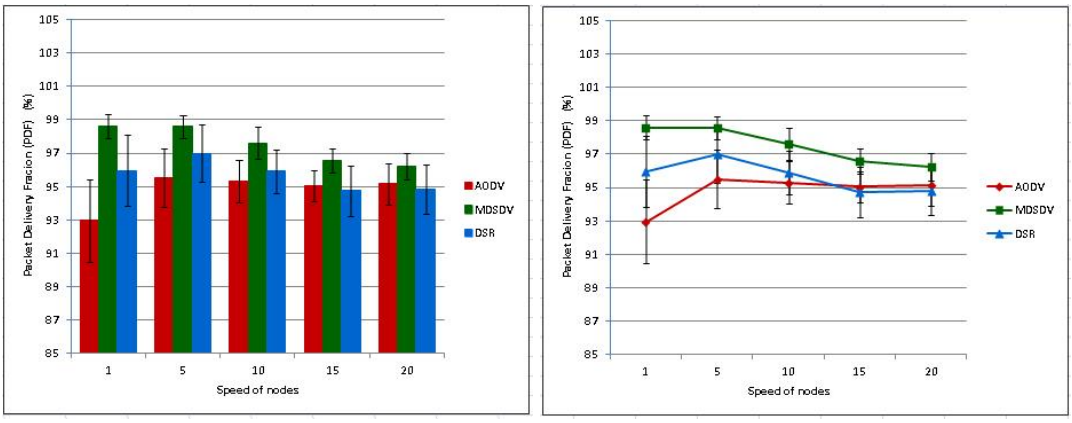

Figure 8.25: PDF vs Speed of Nodes (Dynamic network)

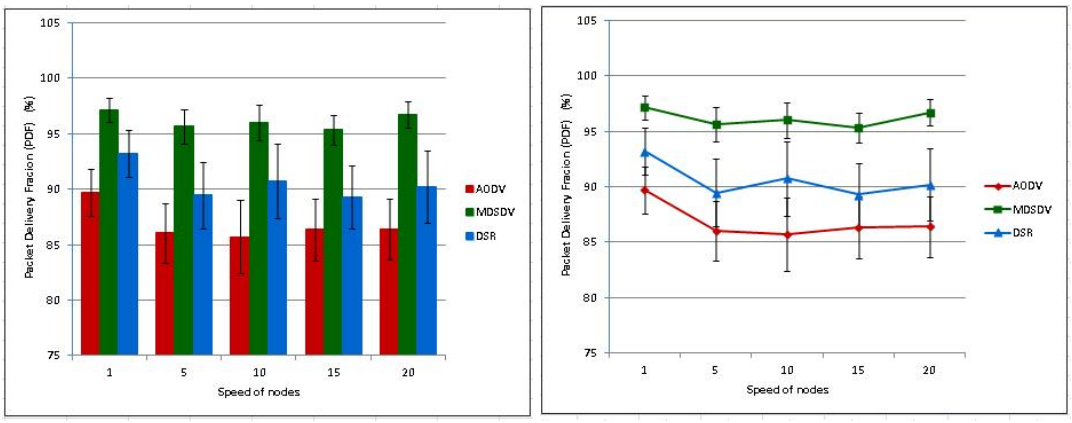

Figure 8.26: PDF vs Speed of Nodes (Static network)

- Average End-to-End Delay

  Figures 8.27 and 8.28 show the *Average End-to-End Delay* produced by the three protocols when varying the speed of nodes in both dynamic and static networks respectively. The figures illustrate that DSR produces the highest delay in all cases. MDSDV reduces the delay of DSR by between 58% and 69% in dynamic networks, and between 54% and 67% in static networks. However MDSDV and AODV produce similar delay at all speeds in static networks (Figure 8.28).

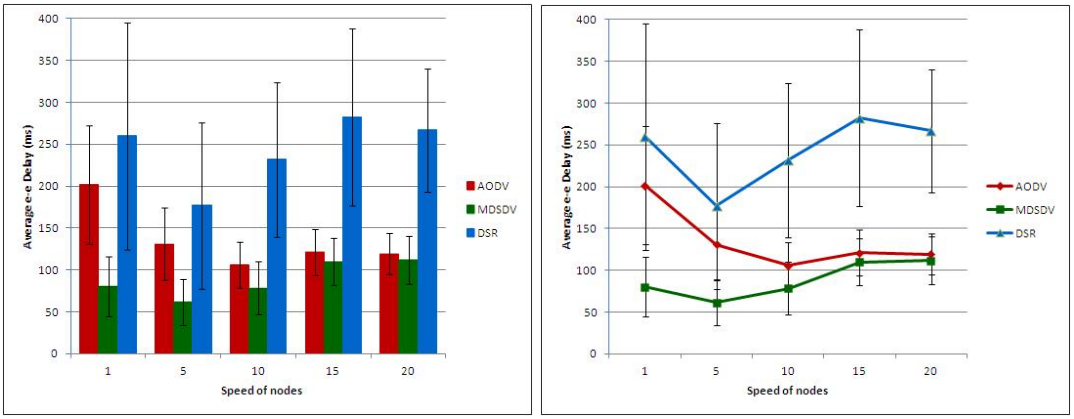

Figure 8.27: Average End to End Delay vs Speed of Nodes (Dynamic network)

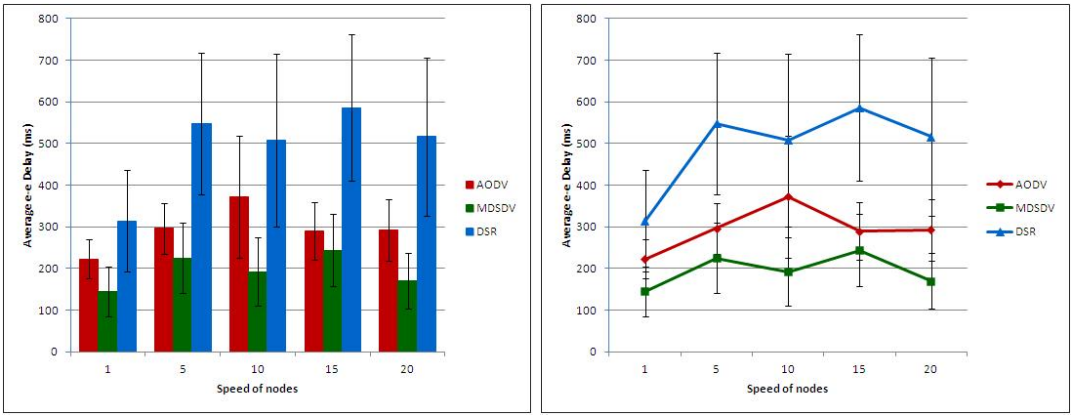

Figure 8.28: Average End to End Delay vs Speed of Nodes (Static network)

Figure 8.27 shows that MDSDV exhibits less delay than AODV at low speeds (1 m/sec and 5 m/sec) by 60% and 53% respectively, whereas it exhibits similar delay at medium and high speeds (10, 15, and 20 m/sec) by between 58% and 66%.

- Normalized Routing Load (NRL)

  Figures 8.29 and 8.30 show the *Normalized Routing Load (NRL)* for both dynamic and static networks when we vary the speed of nodes.

  The simulation results show that AODV demonstrates significantly the highest routing load. We expected this, since AODV is a single on-demand routing protocol, and the control packets are transmitted to the entire network. MDSDV decreases the routing load by between 67% and 86% in dynamic networks (Figure 8.29), and by between 90% and 92% in static networks (Figure 8.30).

  On the other hand, MDSDV and DSR produce similar routing loads in dynamic networks, whereas MDSDV decreases the routing load of DSR by 46% to 55% in static networks.

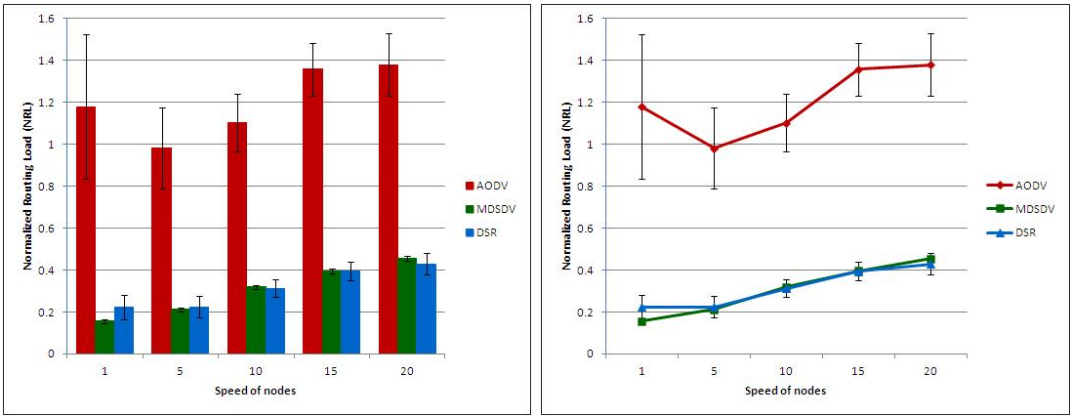

Figure 8.29: NRL vs Speed of Nodes (Dynamic network)

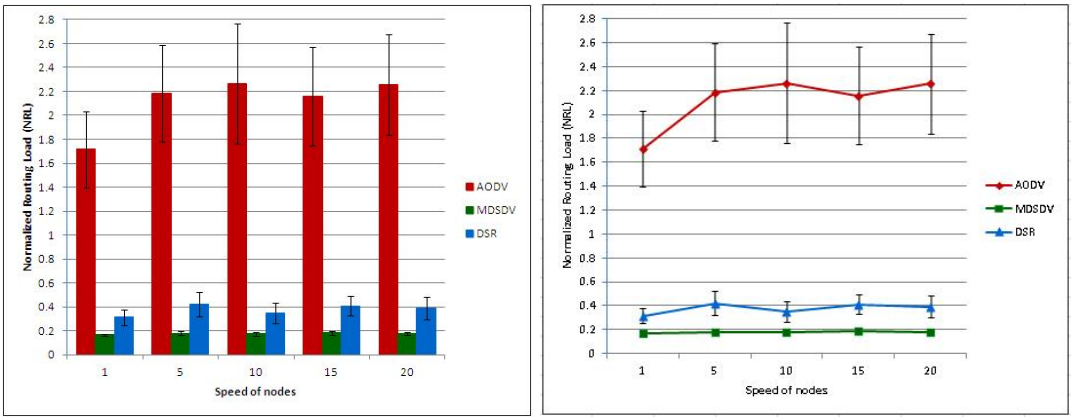

Figure 8.30: NRL vs Speed of Nodes (Static network)

- Data Packets Dropped

  Figures 8.31 and 8.32 show the number of data packets that are dropped by the three protocols. The figures show that AODV drops more data packets than MDSDV and DSR in all cases. MDSDV dropped between 10% and 58% of the data dropped by AODV in dynamic networks (Figure 8.31), and dropped between 12% and 18% of the data dropped by AODV in static networks (Figure 8.32).

  Figure 8.31 show that MDSDV and DSR drop similar data packets in dynamic networks. Whereas Figure 8.32 show that MDSDV dropped between 54% and 62% of the data dropped by DSR in static networks at high speed.

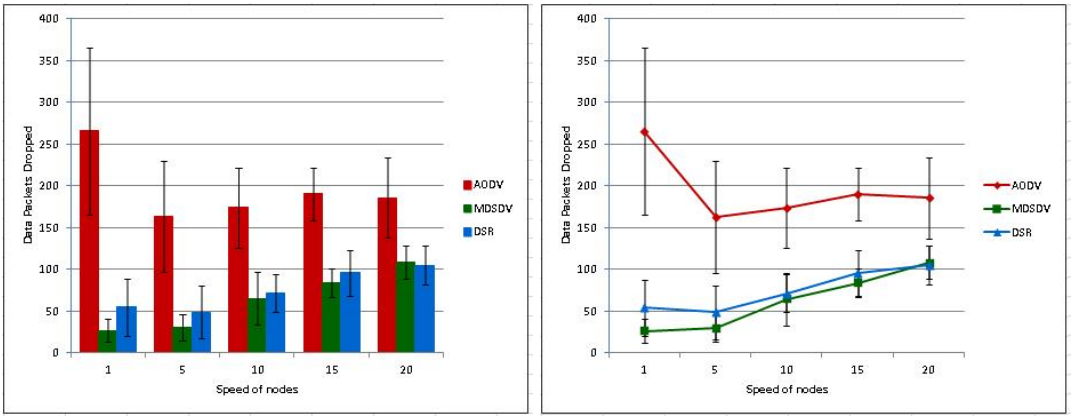

Figure 8.31: Data Dropped vs Speed of Nodes (Dynamic network)

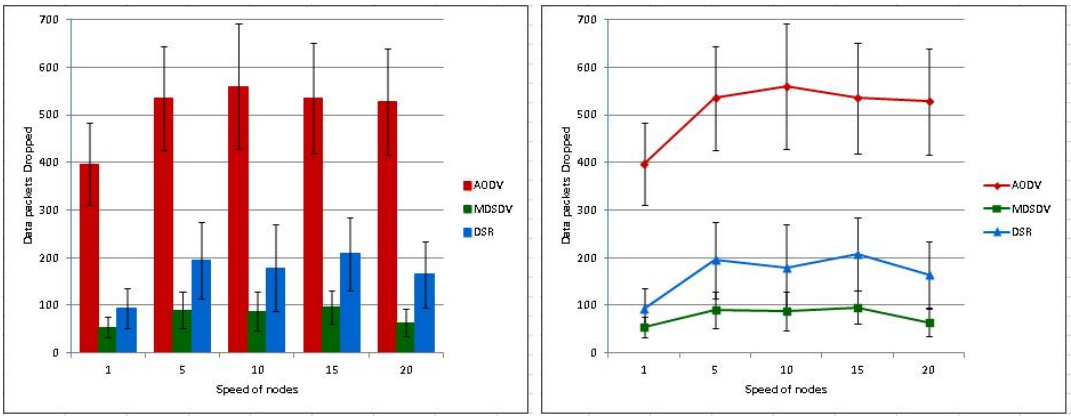

Figure 8.32: Data Dropped vs Speed of Nodes (Static network)

8.3 Summary

This chapter presented a performance comparison of MDSDV with AODV and DSR routing protocols for ad hoc networks using NS-2 simulations. The simulation results show important differences between the three routing protocols. The presence of high mobility implies frequent link failures and each routing protocol reacts differently to link failures. MDSDV uses proactive routing with multiple routes per destination stored in the routing table. In contrast, AODV and DSR use on-demand route discovery, but with different routing mechanics [43]. AODV stores at most one route per destination in its routing table. Thus, AODV has to initiate route discovery when a

link failure is detected. The destination sequence numbers are used to prevent loops and to distinguish freshness of routes. DSR uses source routing and route caches, and does not use periodic advertisements. DSR exploits caching aggressively and maintains multiple routes per destination. Thus, DSR uses route discovery less often than AODV when there are link failures. Route discovery is delayed in DSR until all cached routes fail.

The differences in the mechanics of the protocols lead to the differences in their performances. The performance comparison of MDSDV, AODV, and DSR were measured with respect to four metrics: *Packet Delivery Fraction*, *Average End-to-End delay*, *Normalized Routing Load*, and *Data Packets Dropped* in four different scenarios: varying number of sources, varying number of nodes, varying pause time, and varying node speed. From our observations and results, we conclude:

1. *Packet Delivery Fraction*: the simulation results show that the three protocols have similar performance when the number of traffic sources is low (10 or 20 sources), however MDSDV outperforms AODV and DSR in networks with 30, 40, and 50 traffic sources (Figures 8.1 and 8.2). The difference in performance increases as number of sources increases. This is because MDSDV is proactive and expects to have at least one path for each destination available for immediate use. In contrast AODV and DSR are reactive, and hence the number of route discoveries is directly proportional to the number of sources. Using AODV or DSR, a node has to invoke the route discovery process whenever a new route is needed.

   The network size has an impact on performance of the protocols. MDSDV performs better than the other two protocols, and the difference in performance increases as the number of nodes increases (Figures 8.9 and 8.10). Mobility has less impact on the behaviour of MDSDV, and hence it performs better than AODV and DSR in all cases (Figures 8.17, 8.18, 8.25, and 8.26). The difference in performance decreases as the mobility increases(Figures 8.18 and 8.25). This is because, in low mobility, less Full Dumps and Error Packets are

transmitted which gives a better chance for data packets to be delivered. Also, the availability of alternative paths that can be used in case of link failure, is a main reason for the high delivery ratio.

2. *Average End-to-End Delay*: Compared to DSR, MDSDV incurs lower packet delays in most cases (e.g., Figures 8.12, 8.19, 8.20, 8.27, and 8.28). The main reason for this is the lack of a mechanism that could expire unused routes from caches in DSR, together with the aggressive use of caching [16] [102]. In contrast, the existence of fresh routes helps MDSDV to reduces the delay of DSR. On the other hand MDSDV and AODV demonstrates similar delays in most cases (e.g., Figures 8.19, 8.20, 8.27, and 8.28). MDSDV incurs lower delay than AODV in dense networks (Figure 8.12). This is because it is possible to demonstrate availability of multiple paths, since there are many nodes that offer alternative routes.

3. *Normalized Routing Load*: AODV has the highest routing overhead in all cases (Figures 8.5, 8.6, 8.13, 8.14, 8.21, 8.22, 8.29, and 8.30), because it is a single path routing protocol, and because it floods route requests throughout the entire network whenever a node needs a route to a destination. Compared to DSR, MDSDV has a lower routing overhead in low mobility networks (Figures 8.14, 8.21, and 8.30), and has a similar routing overhead in dynamic networks where the pause time is 0 sec (Figures 8.13 and 8.29). MDSDV and DSR compete with each other when the traffic load is concerned. MDSDV has a lower routing overhead than DSR at a high traffic load, and has a higher routing overhead at a low traffic load (Figures 8.5 and 8.6). One interesting observation is that the overhead of MDSDV decreases as the number of sources increases, whereas the overhead of DSR increases as the number of sources increases (Figure 8.5). An other observation is that MDSDV has approximately constant overhead in static networks (Figure 8.6), regardless of the offered traffic load.

4. *Data Packets Dropped*: Our results show that AODV drops more data pack-

ets than the other two protocols in all cases (Figures 8.15, 8.16, 8.23, 8.24, 8.31, and 8.32). Compared to DSR, MDSDV drops similar data packets in high mobility networks (Figures 8.24, and 8.31), and drops less data packets in low mobility networks (Figures 8.16). This is because MDSDV always uses the shortest and newest route. In contrast, DSR does not have a mechanism to distinguish stale routes. Using stale routes may lead to data packets being dropped. Meanwhile, dropping packets might be as a result of a significant amount of collisions that occur in congested networks [123]. The effective protocol limits the number of rebroadcasts in the network to limit the probability of collisions.

Although MDSDV and DSR maintain multiple paths for a destination, the mechanisms used in the protocols are different (Section 3.4.1, Section 3.4.2, and Section 5.4). DSR broadcasts a route request to find a route for a certain destination. According to the received route reply packets, DSR stores multiple paths for that destination in its cache. DSR does not seek a new route until all cached routes fail, and this may lead to the use of a stale route. In contrast, MDSDV benefits from periodic Update Packets and Full Dumps when discovering a new neighbour to find new routes and refresh stale routes. MDSDV always use the newest and shortest path to transmit data packets. Additionally, when using MDSDV, a node that discovers a link failure broadcasts an Error Packet only to its neighbours. This leads to a reduction in the control overhead.

In summary, MDSDV performs better than AODV and DSR in almost all cases. One of the few exceptions is on small networks (e.g., 30 node networks) (Figure 8.9). This is due to the few opportunities it has to find multiple paths for a destination, since there are few nodes that offer them. Moreover, MDSDV uses a node-disjoint mechanism where the paths for each destination have no common nodes. When using a node-disjoint mechanism, routes are the least abundant and are the hardest to find [81].

Chapter 9

Conclusion

This chapter outlines the summary of the thesis in Section 9.1. Section 9.2 describes the limitations of MDSDV. Finally, Section 9.3 focuses on promising future research directions based on our research.

9.1 Summary

Routing is an important and challenging issue in mobile ad hoc networks (MANETs). Since the transmission is wireless and nodes are free to move, MANETs challenge the design of routing protocols. Routes frequently break due to interference and node mobility, and nodes have limited resources such bandwidth, energy, and processing power. These problems make multipath routing an interesting possibility, and a number of multipath routing protocols [28][53][58][59][63][75][122][124] have been proposed and implemented for MANETs.

Multipath routing protocols maintain multiple paths, and may be used for different purposes, such as increasing fault-tolerance, load balancing, minimising end-to-end delay, enhancing reliability [117]. Multipath routing protocols can provide fault tolerance by having more information routed to the destination using alternative paths.

This reduces the probability that communication is disrupted when link failure is occurred. To the best of our knowledge, most multipath routing algorithms are reactive, where the route is established only when a source node needs to send data to an intended receiver. For this reason, we have designed and implemented a new proactive multipath routing protocol that is based on the well known Destination Sequenced Distance Vector (DSDV) protocol [94].

The major contributions of this thesis are summarized as follows.

Preliminary MDSDV Design: We have designed a new proactive multipath protocol MDSDV that maintains multiple loop free routes between all source and destination nodes (Chapter 4). The preliminary version of MDSDV is called MDSDV0. It ensures that alternate paths at every node are node-disjoint. Node-disjoint paths means the paths do not have common nodes, except for the source and the destination, and hence that routes fail independently of each other. Two additional fields, *second hop* and *link-id*, are stored in the route entry to help address the problems of loop freedom, and path disjointness. In MDSDV0, exchanging routing information between new neighbours and broadcasting both Update packets and Error Packets to the entire network, leads to low performance. The poor performance of MDSDV0 is mainly attributed to the huge number of control packets it generates (Appendix A).

Revised MDSDV Design: To reduce the control overhead while maintaining flexibility and reliability, a revised protocol design is presented in Chapter 5. (i) Only the node that discovers a new neighbour unicasts a Full Dump of its routing table to the new neighbour, (ii) the Update Packets and Error Packets are only broadcast to the current neighbours (i.e., rebroadcasting is not needed). A rigorous argument is presented in Section 5.5 that the paths are indeed node disjoint.

Control Overhead: We investigate the control overheads in Chapter 6. To alleviate the congestion and bottlenecks, MDSDV does not rebroadcast routing packets.

As MDSDV uses five different control packets (i.e., Hello Message, Update Packets, Full Dumps, Error Packets, and Failure Packets), we divided the investigation into six parts. The first part investigates the total number of control packets generated for routing. Each part of the other five parts investigates one of the control packets.
The results show that the number of control packets increases as the mobility increases. The major contribution to MDSDV's routing load overhead is from *Full Dumps* and *Error Packets*. This is because as the mobility increases, the probability of meeting a new neighbour or of losing contact with an old neighbour increases. As a result, the node has to unicast a *Full Dump* or broadcast an *Error Packet*.
The results show that the number of *Hello Messages* and *Failure Packets* is very low. The number of *Hello Messages* is low, because the node broadcasts a *Hello Message* only when it has no neighbours. The number of *Failure Packets* is low, because the node unicasts a *Failure Packet* only when it fails to forward a data packet to the node that is specified in the header of the data packet.
Also, the results show that the mobility has no impact on the number of *Update Packets*, because they are time-driven updates.

Evaluation: Comparison with a proactive routing protocol: The results in Chapter 7 show that the performance of MDSDV is superior to standard DSDV. The results show that MDSDV is robust and improves the Packet Delivery Fraction of DSDV (up to 32%) especially in dynamic networks (Figure 7.17), reduces the Average end-to-end Delay of DSDV by about 7% in low mobility environments (Figures 7.11 and 7.20), provides lower routing load in low mobility (Figures 7.13 and 7.22), and dramatically decreases the number of dropped packets in all cases.

Unlike MDSDV, DSDV can not adapt to fast topology changes, because it is a single path routing protocol. When a link failure occurs, DSDV has to wait for a period of time to get a new information. On the other hand MDSDV can immediately use an alternative path when a link failure occurs. In contrast, in high mobility networks, MDSDV has a longer delay (Figures 7.3, 7.12, and 7.19) and imposes a higher routing

load than DSDV (Figure 7.21).

Evaluation: Comparison with reactive single & multipath routing protocol: The results in Chapter 8 show that MDSDV has similar performance to AODV and DSR at light traffic load (i.e., 10 and 20 sources). In contrast, it performs better than AODV and DSR in heavy traffic situations. Moreover, the difference increases as the traffic load increases. This is because MDSDV is proactive and has multiple paths for each destination that are available for immediate use. Using AODV or DSR, a node has to invoke the route discovery process whenever a new route is needed. This leads to an increase in the number of control packets.

Since the three protocols deliver similar data packets at low traffic situations, we used 30 traffic sources to evaluate the performance difference between them. MDSDV performs better than AODV and DSR in almost all cases in terms of Packet Delivery Fraction especially in dense networks and in low mobility situations. The difference increases as the network size increases or the mobility decreases. One of the few exceptions is in small networks (e.g., 30 node networks) (Figure 8.9), where MDSDV has a similar or slightly lower Packet Delivery Fraction.

MDSDV has a lower delay than DSR in almost all cases. The reduction in delay is up to 67% in dynamic networks (Figure 8.11), and up to 81% in static networks (Figure 8.12). The only exceptions are in 30 node networks where the two protocols have similar delays. Meanwhile, MDSDV reduces the delay of AODV by up to 60% in dynamic networks (Figure 8.27), and up to 48% in static networks (Figure 8.28). The difference in performance increases as the mobility decreases.

MDSDV and DSR have similar overhead in dynamic networks, but MDSDV reduces the overhead of DSR by up to 55% in static networks (Figure 8.30). Meanwhile, MDSDV has much lower overhead than AODV in all cases (Figures 8.5, 8.6, 8.13, 8.14, 8.21, 8.22, 8.29, 8.30). It reduces the overhead of AODV by up to 86% in dynamic networks (Figure 8.29), and up to 92% in static networks (Figure 8.30).

In terms of dropped data packets, MDSDV drops less data packets than AODV in almost all cases. The only exception is on small dynamic networks (30 node networks) (Figure 8.15), where MDSDV drops slightly more packets than AODV. MDSDV drops between 10% and 58% of data packets dropped by AODV in dynamic networks (Figure 8.31), and between 12% and 18% in static networks (Figure 8.32). In contrast, Figure 8.31 shows that MDSDV and DSR drop similar data packets in high dynamic networks, whereas MDSDV drops between 38% and 57% of data dropped by DSR in static networks (Figure 8.32).

Performance Summary: From the simulation results in Chapter (7) and Chapter (8), we can conclude that MDSDV improves the packet delivery fraction of DSDV in all cases, reduces the delay in low mobility environments, provides lower routing load in low mobility environments, and dramatically decreases the number of dropped packets in all cases. Moreover, MDSDV performs better than AODV and DSR in heavy traffic loads. It reduces the delay of DSR and AODV, provides much less overhead than AODV in all cases and less overhead than DSR in static networks, and finally MDSDV drops less data packets than AODV in all cases and drops less data packets than DSR in static networks.

MDSDV can support packet loss and delay sensitive applications over MANETs. From the results we believe that MDSDV can be used in the domain of multimedia applications such as World Wide Web, e-mail, and video on demand. On the other hand, as we did not pay attention to the security, MDSDV is not suitable protocol for applications that need security such as the financial domain.

9.2 Limitations

The major cost of MDSDV is additional control packets in high mobility networks. The major contribution to MDSDV's routing load overhead is from *Full Dumps* and

Error Packets. In spite of broadcasting the Error Packets to only one hop neighbours, the number of these packets is high in dynamic networks.

The other limitation is that the performance of MDSDV is slightly lower than AODV and DSR in low density networks with high mobility. This is because of the few neighbours that can provide multiple paths for each destination. As the network density increases, the probability of multiple paths being available increases. Moreover, MDSDV uses the node-disjoint mechanism which means that all paths for a certain destination have no common nodes. As a result, MDSDV can provide a lower number of alternative paths for each destination.

9.3 Future Work

This section focuses on promising future research directions based on our research.

Sending packets via different paths simultaneously can increase the reliability of the transmission as the effect of link failures is reduced. MDSDV finds multiple node-disjoint paths to a destination and uses them only as backup paths. Additionally, using the multiple paths as a delivery mechanism might lead to higher delivery ratios or lower delay.

A major overhead in MDSDV is the large number of control packets when the network is highly mobile. Reducing the number of control packets is a fruitful avenue for future research. One of the possibilities for doing so is avoiding using the Error Packet. We could stop broadcasting Error Packets, and depend only on broadcasting Update packets and unicasting Full Dumps to update the routing tables.

The simulation results show that the performance of MDSDV is slightly lower than AODV and DSR in sparse networks (e.g., 30 node networks). In addition to the low number of neighbours that offer multiple paths, using a node-disjoint technique can be another reason that makes MDSDV performs slightly lower than AODV and DSR

in sparse networks. In future work we could see whether performance is increased in such cases by using non-disjoint or link-disjoint techniques instead of using a node-disjoint technique, because node-disjoint routes are the least abundant and are the hardest to find.

We have only evaluated MDSDV using a random way point mobility model and CBR/UDP traffic. It would be useful to see how improvements vary with other mobility models and other traffic types such as TCP.

Additional information could be obtained by investigating in particular longer simulation times, larger areas, or other area shapes (e.g., rectangular areas).

Even though this thesis concentrated on designing and evaluating a multipath proactive routing protocol that is based on DSDV, some of the ideas in this thesis can be applied to other ad hoc routing protocols. For instance, we could easily modify the AODV protocol to maintain multiple loop-free paths using the *second hop* and *link-id* concept.

References

[1] Scalable network technologies, qualnet simulator. Available from: http://www.scalable-networks.com, 2003.

[2] M. Abolhasan, T. Wysocki, and E. Dutkiewicz. A review of routing protocols for mobile ad hoc networks. *Ad Hoc Networks*, 2(1):1–22, 2004.

[3] S. Ahn and A.U. Shankar. Adapting to route-demand and mobility in ad hoc network routing. *Computer Networks*, 38(6):745–764, April 2002.

[4] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *Proceedings of the 1st ACM workshop on Wireless security*, pages 21–30. ACM New York, NY, USA, 2002.

[5] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H.Y. Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer Magazine*, 31(10):77–85, 1998.

[6] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla. GloMoSim: A scalable network simulation environment. *Technical Report 990027, UCLA Computer Science Department*, May 1999.

[7] R. Barr, Z.J. Haas, and R. van Renesse. JiST: An efficient approach to simulation using virtual machines. *Software: Practice and Experience*, 35(6):539–576, 2005.

[8] R. Barr, Z.J. Haas, and R. van Renesse. Scalable Wireless Ad Hoc Network Simulation. *Handbook on theoretical and algorithmic aspects of sensor, ad hoc wireless, and peer-to-peer networks, Chapter 19*, pages 297–311, Auerbach, 2005.

[9] S. Basagni, I. Chlamtac, V.R. Syrotiuk, and B.A. Woodward. A distance routing effect algorithm for mobility (DREAM). In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 76–84. ACM New York, NY, USA, 1998.

[10] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang. MACAW: a media access protocol for wireless LAN's. In *Proceedings of the SIGCOMM 94 conference on Communications architectures, protocols and applications*, pages 212–225, London, United Kingdom, August 1994.

[11] E. Biagioni and S.H. Chen. A reliability layer for ad-hoc wireless sensor network routing. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, volume 37, pages 4799–4806, United States, January 2004.

[12] A. Boukerche and S.K. Das. Congestion control performance of R-DSDV protocol in multihop wireless ad hoc networks. *Wireless Networks*, 9(3):261–270, 2003.

[13] A. Boukerche, S.K. Das, and A. Fabbri. Analysis of a Randomized Congestion Control Scheme with DSDV Routing in ad Hoc Wireless Networks. *Journal of Parallel and Distributed Computing*, 61(7):967–995, July 2001.

[14] A. Boukerche and S. Rogers. Gps query optimization in mobile and wireless networks. In *Proceedings of 6th IEEE Symposium on Computers and Communications (ISCC 2001)*, pages 198–203, Tunisia, July 2001.

[15] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 85–97, October 1998.

[16] I. Broustis, G. Jakllari, T. Repantis, and M. Molle. A performance comparison of routing protocols for large-scale wireless mobile Ad Hoc networks. In

IEEE International Workshop on Wireless Ad Hoc and Sensor Networks (in conjunction with SECON), New York, USA, 2006.

[17] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless communications and mobile computing*, 2(5):483–502, 2002.

[18] S. Čapkun, M. Hamdi, and J.P. Hubaux. GPS-free positioning in mobile ad hoc networks. *Cluster Computing*, 5(2):157–167, 2002.

[19] ID Chakeres and EM Belding-Royer. AODV routing protocol implementation design. In *Proceedings of 24th International Conference on Distributed Computing Systems Workshops*, pages 698–703, Hachioji, Japan, March 2004.

[20] S. Chang, W. Ting, and J. Chen. Method for Reducing Routing Overhead for Mobile Ad Hoc Network. In *Proceedings of IEEE International Conference on Wireless Communications and Signal Processing (WCSP 2010)*, Suzhou, China, October 2010.

[21] T.W. Chen and M. Gerla. Global state routing: A new routing scheme for ad-hoc wireless networks. In *Proceedings of IEEE International Conference on Communications (ICC'98)*, volume 1, pages 171–175, Atlanta, GA, USA, June 1998.

[22] C.C. Chiang, H.K. Wu, W. Liu, and M. Gerla. Routing in clustered multihop, mobile wireless networks with fading channel. In *Proceedings of IEEE SICON'97*, volume 97, pages 197–211, April 1997.

[23] T. Clausen and P. Jacquet. RFC3626: Optimized Link State Routing Protocol (OLSR). *http://www.faqs.org/rfcs/rfc3626.html*, October 2003.

[24] IEEE Computer Society LAN MAN Standards Committee. *Wireless LAN Medium Access Control (MAC) and Physical Layer(PHY) Specifications, IEEE Standard 802.11-1997*. The Institute of Electrical and Electronics Engineers, New York 1997.

[25] S. Corson and J. Macker. RFC2501: Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations. *http://www.ietf.org/rfc/rfc2501.txt*, January 1999.

[26] F. Dai and J. Wu. A highly reliable multi-path routing scheme for ad hoc wireless networks. *International Journal of Parallel, Emergent and Distributed Systems*, 20(3):205–219, 2005.

[27] F. Dai and J. Wu. Proactive route maintenance in wireless ad hoc networks. In *Proceedings of IEEE International Conference on Communications (ICC 2005)*, volume 2, pages 1236–1240, Seoul, Korea, May 2005.

[28] A. Darehshoorzadeh, N.T. Javan, M. Dehghan, and M. Khalili. Lbaodv: A new load balancing multipath routing algorithm for mobile ad hoc networks. In *Proceedings of IEEE 2008 6th National Conference on Telecommunication Technologies and IEEE 2008 2nd Malaysia Conference on Photonics, NCTT-MCP 2008*, pages Putrajaya, Malaysia, August 344–349.

[29] G.A. Di Caro. Analysis of simulation environments for mobile ad hoc networks. Technical report, IDSIA-24-03, Dalle Molle Institute for Artifical Intelligence, Switzerland, December 2003.

[30] S. Doshi, S. Bhandare, and T.X. Brown. An on-demand minimum energy routing protocol for a wireless ad hoc network. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(3):50–66, 2002.

[31] J.M. Dricot and P. De Doncker. High-accuracy physical layer model for wireless network simulations in NS-2. In *Proceedings of the International Workshop on Wireless Ad-Hoc Networks (IWWAN 2004)*, pages 249–253, Finland, 2004.

[32] A. Etorban, P.J.B King, and P. Trinder. A performance comparison of MDSDV with AODV and DSDV routing protocols. In *Proceedings of the*

25th UKPEW2009, Performance Engineering Workshop, pages 144–155. University of Leeds, United Kingdom, July 2009.

[33] K. Fall and K. Varadhan. The ns Manual (formerly ns Notes and Documentation). *The VINT project*, http://www.isi.edu/nsnam/, 2002.

[34] C.L. Fullmer and J.J Garcia-Luna-Aceves. Floor Acquisition Multiple Access (FAMA) for packet-radio networks. *Proceedings of ACM SIGCOMM'95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 25(4):262–273, Cambridge, MA, United states, September 1995.

[35] Y. Ge, G. Wang, W. Jia, and Y. Xie. Node-disjoint multipath routing with zoning method in manets. In *Proceedings of 10th IEEE International Conference on High Performance Computing and Communications (HPCC'08)*, pages 456–462, Dalian, China, September 2008.

[36] Z.J Haas. A new routing protocol for the reconfigurable wireless networks. In *Proceedings of the 6th IEEE International Conference on Universal Personal Communications (ICUPC'97)*, volume 2, pages 562–566, San Diego, CA, USA, October 1997.

[37] H. Hassanein and A. Zhou. Load-aware destination-controlled routing for MANETs. *Computer Communications*, 26(14):1551–1559, 2003.

[38] G. He. Destination-sequenced distance vector (DSDV) protocol. Technical report, Networking Laboratory, Helsinki University of Technology, Finland, 2002 Available: http://keskus.hut.fi/opetus/s38030/k02/Papers/03-Guoyou.pdf.

[39] X. Hong, M. Gerla, G. Pei, and C.C. Chiang. A group mobility model for ad hoc wireless networks. In *Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 53–60, USA, August 1999.

[40] Y.C. Hu, D.B. Johnson, and A. Perrig. SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks. *Ad Hoc Networks*, 1(1):175–192, 2003.

[41] Y.C. Hu, A. Perrig, and D.B. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. *Wireless Networks*, 11(1):21–38, 2005.

[42] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Proceedings of IEEE Multi Topic Conference (INMIC 2001)*, pages 62–68, Pakistan, December 2001.

[43] G. Jayakumar and G. Ganapathy. Performance comparison of mobile ad-hoc network routing protocol. *International Journal of Computer Science and Network Security (IJCSNS)*, 7(11):77–84, 2007.

[44] M. Joa-Ng and I. Lu. A peer-to-peer zone-based two-level link state routing for mobile ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1415–1425, 1999.

[45] P. Johansson, T. Larsson, N. Hedman, B. Mielczarek, and M. Degermark. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 195–206, USA, August 1999.

[46] D.B. Johnson, J. Broch, Y.C. Hu, J. Jetcheva, and D.A. Maltz. The CMU Monarch projects wireless and mobility extensions to ns. In *Proceedings of the 42nd Internet Engineering Task Force Conference*, August 1998.

[47] D.B. Johnson, D.A. Maltz, J. Broch, et al. DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks. *Ad hoc networking*, 5:139–172, 2001.

[48] E.S. Jung and N.H. Vaidya. A power control MAC protocol for ad hoc networks. *Wireless Networks*, 11(1):55–66, 2005.

[49] P. Karavetsios and A. Economides. Performance comparison of distributed routing algorithms in ad hoc mobile networks. *WSEAS Transactions on Communications*, 3(1):317–321, 2004.

[50] F. Kargl and E. Schoch. Simulation of MANETs: A Qualitative Comparison between JiST/SWANS and NS-2. In *Proceedings of the 1st international workshop on System evaluation for mobile platforms*, pages 41–46, San Juan, Puerto Rico, June 2007.

[51] P. Karn. MACA-a new channel access method for packet radio. In *ARRL/CRRL Amateur radio 9th computer networking conference*, pages 134–140, September 1990.

[52] K.U.R. Khan, R.U. Zaman, A.V. Reddy, K.A. Reddy, and T. Harsha. An Efficient DSDV Routing Protocol for Wireless Mobile Ad Hoc Networks and its Performance Comparison. In *Proceedings of the 2nd UKSIM European Symposium on Computer Modeling and Simulation*, pages 506–511, Liverpool, United kingdom, September 2008.

[53] M. Khazaei and R. Berangi. A multi-path routing protocol with fault tolerance in mobile ad hoc networks. In *Proceedings of the 14th International CSI Computer Conference (CSICC 2009)*, pages 77–82, Tehran, Iran, October 2009.

[54] P.J.B. King, A. Etorban, and I.S. Ibrahim. A DSDV-based multipath routing protocol for mobile ad-hoc networks. In *Proceedings of the 8th Annual PostGraduate Symposium on The Convergence of Telecommunications, Networking and Broadcasting*, pages 93–98, The School of Computing and Mathematical Sciences, Liverpool John Moores University, United Kingdom, 2007.

[55] K. Kuladinithi, A. Udugama, and C. Görg. On demand self organising ad hoc networks-Implementation architectures. In *12th WWRF meeting*, Toronto, Canada, 2004.

[56] T. Kullberg. Performance of the ad-hoc on-demand distance vector routing protocol. In *HUT T-110.551 Helsinki University of Technology Seminar on Internetworking*, Finland, April 2004.

[57] S. Kumar, RK Rathy, and D. Pandey. OPR: DSDV Based New Proactive Routing Protocol for Ad-Hoc Networks. In *International Association of Computer Science and Information Technology - Spring Conference (IACSIT-SC 2009)*, pages 204–207, Singapore, April 2009.

[58] S.J. Lee and M. Gerla. AODV-BR: Backup routing in ad hoc networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2000)*, volume 3, pages 1311–1316, Chicago, IL, USA, September 2000.

[59] S.J. Lee and M. Gerla. Split multipath routing with maximally disjoint paths in ad hoc networks. In *Proceedings of IEEE International Conference on Communications (ICC 2001)*, volume 10, pages 3201–3205, Helsinki, Finland, June 2001.

[60] U. Lee, S.F. Midkiff, and J.S. Park. A proactive routing protocol for multichannel wireless ad-hoc networks (DSDV-MC). In *Proceedings of IEEE International Conference on Information Technology: Coding and Computing (ITCC 2005)*, volume 2, pages 710–715, Las Vegas, NV, United states, April 2005.

[61] J. Li, C. Blake, D.S.J. De Couto, H.I. Lee, and R. Morris. Capacity of ad hoc wireless networks. In *Proceedings of the 7th ACM annual international conference on Mobile computing and networking*, pages 61–69, Rome, Italy, July 2001.

[62] Q. Li, M. Zhao, J. Walker, Y.C. Hu, A. Perrig, and W. Trappe. Sear: a secure efficient ad hoc on demand routing protocol for wireless networks. *Security and Communication Networks*, 2(4):325–340, 2009.

[63] X. Li and L. Cuthbert. Stable node-disjoint multipath routing with low overhead in mobile ad hoc networks. In *Proceedings of Proceedings - IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS 2004*, pages 184–191, Volendam, Netherlands, October 2004.

[64] Yan Li. *Mobile Ad Hoc Network Simulation: Analysis and Enhancements*. PhD thesis, Heriot-Watt University, Edinburgh, UK, 2008.

[65] B. Liang and Z.J. Haas. Predictive distance-based mobility management for PCS networks. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1999)*, volume 3, pages 1377–1384, New York, NY , USA, March 1999.

[66] B. Liang and Z.J. Haas. Predictive distance-based mobility management for multidimensional PCS networks. *IEEE/ACM Transactions on Networking (TON)*, 11(5):718–732, October 2003.

[67] C.H. Lin, W.S. Lai, Y.L. Huang, and M.C. Chou. I-sead: A secure routing protocol for mobile ad hoc networks. In *Proceedings of the International Conference on Multimedia and Ubiquitous Engineering (MUE 2008)*, pages 102–107, Busan, Korea, April 2008.

[68] X. Lin and I. Stojmenovic. Location-based localized alternate, disjoint and multi-path routing algorithms for wireless networks. *Journal of Parallel and Distributed Computing*, 63(1):22–32, 2003.

[69] C. Liu and J. Kaiser. A survey of mobile ad hoc network routing protocols. *Technical report No. 2003-08, University of Magdeburg, Germany*, 2005, http://www.minema.di.fc.ul.pt/papers.html.

[70] T. Liu and K. Liu. Improvements on DSDV in mobile ad hoc networks. In *Proceedings of International Conference on Wireless Communications, Networking and Mobile Computing (WiCom 2007)*, pages 1637–1640, Shanghai, China, September 2007.

[71] T.T. Luong, B.S. Lee, and C.K. Yeo. Dual-interface multiple channels dsdv protocol. In *Proceedings of 5th IEEE International Conference on Wireless and Mobile Computing Networking and Communication, WiMob 2009*, pages 104–109, Marrakech, Morocco, October 2009.

[72] T.T. Luong, B.S. Lee, and C.K. Yeo. Channel allocation for multiple channels multiple interfaces communication in wireless ad hoc networks. In *Proceedings of the 7th international IFIP-TC6 networking conference on AdHoc and sensor networks*, pages 87–98, Singapore, May 2008.

[73] J. Macker et al. Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations, rfc2501. January 1999, available at http://www.ietf.org/rfc/rfc2501.txt.

[74] M. Maleki, K. Dantu, and M. Pedram. Power-aware source routing protocol for mobile ad hoc networks. In *Proceedings of the 2002 international symposium on Low power electronics and design*, pages 72–75, Monterey, CA, United states, August 2002.

[75] M.K. Marina and S.R. Das. Ad hoc on-demand multipath distance vector routing. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(3):92–93, 2002.

[76] M.K. Marina and S.R. Das. Ad hoc on-demand multipath distance vector routing. *Wireless Communications and Mobile Computing*, 6(7):969–988, 2006.

[77] M.K. Marina and S.R. Das. On-demand multipath distance vector routing in ad hoc networks. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, pages 14–23, Riverside, CA, United states, November 2001.

[78] S. Marti, TJ Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking (MOBICOM 2000)*, pages 255–265, Boston, MA, USA, August 2000.

[79] M. Mohammadizadeh, A. Movaghar, and S.M. Safi. SEAODV: Secure Efficient AODV Routing Protocol for MANETs Networks. In *Proceedings of 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, volume 403, pages 940–944, Seoul, Korea, November 2009.

[80] A.A.S. Mohammed and S.I. Abdul. Internet compressed traffic: A solution for the explosion of the internet. *International Journal of Computer Applications*, 21(6):12–15, 2011.

[81] S. Mueller, R.P. Tsang, and D. Ghosal. Multipath routing in mobile ad hoc networks: Issues and challenges. *Performance Tools and Applications to Networked Systems*, 2965 of Lecture Notes in Computer Science:209–234, 2004.

[82] S. Murthy and J.J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mobile Networks and Applications*, 1(2):183–197, 1996.

[83] S. Murthy and JJ Garcia-Luna-Aceves. A routing protocol for packet radio networks. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 86–95, Berkeley, CA, USA, November 1995.

[84] S. Narayanaswamy, V. Kawadia, R.S. Sreenivas, and P.R. Kumar. Power control in ad-hoc networks: Theory, architecture, algorithm and implementation of the COMPOW protocol. In *Proceedings of European Wireless Conference, Next Generation Wireless Networks: Technologies, Protocols, Services and Applications*, pages 156–162, Italy, February 2002.

[85] N. Nikaein, C. Bonnet, and N. Nikaein. Harp-hybrid ad hoc routing protocol. In *Proceedings of IST: International Symposium on Telecommunications*, Tehran, Iran, September 2001.

[86] N. Nikaein, H. Labiod, and C. Bonnet. DDR: distributed dynamic routing algorithm for mobile ad hoc networks. In *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing*, pages 19–27, Boston, MA, USA, August 2000.

[87] R. Ogier, F. Templin, and M. Lewis. RFC3684: Topology Dissemination Based on Reverse-Path Forwarding (TBRPF). February 2004, Available: http://www.ietf.org/rfc/rfc3684.txt.

[88] P. Papadimitratos and Z.J. Haas. Secure link state routing for mobile ad hoc networks. In *Proceedings of IEEE Workshop on Security and Assurance in Ad hoc Networks, in conjunction with the 2003 International Symposium on Applications and the Internet*, pages 379–383, Orlando, FL, January 2003.

[89] P. Papadimitratos and Z.J. Haas. Secure routing for mobile ad hoc networks. In *Proceedings of SCS Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002)*, pages 27–31, San Antonio, TX, January 2002.

[90] V.D. Park and M.S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of the 16th IEEE Annual Conference on Computer Communications, INFOCOM. Part 1 (of 3)*, volume 3, pages 1405–1413, April 1997.

[91] G. Pei, M. Gerla, and T.W. Chen. Fisheye state routing: A routing scheme for ad hoc wireless networks. In *Proceedings of IEEE International Conference on Communications*, volume 1, pages 70–74, New orleans, LA, USA, June 2000.

[92] G. Pei, M. Gerla, and T.W. Chen. Fisheye state routing in mobile ad hoc networks. In *Proceedings of the 2000 ICDCS Workshop on Wireless Networks and Mobile Computing*, pages D71–D78, Taiwan, April 2000.

[93] C. Perkins, E. Belding-Royer, and S. Das. RFC3561: Ad hoc on-demand distance vector (AODV) routing. July 2003, Available: http://www.ietf.org/rfc/rfc3561.

[94] C.E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. *ACM SIGCOMM Computer Communication Review*, 24(4):234–244, 1994.

[95] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *proceedings of 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, pages 90–100, New Orleans, LA, United states, February 1999.

[96] S. Radhakrishnan, G. Racherla, CN Sekharan, NSV Rao, and SG Batsell. DST-A routing protocol for ad hoc networks using distributed spanning trees. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 1999)*, pages 1543–1547, New Orleans, LA , USA, September 1999.

[97] A.H.A. Rahman and Z.A. Zukarnain. Performance Comparison of AODV, DSDV and I-DSDV Routing Protocols in Mobile Ad Hoc Networks. *European Journal of Scientific Research*, 31(4):566–576, 2009.

[98] R. Rajaraman. Topology control and routing in ad hoc networks: a survey. *ACM SIGACT News*, 33(2):60–73, 2002.

[99] V. Ramasubramanian, Z.J. Haas, and E.G. Sirer. SHARP: A hybrid adaptive routing protocol for mobile ad hoc networks. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, pages 303–314. Annapolis, MD, USA, June 2003.

[100] L. Reddeppa Reddy and SV Raghavan. SMORT: Scalable multipath on-demand routing for mobile ad hoc networks. *Ad Hoc Networks*, 5(2):162–188, 2007.

[101] J.W. Roberts. Variable-bit-rate traffic control in b-isdn. *IEEE Communications Magazine*, 29(9):50–56, 1991.

[102] E.M. Royer, S.R. Das, and M.K. Marina. Performance comparison of two on-demand routing protocols for ad hoc networks. *IEEE Personal Communications*, 8(1):16–28, 2001.

[103] EM Royer, PM Melliar-Smith, and LE Moser. An analysis of the optimum node density for ad hoc mobile networks. In *Proceedings of IEEE International Conference on Communications (ICC 2001)*, volume 3, pages 857–861, Helsinki, Finland, June 2001.

[104] E.M Royer and C.K. Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, 6(2):46–55, 1999.

[105] S. Saha, R. Chaki, and N. Chaki. A New Reactive Secure Routing Protocol for Mobile Ad-Hoc Networks. In *Proceedings of the 7th IEEE International Conference on Computer Information Systems and Industrial Management Applications (CISIM 2008)*, pages 103–108, Ostrava, Czech republic, June 2008.

[106] M. Sanchez, P. Manzoni, and Z.J. Haas. Determination of critical transmission range in ad-hoc networks. In *Proceedings of Workshop on Multiaccess, Mobility and Teletraffic for Wireless Communications (MMT'99)*, pages 293–304, Venice, Italy, October 1999.

[107] K. Sanzgiri, B. Dahill, BN Levine, C. Shields, and EM Belding-Royer. A secure routing protocol for ad hoc networks. In *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP02)*, pages 78–87, Paris, France, November 2002.

[108] S. Shah, A. Khandre, M. Shirole, and G. Bhole. Performance Evaluation of Ad Hoc Routing Protocols using NS2 Simulation. In *Proceedings of the National Conference on Mobile and Pervasive Computing (CoMPC-2008)*, pages 167–171, Chennai, India, August 2008.

[109] S. Singh, M. Woo, and CS Raghavendra. Power-aware routing in mobile ad hoc networks. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 181–190, Dallas, TX, October 1998.

[110] J.H. Song, V.W.S. Wong, and V. Leung. Secure position-based routing protocol for mobile ad hoc networks. *Ad Hoc Networks*, 5(1):76–86, 2007.

[111] M. Steenstrup. *Routing in Communications Networks*. Prentice Hall, Inc, Englewood Cliffs, NJ, 1995.

[112] Y. Tao and W. Luo. Modified energy-aware dsr routing for ad hoc network. In *Proceedings of the 2007 International Conference on Wireless Communications, Networking and Mobile Computing (WiCom 2007)*, pages 1601–1603, Shanghai, China, September 2007.

[113] M. Tarique, KE Tepe, and M. Naserian. Energy saving dynamic source routing for ad hoc wireless networks. In *Proceedings of 3rd International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks WIOPT'2005*, pages 305–310, Italy, April 2005.

[114] F. Tobagi, W. Noureddine, B. Chen, A. Markopoulou, C. Fraleigh, M. Karam, J.M. Pulido, and J. Kimura. Service differentiation in the internet to support multimedia traffic. *Evolutionary Trends of the Internet*, pages 381–400, 2001.

[115] C.K. Toh. Maximum battery life routing to support ubiquitous mobile computing in wireless ad hoc networks. *IEEE Communications Magazine*, 39(6):138–147, 2001.

[116] Ha Duyen Trung, Watit Benjapolakul, and Phan Minh Duc. Performance evaluation and comparison of different ad hoc routing protocols. *Computer Communications*, 30(11-12):2478 – 2496, 2007.

[117] J. Tsai and T. Moors. A review of multipath routing protocols: from wireless ad hoc to mesh networks. In *Proceedings of ACoRN Early Career Researcher Workshop on Wireless Multihop Networking*, pages 17–18, Australia, July 2006.

[118] T. Wan, E. Kranakis, and P.C. van Oorschot. Securing the Destination-Sequenced Distance Vector Routing Protocol (S-DSDV). In *Proceedings of the 6th International Conference on Information and Communications Security (ICICS04)*, volume 3269, pages 358–374, Malaga, Spain, October 2004.

[119] J.W. Wang, H.C. Chen, and Y.P. Lin. A Secure DSDV Routing Protocol for Ad Hoc Mobile Networks. In *Proceedings of 5th International Joint Conference on INC, IMS and IDC, (NCM 2009)*, pages 2079–2084, Seoul, Korea, August 2009.

[120] L. Wang, Y. Shu, M. Dong, L. Zhang, and O.W.W. Yang. Adaptive multipath source routing in ad hoc networks. In *Proceedings of IEEE International Conference on Communications (ICC01)*, volume 3, pages 867–871, Finland, June 2001.

[121] L. Wang, L. Zhang, Y. Shu, and M. Dong. Multipath source routing in wireless ad hoc networks. In *Proceedings of Canadian Conference on Electrical and Computer Engineering 2000*, volume 1, pages 479–483, Canada, May 2000.

[122] S. Wang, Q. Li, Y. Jiang, and H. Xiong. Stable on-demand multipath routing for mobile ad hoc networks. pages 318–321, Wuhan, China, November 2009.

[123] B. Williams and T. Camp. Comparison of broadcasting techniques for mobile ad hoc networks. In *Proceedings of the 3rd ACM international symposium on*

Mobile ad hoc networking & computing, pages 194–205, Switzerland, June 2002.

[124] N. Wisitpongphan and OK Tonguz. Disjoint Multi-Path Source Routing in ad hoc networks: transport capacity. In *Proceedings of the IEEE 58th Vehicular Technology Conference (VTC 2003)*, volume 4, pages 2207–2211, Orlando, Florida USA, October 2003.

[125] Z. Ye, S.V. Krishnamurthy, and S.K. Tripathi. A routing framework for providing robustness to node failures in mobile ad hoc networks. *Ad Hoc Networks*, 2(1):87–107, 2004.

[126] Z. Ye, S.V. Krishnamurthy, and S.K. Tripathi. A framework for reliable routing in mobile ad hoc networks. In *Proceedings of IEEE INFOCOM 2003, The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 270–280, San Francisco, CA, USA, April 2003.

[127] J. Yoon, M. Liu, and B. Noble. Random waypoint considered harmful. In *Proceedings of IEEE INFOCOM 2003, The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1312–1321, San Francisco, CA, USA, April 2003.

[128] M.G. Zapata. Secure ad hoc on-demand distance vector routing. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(3):106–107, 2002.

[129] M.G. Zapata and N. Asokan. Securing ad hoc routing protocols. In *Proceedings of the 2002 ACM workshop on Wireless security*, pages 1–10, Atlanta, GA, USA, September 2002.

[130] Z. Zhaoxiao, P. Tingrui, and Z. Wenli. Modified energy-aware aodv routing for ad hoc networks. In *Proceedings of the 2009 WRI Global Congress on Intelligent Systems, GCIS 2009*, volume 3, pages 338–342, Xiamen, China, May 2009.

[131] L. Zhou and ZJ Haas. Securing ad hoc networks. *IEEE Network Magazine*, 13(6):24–30, 1999.

[132] X. Zou, B. Ramamurthy, and S. Magliveras. Routing Techniques in Wireless Ad Hoc Networks Classification and Comparison. In *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics*, pages 1–6, Orlando, Florida, USA, July 2002.

Appendix A

MDSDV0 and MDSDV comparison Data

This appendix gives a brief comparison of MDSDV0 with MDSDV using simulations as discussed in section 4.6. In this experiment, four network sizes (30, 50, 70, and 90 node networks) and two pause times (0 and 100 seconds) are used. Nodes move with a maximum speed of 20 m/sec. Table A.1 lists the parameters used for the simulations, and the results comparison are listed in tables A.2 - A.9.

Parameter	Value
Simulator	NS-2
Simulation time	100 seconds
Area of the network	670m x 670m
Number of nodes	30, 50, 70, and 90 nodes
MAC layer	IEEE 802.11
Transmission range	250 m
Pause time	0 and 100 seconds
Maximum speed of nodes	20 m/s
Mobility model	Random waypoint
Traffic type	CBR (UDP)
Number of data sources	30 Sources
Packet size	512 byte
Transmission rate	4 packets/second
Bandwidth	2 Mb/s

Table A.1: Parameters used to compare MDSDV0 with MDSDV

A.1 30 Node Networks

Parameter	MDSDV0	MDSDV
Sent packets	4024	3976
Received packets	1907	3794
Packet Delivery Fraction (PDF)	**47.39**	**95.42**
Normalized Routing Load (NRL)	1.058	0.231
Average e-e delay(ms)	20.53	62.81
No. of dropped data (packets)	2047	155
Average Throughput[kbps]	80.22	159.52
Routing packets	**2017**	**878**
Hello packets	5	5
Available packets	351	0
Full_Dump packets	45	74
Update packets	953	353
Error packets	639	431
Failure packets	24	15

Table A.2: 30 node Network with 0 sec pause time (dynamic network)

Parameter	MDSDV0	MDSDV
Sent packets	3981	3988
Received packets	3592	3987
Packet Delivery Fraction (PDF)	**90.23**	**99.98**
Normalized Routing Load (NRL)	0.507	0.090
Average e-e delay(ms)	18.85	19.58
No. of dropped data (packets)	384	1
Average Throughput[kbps]	151.02	167.60
Routing packets	**1820**	**359**
Hello packets	4	3
Available packets	353	0
Full_Dump packets	62	3
Update packets	1398	350
Error packets	3	3
Failure packets	0	0

Table A.3: 30 node Network with 100 sec pause time (static network)

We observe that in dynamic network PDF increases from 47.39% to 95.42%, and control packets fall from 2017 to 878 (A.2). Likewise, for static networks PDF in-

creases from 90.23% to 99.98%, and control packets fall from 1820 to 359 (A.3). The remaining tables (A.4 - A.9) show very similar patterns.

A.2 50 Node Networks

Parameter	MDSDV0	MDSDV
Sent packets	4631	4616
Received packets	2347	4464
Packet Delivery Fraction (PDF)	**50.68**	**96.71**
Normalized Routing Load (NRL)	2.657	0.473
Average e-e delay(ms)	72.25	153.70
No. of dropped data (packets)	2218	133
Average Throughput[kbps]	98.66	187.65
Routing packets	**6237**	**2113**
Hello packets	4	3
Available packets	588	0
Full_Dump packets	74	150
Update packets	3159	588
Error packets	2331	1357
Failure packets	81	15

Table A.4: 50 node Network with 0 sec pause time (dynamic network)

Parameter	MDSDV0	MDSDV
Sent packets	4601	4586
Received packets	3601	4577
Packet Delivery Fraction (PDF)	**78.27**	**99.80**
Normalized Routing Load (NRL)	1.138	0.144
Average e-e delay(ms)	16.56	30.44
No. of dropped data (packets)	964	1
Average Throughput[kbps]	151.45	192.41
Routing packets	**4097**	**659**
Hello packets	4	4
Available packets	589	0
Full_Dump packets	74	26
Update packets	3422	591
Error packets	4	38
Failure packets	4	0

Table A.5: 50 node Network with 100 sec pause time (static network)

A.3 70 Node Networks

Parameter	MDSDV0	MDSDV
Sent packets	4600	4612
Received packets	2525	4421
Packet Delivery Fraction (PDF)	**54.89**	**95.86**
Normalized Routing Load (NRL)	6.619	0.838
Average e-e delay(ms)	311.22	135.38
No. of dropped data (packets)	2015	123
Average Throughput[kbps]	106.18	185.91
Routing packets	**16712**	**3705**
Hello packets	8	4
Available packets	827	0
Full_Dump packets	187	340
Update packets	10672	831
Error packets	4991	2530
Failure packets	27	0

Table A.6: 70 node Network with 0 sec pause time (dynamic network)

Parameter	MDSDV0	MDSDV
Sent packets	4629	4625
Received packets	3169	4509
Packet Delivery Fraction (PDF)	**68.46**	**97.49**
Normalized Routing Load (NRL)	2.009	0.220
Average e-e delay(ms)	19.73	137.35
No. of dropped data (packets)	1415	23
Average Throughput[kbps]	133.21	189.56
Routing packets	**6368**	**994**
Hello packets	3	3
Available packets	831	0
Full_Dump packets	92	64
Update packets	5437	823
Error packets	3	104
Failure packets	2	0

Table A.7: 70 node Network with 100 sec pause time (static network)

A.4 100 Node Networks

Parameter	MDSDV0	MDSDV
Sent packets	4577	4607
Received packets	1907	3993
Packet Delivery Fraction (PDF)	**41.67**	**86.672**
Normalized Routing Load (NRL)	13.483	1.678
Average e-e delay(ms)	74.55	423.37
No. of dropped data (packets)	2590	366
Average Throughput[kbps]	80.17	167.85
Routing packets	**25713**	**6702**
Hello packets	6	5
Available packets	1187	0
Full_Dump packets	234	531
Update packets	16215	1186
Error packets	8051	4978
Failure packets	20	2

Table A.8: 100 node Network with 0 sec pause time (dynamic network)

Parameter	MDSDV0	MDSDV
Sent packets	4630	4593
Received packets	2781	4391
Packet Delivery Fraction (PDF)	**60.07**	**95.60**
Normalized Routing Load (NRL)	5.70	0.33
Average e-e delay(ms)	34.60	252.59
No. of dropped data (packets)	1782	60
Average Throughput[kbps]	116.91	184.60
Routing packets	**15845**	**1437**
Hello packets	5	4
Available packets	1179	0
Full_Dump packets	150	112
Update packets	14307	1182
Error packets	197	135
Failure packets	7	4

Table A.9: 100 node Network with 100 sec pause time (static network)

Appendix B

Control Packets

This appendix presents the data that are used to produce all the figures in Chapter 6. The data included in the appendix represents the control packets that are generated by MDSDV as a function of Pause Time. Results are based on simulation of 30 runs, and the error bars represent the 95% confidence interval of the mean.
This Appendix is divided into six sections. The first section (Section B.1) represents the total control packets generated by MDSDV, whereas each section of the other five sections represents one type of control packet. Specifically, Sectin B.2 represents the Full Dumps, Sectin B.3 represents update packets, Sectin B.4 represents error packets, Sectin B.5 represents hello packets, and finally failure packets are represented in Sectin B.6.

Each of the following tables are presented in two parts for convenience. the first part has the results when the speed of nodes is 1, 5, and 10 m/sec, and the second part contains the results for speeds15, 20, and 25 m/sec. Each row contains the pause time, mean, Standard Deviation (StD), Confidence Interval (CoIn), and the Population Mean Range.

B.1 Total control packets

Pause Time	Speed of nodes														
	Speed 1					Speed 5					Speed 10				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	1273.90	26.06	9.33	1264.57	1283.23	2166.70	120.52	43.13	2123.57	2209.83	3124.23	159.40	57.04	3067.19	3181.28
50	1239.17	24.17	8.65	1230.52	1247.81	1780.43	97.57	34.91	1745.52	1815.35	2309.30	128.73	46.07	2263.23	2355.37
100	1219.07	17.86	6.39	1212.68	1225.46	1472.47	65.46	23.42	1449.04	1495.89	1892.83	88.48	31.66	1861.17	1924.50
150	1190.67	21.40	7.66	1183.01	1198.33	1248.07	24.50	8.77	1239.30	1256.84	1368.00	33.88	12.12	1355.88	1380.12
200	1177.10	18.97	6.79	1170.31	1183.89	1176.27	16.49	5.90	1170.36	1182.17	1178.60	24.45	8.75	1169.85	1187.35

Pause Time	Speed of nodes														
	Speed 15					Speed 20					Speed 25				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	3831.90	169.82	60.77	3771.13	3892.67	4230.67	211.40	75.65	4155.02	4306.32	4652.97	171.02	61.20	4591.77	4714.17
50	2619.33	168.10	60.15	2559.18	2679.49	2871.50	146.23	52.33	2819.17	2923.83	3098.53	150.41	53.82	3044.71	3152.36
100	2003.40	99.46	35.59	1967.81	2038.99	2054.47	114.69	41.04	2013.42	2095.51	2109.70	101.47	36.31	2073.39	2146.01
150	1492.43	48.95	17.52	1474.92	1509.95	1644.63	89.91	32.17	1612.46	1676.81	1737.93	97.25	34.80	1703.13	1772.73
200	1185.97	44.80	16.03	1169.93	1202.00	1183.17	20.82	7.45	1175.72	1190.62	1180.63	20.46	7.32	1173.31	1187.95

Table B.1: Total Control Packets generated by MDSDV vs Pause Time (related to Figure 6.1 in Chapter 6)

B.2 Full Dumps

Pause Time	Speed of nodes														
	Speed 1					**Speed 5**					**Speed 10**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	3.17	3.31	1.19	1.98	4.35	95.47	18.11	6.48	88.98	101.95	312.90	43.19	15.45	297.45	328.35
50	3.50	4.45	1.59	1.91	5.09	27.97	8.47	3.03	24.94	31.00	119.77	23.36	8.36	111.41	128.12
100	8.17	8.26	2.96	5.21	11.12	4.67	3.12	1.12	3.55	5.78	13.53	5.77	2.06	11.47	15.60
150	8.57	10.75	3.85	4.72	12.41	7.67	10.27	3.68	3.99	11.34	9.93	11.31	4.05	5.89	13.98
200	7.63	8.82	3.16	4.48	10.79	7.47	8.05	2.88	4.58	10.35	7.73	11.43	4.09	3.64	11.82

Pause Time	Speed of nodes														
	Speed 15					**Speed 20**					**Speed 25**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	497.10	59.71	21.37	475.73	518.47	583.23	45.81	16.39	566.84	599.63	701.10	51.84	18.55	682.55	719.65
50	181.10	38.59	13.81	167.29	194.91	218.43	29.86	10.69	207.75	229.12	284.47	42.40	15.17	269.30	299.64
100	15.17	4.78	1.71	13.46	16.88	15.33	6.21	2.22	13.11	17.55	16.13	5.57	1.99	14.14	18.13
150	9.37	6.71	2.40	6.97	11.77	10.50	8.38	3.00	7.50	13.50	8.67	6.19	2.21	6.45	10.88
200	11.10	21.06	7.54	3.56	18.64	9.77	9.58	3.43	6.34	13.19	9.17	8.89	3.18	5.99	12.35

Table B.2: Full Dumps vs Pause Time (related to Figure 6.2 in Chapter 6)

B.3 Update Packets

Pause Time	Speed of nodes														
	Speed 1					**Speed 5**					**Speed 10**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	1158.77	3.84	1.37	1157.39	1160.14	1158.13	3.26	1.17	1156.97	1159.30	1158.57	3.05	1.09	1157.48	1159.66
50	1158.20	2.63	0.94	1157.26	1159.14	1158.07	3.60	1.29	1156.78	1159.35	1157.00	2.97	1.06	1155.94	1158.06
100	1158.40	3.95	1.41	1156.99	1159.81	1157.17	3.34	1.20	1155.97	1158.36	1157.57	3.86	1.38	1156.19	1158.95
150	1157.47	3.38	1.21	1156.26	1158.68	1158.40	3.15	1.13	1157.27	1159.53	1156.57	3.43	1.23	1155.34	1157.79
200	1157.23	4.07	1.45	1155.78	1158.69	1157.47	3.45	1.24	1156.23	1158.70	1158.70	3.61	1.29	1157.41	1159.99

Pause Time	Speed of nodes														
	Speed 15					**Speed 20**					**Speed 25**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	1157.80	3.60	1.29	1156.51	1159.09	1158.00	3.70	1.32	1156.68	1159.32	1158.90	3.75	1.34	1157.56	1160.24
50	1157.50	3.31	1.18	1156.32	1158.68	1157.53	3.20	1.15	1156.39	1158.68	1158.10	3.61	1.29	1156.81	1159.39
100	1157.17	3.97	1.42	1155.74	1158.59	1158.33	2.83	1.01	1157.32	1159.35	1157.80	3.69	1.32	1156.48	1159.12
150	1158.00	2.44	0.87	1157.13	1158.87	1158.00	3.34	1.20	1156.80	1159.20	1158.47	2.49	0.89	1157.58	1159.36
200	1159.17	2.88	1.03	1158.14	1160.20	1158.63	3.26	1.17	1157.47	1159.80	1157.97	3.32	1.19	1156.78	1159.15

Table B.3: Update Packets vs Pause Time (related to Figure 6.3 in Chapter 6)

B.4 Error Packets

Pause Time	Speed of nodes														
	Speed 1					**Speed 5**					**Speed 10**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	108.23	26.37	9.44	98.80	117.67	908.80	117.35	41.99	866.81	950.79	1648.70	129.09	46.19	1602.51	1694.89
50	73.80	21.43	7.67	66.13	81.47	590.27	96.16	34.41	555.86	624.68	1027.67	116.54	41.70	985.96	1069.37
100	48.43	13.16	4.71	43.73	53.14	306.67	65.19	23.33	283.34	330.00	717.53	87.62	31.35	686.18	748.89
150	20.20	11.23	4.02	16.18	24.22	78.20	20.62	7.38	70.82	85.58	197.30	31.77	11.37	185.93	208.67
200	8.20	9.05	3.24	4.96	11.44	7.70	8.31	2.97	4.73	10.67	8.27	12.77	4.57	3.70	12.83

Pause Time	Speed of nodes														
	Speed 15					**Speed 20**					**Speed 25**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	2172.63	126.40	45.23	2127.40	2217.87	2483.57	178.67	63.94	2419.63	2547.50	2787.23	158.55	56.74	2730.50	2843.97
50	1276.60	143.21	51.25	1225.35	1327.85	1490.33	130.63	46.75	1443.59	1537.08	1649.90	125.20	44.80	1605.10	1694.70
100	826.43	101.99	36.50	789.93	862.93	876.40	116.41	41.66	834.74	918.06	931.47	100.86	36.09	895.38	967.56
150	320.47	48.23	17.26	303.21	337.73	470.60	93.98	33.63	436.97	504.23	566.23	95.77	34.27	531.96	600.51
200	11.60	21.89	7.83	3.77	19.43	10.67	10.59	3.79	6.88	14.46	9.50	9.22	3.30	6.20	12.80

Table B.4: Error Packets vs Pause Time (related to Figure 6.4 in Chapter 6)

B.5 Hello Messages

Pause Time	Speed of nodes														
	Speed 1					**Speed 5**					**Speed 10**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	3.60	1.00	0.36	3.24	3.96	3.77	1.07	0.38	3.38	4.15	3.37	0.93	0.33	3.03	3.70
50	3.47	0.97	0.35	3.12	3.81	3.47	0.82	0.29	3.17	3.76	3.73	1.05	0.38	3.36	4.11
100	3.63	1.03	0.37	3.26	4.00	3.80	0.81	0.29	3.51	4.09	3.50	0.97	0.35	3.15	3.85
150	3.67	0.99	0.36	3.31	4.02	3.53	1.20	0.43	3.11	3.96	3.83	1.12	0.40	3.43	4.23
200	3.63	1.13	0.40	3.23	4.04	3.43	0.86	0.31	3.13	3.74	3.50	1.01	0.36	3.14	3.86

Pause Time	Speed of nodes														
	Speed 15					**Speed 20**					**Speed 25**				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	3.67	0.84	0.30	3.36	3.97	3.27	0.91	0.32	2.94	3.59	3.30	0.88	0.31	2.99	3.61
50	3.40	0.97	0.35	3.05	3.75	3.70	0.88	0.31	3.39	4.01	3.67	0.96	0.34	3.32	4.01
100	4.00	1.02	0.36	3.64	4.36	3.47	1.07	0.38	3.08	3.85	3.40	0.93	0.33	3.07	3.73
150	3.77	1.07	0.38	3.38	4.15	3.97	1.03	0.37	3.60	4.34	3.63	0.96	0.35	3.29	3.98
200	3.77	1.01	0.36	3.41	4.13	3.53	0.97	0.35	3.19	3.88	3.50	1.20	0.43	3.07	3.93

Table B.5: Hello Messages vs Pause Time (related to Figure 6.5 in Chapter 6)

B.6 Failure Packets

Pause Time	Speed of nodes														
	Speed 1					Speed 5					Speed 10				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	0.13	0.51	0.18	-0.05	0.31	0.53	1.07	0.38	0.15	0.92	0.70	1.47	0.52	0.18	1.22
50	0.20	0.61	0.22	-0.02	0.42	0.67	1.73	0.62	0.05	1.29	1.13	1.53	0.55	0.59	1.68
100	0.43	1.04	0.37	0.06	0.81	0.17	0.65	0.23	-0.07	0.40	0.70	1.58	0.57	0.13	1.27
150	0.77	1.70	0.61	0.16	1.37	0.27	0.69	0.25	0.02	0.51	0.37	1.00	0.36	0.01	0.72
200	0.40	0.81	0.29	0.11	0.69	0.20	0.66	0.24	-0.04	0.44	0.40	0.97	0.35	0.05	0.75

Pause Time	Speed of nodes														
	Speed 15					Speed 20					Speed 25				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	0.70	1.53	0.55	0.15	1.25	2.60	3.42	1.22	1.38	3.82	2.43	2.81	1.01	1.43	3.44
50	0.73	1.48	0.53	0.20	1.26	1.50	2.08	0.74	0.76	2.24	2.40	3.21	1.15	1.25	3.55
100	0.63	1.19	0.43	0.21	1.06	0.93	1.96	0.70	0.23	1.64	0.90	1.56	0.56	0.34	1.46
150	0.83	1.23	0.44	0.39	1.27	1.57	3.10	1.11	0.46	2.68	0.93	2.12	0.76	0.18	1.69
200	0.33	1.15	0.41	-0.08	0.75	0.57	1.65	0.59	-0.03	1.16	0.50	1.74	0.62	-0.12	1.12

Table B.6: Failure Packets vs Pause Time (related to Figure 6.6 in Chapter 6)

Appendix C

MDSDV and DSDV comparison Data

This appendix presents the data that are used to produce all the figures in Chapter 7. The data included in the appendix represents the quantitative comparison of the DSDV and MDSDV routing protocols. Results are based on simulation of 30 runs, and the error bars represent the 95% confidence interval of the mean.

We conduct three experiments to produce the data. The network size is varied in the first experiment (Appendix C.1), the Pause time is varied in the second experiment (Appendix C.2), and the speed of nodes is varied in the third experiment (Appendix C.3).

Each row in the table contains the parameter (i.e., number of nodes, pause time, or speed of nodes), mean, Standard Deviation (StD), Confidence Interval (CoIn), and the Population Mean Range.

C.1 Network Size (Varying Number of Nodes)

Number of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	65.37	6.50	2.33	63.04	67.69	96.49	1.90	0.68	95.81	97.17
30	68.81	5.14	1.84	66.97	70.65	98.99	0.46	0.16	98.83	99.16
40	72.42	5.48	1.96	70.46	74.38	99.04	0.58	0.21	98.83	99.25
50	68.99	5.03	1.80	67.19	70.79	99.13	0.39	0.14	98.99	99.27
60	70.05	6.58	2.35	67.69	72.40	99.16	0.41	0.15	99.01	99.30
70	71.71	6.46	2.31	69.39	74.02	99.07	0.61	0.22	98.85	99.29
80	71.72	4.35	1.56	70.16	73.28	99.06	0.47	0.17	98.90	99.23
90	70.57	7.16	2.56	68.01	73.13	98.89	0.69	0.25	98.64	99.14
100	70.02	6.15	2.20	67.81	72.22	98.78	0.63	0.23	98.56	99.01

Table C.1: PDF vs Number of Nodes (related to Figure 7.1 in Chapter 7)

Number of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	88.54	14.38	5.15	83.39	93.68	90.29	14.86	5.32	84.95	95.58
30	97.00	1.99	0.71	96.29	97.71	99.66	0.80	0.29	99.37	99.94
40	97.35	1.67	0.60	96.75	97.94	99.91	0.15	0.05	99.86	99.97
50	97.58	1.44	0.52	97.07	98.10	99.92	0.16	0.06	99.86	99.98
60	97.48	1.60	0.57	96.91	98.05	99.95	0.06	0.02	99.92	99.97
70	98.30	1.23	0.44	97.86	98.74	99.96	0.06	0.02	99.94	99.98
80	97.77	1.39	0.50	97.28	98.27	99.96	0.07	0.03	99.93	99.98
90	97.87	1.68	0.60	97.27	98.47	99.95	0.05	0.02	99.93	99.97
100	97.53	1.32	0.47	97.06	98.00	99.95	0.06	0.02	99.93	99.97

Table C.2: PDF vs Number of Nodes (related to Figure 7.2 in Chapter 7)

Number of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	36.60	17.11	6.12	30.48	42.72	45.53	19.71	7.05	38.48	52.59
30	29.34	16.55	5.92	23.42	35.26	30.58	9.64	3.45	27.14	34.03
40	32.60	26.47	9.47	23.13	42.07	32.85	13.81	4.94	27.91	37.80
50	36.81	31.13	11.14	25.67	47.95	39.70	12.39	4.43	35.27	44.13
60	32.07	21.14	7.57	24.51	39.64	40.00	16.35	5.85	34.15	45.85
70	42.88	38.33	13.72	29.16	56.59	49.37	20.72	7.42	41.95	56.78
80	33.72	20.67	7.40	26.32	41.12	54.96	22.13	7.92	47.04	62.88
90	43.24	38.31	13.71	29.54	56.95	62.45	45.19	16.17	46.28	78.62
100	41.93	34.39	12.31	29.62	54.24	79.81	48.75	17.45	62.37	97.26

Table C.3: Average End to End Delay vs Number of Nodes (related to Figure 7.3) in Chapter 7)

Number of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	19.44	6.49	2.32	17.12	21.77	20.68	9.08	3.25	17.43	23.93
30	18.61	4.75	1.70	16.91	20.31	18.96	8.11	2.90	16.05	21.86
40	18.39	5.06	1.81	16.58	20.20	17.49	4.97	1.78	15.71	19.27
50	18.68	4.62	1.65	17.03	20.33	17.56	4.33	1.55	16.01	19.11
60	19.34	4.78	1.71	17.63	21.05	17.80	4.37	1.56	16.24	19.37
70	18.72	4.22	1.51	17.21	20.24	17.21	3.97	1.42	15.79	18.64
80	18.45	4.99	1.79	16.66	20.23	16.51	4.28	1.53	14.98	18.04
90	20.45	6.49	2.32	18.13	22.78	18.08	6.32	2.26	15.82	20.34
100	20.14	5.52	1.98	18.17	22.12	16.91	4.08	1.46	15.45	18.37

Table C.4: Average End to End Delay vs Number of Nodes (related to Figure 7.4 in Chapter 7)

Number of nodes	AODV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	0.1307	0.0161	0.0058	0.1249	0.1365	0.1850	0.0172	0.0061	0.1788	0.1911
30	0.2009	0.0215	0.0077	0.1932	0.2086	0.3171	0.0171	0.0061	0.3110	0.3232
40	0.2698	0.0307	0.0110	0.2588	0.2808	0.5046	0.0286	0.0102	0.4944	0.5149
50	0.3880	0.0372	0.0133	0.3747	0.4013	0.7066	0.0359	0.0129	0.6938	0.7195
60	0.4818	0.0574	0.0205	0.4612	0.5023	0.9595	0.0312	0.0111	0.9484	0.9707
70	0.5722	0.0695	0.0249	0.5473	0.5971	1.2607	0.0545	0.0195	1.2412	1.2802
80	0.6827	0.0526	0.0188	0.6639	0.7015	1.5887	0.0556	0.0199	1.5688	1.6086
90	0.8287	0.0966	0.0346	0.7942	0.8633	1.9534	0.0711	0.0254	1.9279	1.9788
100	0.9495	0.1152	0.0412	0.9083	0.9908	2.3520	0.0893	0.0320	2.3200	2.3839

Table C.5: NRL vs Number of Nodes (related to Figure 7.5 in Chapter 7)

Number of nodes	AODV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	0.0923	0.0169	0.0061	0.0862	0.0983	0.0916	0.0186	0.0067	0.0850	0.0983
30	0.1355	0.0052	0.0019	0.1336	0.1373	0.1183	0.0031	0.0011	0.1172	0.1194
40	0.1934	0.0093	0.0033	0.1901	0.1967	0.1565	0.0029	0.0010	0.1555	0.1576
50	0.2504	0.0106	0.0038	0.2466	0.2542	0.1957	0.0038	0.0014	0.1943	0.1970
60	0.3120	0.0148	0.0053	0.3067	0.3173	0.2344	0.0038	0.0014	0.2330	0.2357
70	0.3871	0.0218	0.0078	0.3793	0.3949	0.2736	0.0060	0.0021	0.2715	0.2758
80	0.4535	0.0313	0.0112	0.4423	0.4647	0.3114	0.0046	0.0017	0.3097	0.3130
90	0.5516	0.0309	0.0111	0.5406	0.5627	0.3547	0.0089	0.0032	0.3515	0.3579
100	0.6086	0.0392	0.0140	0.5946	0.6227	0.3894	0.0071	0.0026	0.3869	0.3920

Table C.6: NRL vs Number of Nodes (related to Figure 7.6 in Chapter 7)

Number of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	2089.20	390.80	139.85	1949.35	2229.05	199.70	109.08	39.04	160.66	238.74
30	1883.37	310.14	110.98	1772.39	1994.35	56.77	25.32	9.06	47.71	65.83
40	1665.40	330.37	118.22	1547.18	1783.62	55.57	33.85	12.11	43.45	67.68
50	1868.67	303.93	108.76	1759.91	1977.43	49.47	23.59	8.44	41.02	57.91
60	1804.97	394.00	140.99	1663.98	1945.96	48.83	24.23	8.67	40.16	57.50
70	1704.63	387.78	138.77	1565.87	1843.40	53.57	33.65	12.04	41.53	65.61
80	1703.70	261.74	93.66	1610.04	1797.36	52.13	24.79	8.87	43.26	61.01
90	1774.00	432.10	154.63	1619.37	1928.63	63.70	40.26	14.41	49.29	78.11
100	1806.80	370.40	132.55	1674.25	1939.35	71.47	35.69	12.77	58.70	84.24

Table C.7: Data Dropped vs Number of Nodes (related to Figure 7.7 in Chapter 7)

Number of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
20	687.00	860.47	307.92	379.08	994.92	570.33	892.43	319.35	250.98	889.69
30	180.73	119.84	42.88	137.85	223.62	17.63	43.43	15.54	2.09	33.18
40	159.57	100.30	35.89	123.67	195.46	2.93	3.55	1.27	1.66	4.20
50	145.37	87.08	31.16	114.21	176.53	3.50	9.05	3.24	0.26	6.74
60	151.17	96.31	34.47	116.70	185.63	1.97	2.68	0.96	1.01	2.93
70	102.10	74.10	26.52	75.58	128.62	1.67	3.34	1.19	0.47	2.86
80	133.53	83.63	29.93	103.61	163.46	1.83	3.51	1.26	0.58	3.09
90	127.80	101.49	36.32	91.48	164.12	2.00	2.55	0.91	1.09	2.91
100	148.23	79.13	28.32	119.92	176.55	2.03	2.91	1.04	0.99	3.07

Table C.8: Data Dropped vs Number of Nodes (related to Figure 7.8 in Chapter 7)

C.2 Varying Pause Time

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	94.99	2.81	1.00	93.99	96.00	99.96	0.04	0.02	99.95	99.98
50	95.73	2.22	0.79	94.94	96.52	99.96	0.04	0.01	99.95	99.98
100	96.42	1.98	0.71	95.71	97.13	99.95	0.05	0.02	99.93	99.97
150	96.62	1.98	0.71	95.92	97.33	99.91	0.13	0.05	99.87	99.96
200	97.70	1.25	0.45	97.25	98.14	99.95	0.05	0.02	99.94	99.97

Table C.9: PDF vs Pause Time (related to Figure 7.9 in Chapter 7)

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	68.99	5.03	1.80	67.19	70.79	99.13	0.39	0.14	98.99	99.27
50	73.17	6.09	2.18	70.99	75.35	99.21	0.39	0.14	99.08	99.35
100	85.63	4.34	1.55	84.08	87.19	99.47	0.42	0.15	99.32	99.62
150	87.68	3.50	1.25	86.43	88.94	99.33	0.49	0.17	99.16	99.50
200	97.58	1.44	0.52	97.07	98.10	99.92	0.16	0.06	99.86	99.98

Table C.10: PDF vs Pause Time (related to Figure 7.10 in Chapter 7)

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	15.11	3.55	1.27	13.84	16.38	14.20	2.77	0.99	13.21	15.19
50	15.84	3.85	1.38	14.46	17.22	15.19	3.92	1.40	13.79	16.60
100	18.39	4.45	1.59	16.80	19.99	16.45	3.71	1.33	15.12	17.77
150	18.76	4.86	1.74	17.02	20.50	17.74	4.85	1.74	16.00	19.47
200	18.32	4.44	1.59	16.73	19.91	17.15	3.86	1.38	15.77	18.53

Table C.11: Average End-to-End Delay vs Pause Time (related to Figure 7.11 in Chapter 7)

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	36.81	31.13	11.14	25.67	47.95	39.70	12.39	4.43	35.27	44.13
50	21.60	16.19	5.79	15.81	27.40	28.64	10.07	3.60	25.03	32.24
100	16.46	2.78	1.00	15.46	17.45	18.05	4.26	1.52	16.52	19.57
150	18.88	4.23	1.51	17.37	20.40	20.32	5.42	1.94	18.38	22.27
200	18.68	4.62	1.65	17.03	20.33	17.56	4.33	1.55	16.01	19.11

Table C.12: Average End-to-End Delay vs Pause Time (related to Figure 7.12 in Chapter 7)

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	0.2602	0.0134	0.0048	0.2554	0.2650	0.2107	0.0044	0.0016	0.2091	0.2123
50	0.2617	0.0131	0.0047	0.2570	0.2663	0.2052	0.0035	0.0013	0.2039	0.2064
100	0.2608	0.0135	0.0048	0.2559	0.2656	0.2017	0.0030	0.0011	0.2006	0.2027
150	0.2565	0.0159	0.0057	0.2508	0.2622	0.1970	0.0038	0.0014	0.1956	0.1983
200	0.2493	0.0093	0.0033	0.2459	0.2526	0.1948	0.0032	0.0011	0.1936	0.1959

Table C.13: NRL vs Pause Time (related to Figure 7.13 in Chapter 7)

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	0.3880	0.0372	0.0133	0.3747	0.4013	0.7066	0.0359	0.0129	0.6938	0.7195
50	0.4120	0.0438	0.0157	0.3964	0.4277	0.4790	0.0244	0.0087	0.4703	0.4878
100	0.3233	0.0274	0.0098	0.3135	0.3331	0.3405	0.0189	0.0068	0.3337	0.3472
150	0.3145	0.0236	0.0085	0.3061	0.3230	0.2727	0.0159	0.0057	0.2670	0.2783
200	0.2504	0.0106	0.0038	0.2466	0.2542	0.1957	0.0038	0.0014	0.1943	0.1970

Table C.14: NRL vs Pause Time (related to Figure 7.14 in Chapter 7)

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	301.87	169.50	60.66	241.21	362.52	1.43	2.03	0.73	0.71	2.16
50	257.30	133.78	47.87	209.43	305.17	1.50	2.08	0.74	0.76	2.24
100	215.60	119.64	42.81	172.79	258.41	1.77	1.99	0.71	1.05	2.48
150	203.33	119.51	42.77	160.57	246.10	3.23	5.24	1.87	1.36	5.11
200	138.80	74.99	26.83	111.97	165.63	1.47	1.76	0.63	0.84	2.10

Table C.15: Data Dropped vs Pause Time (related to Figure 7.15 in Chapter 7)

Pause Time	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
0	1868.67	303.93	108.76	1759.91	1977.43	49.47	23.59	8.44	41.02	57.91
50	1617.20	367.95	131.67	1485.53	1748.87	43.23	21.73	7.78	35.46	51.01
100	865.07	261.32	93.51	771.56	958.58	28.57	23.69	8.48	20.09	37.04
150	742.27	210.52	75.33	666.93	817.60	34.80	26.29	9.41	25.39	44.21
200	145.37	87.08	31.16	114.21	176.53	3.50	9.05	3.24	0.26	6.74

Table C.16: Data Dropped vs Pause Time (related to Figure 7.16 in Chapter 7)

C.3 Varying Speed of Nodes

Speed of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	94.99	2.81	1.00	93.99	96.00	99.96	0.04	0.02	99.95	99.98
5	85.05	4.68	1.67	83.38	86.72	99.81	0.15	0.05	99.76	99.87
10	76.36	5.68	2.03	74.33	78.40	99.50	0.34	0.12	99.38	99.63
15	73.52	6.73	2.41	71.11	75.92	99.36	0.27	0.10	99.26	99.46
20	68.99	5.03	1.80	67.19	70.79	99.13	0.39	0.14	98.99	99.27
25	66.41	5.92	2.12	64.29	68.53	98.88	0.45	0.16	98.72	99.04

Table C.17: PDF vs Speed of Nodes (related to Figure 7.17 in Chapter 7)

Speed of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	97.70	1.25	0.45	97.25	98.14	99.95	0.05	0.02	99.94	99.97
5	98.11	1.18	0.42	97.69	98.53	99.96	0.04	0.01	99.95	99.98
10	97.77	1.61	0.57	97.20	98.35	99.96	0.04	0.01	99.95	99.98
15	97.76	1.93	0.69	97.07	98.45	99.92	0.20	0.07	99.85	99.99
20	97.58	1.44	0.52	97.07	98.10	99.92	0.16	0.06	99.86	99.98
25	97.96	1.37	0.49	97.47	98.45	99.96	0.07	0.02	99.93	99.98

Table C.18: PDF vs Speed of Nodes (related to Figure 7.18 in Chapter 7)

Speed of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	15.11	3.55	1.27	13.84	16.38	14.20	2.77	0.99	13.21	15.19
5	18.27	9.11	3.26	15.01	21.53	16.64	11.33	4.06	12.58	20.70
10	29.49	21.04	7.53	21.96	37.02	23.33	12.01	4.30	19.04	27.63
15	29.77	23.12	8.27	21.50	38.04	29.72	12.10	4.33	25.39	34.05
20	36.81	31.13	11.14	25.67	47.95	39.70	12.39	4.43	35.27	44.13
25	51.36	29.36	10.51	40.86	61.87	52.76	14.55	5.21	47.55	57.97

Table C.19: Average End to End Delay vs Speed of Nodes (related to Figure 7.19 in Chapter 7)

Speed of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	18.32	4.44	1.59	16.73	19.91	17.15	3.86	1.38	15.77	18.53
5	17.61	3.62	1.30	16.32	18.91	16.45	3.25	1.16	15.28	17.61
10	17.36	4.46	1.60	15.77	18.96	16.43	4.18	1.49	14.94	17.93
15	18.63	7.19	2.57	16.06	21.20	16.85	6.29	2.25	14.60	19.10
20	18.68	4.62	1.65	17.03	20.33	17.56	4.33	1.55	16.01	19.11
25	17.83	4.20	1.50	16.33	19.33	16.65	3.73	1.33	15.31	17.98

Table C.20: Average End to End Delay vs Speed of Nodes (related to Figure 7.20 in Chapter 7)

Speed of nodes	AODV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	0.2602	0.0134	0.0048	0.2554	0.2650	0.2107	0.0044	0.0016	0.2091	0.2123
5	0.2973	0.0246	0.0088	0.2885	0.3061	0.3591	0.0200	0.0072	0.3519	0.3663
10	0.3419	0.0335	0.0120	0.3299	0.3539	0.5199	0.0276	0.0099	0.5100	0.5298
15	0.3527	0.0387	0.0139	0.3389	0.3666	0.6376	0.0289	0.0103	0.6273	0.6479
20	0.3880	0.0372	0.0133	0.3747	0.4013	0.7066	0.0359	0.0129	0.6938	0.7195
25	0.3934	0.0470	0.0168	0.3766	0.4102	0.7794	0.0284	0.0102	0.7693	0.7896

Table C.21: NRL vs Speed of Nodes (related to Figure 7.21 in Chapter 7)

Speed of nodes	AODV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	0.2493	0.0093	0.0033	0.2459	0.2526	0.1948	0.0032	0.0011	0.1936	0.1959
5	0.2479	0.0110	0.0039	0.2440	0.2519	0.1947	0.0028	0.0010	0.1937	0.1957
10	0.2507	0.0110	0.0039	0.2468	0.2547	0.1952	0.0042	0.0015	0.1937	0.1967
15	0.2487	0.0124	0.0044	0.2442	0.2531	0.1960	0.0073	0.0026	0.1934	0.1986
20	0.2504	0.0106	0.0038	0.2466	0.2542	0.1957	0.0038	0.0014	0.1943	0.1970
25	0.2484	0.0143	0.0051	0.2433	0.2536	0.1956	0.0037	0.0013	0.1942	0.1969

Table C.22: NRL vs Speed of Nodes (related to Figure 7.22 in Chapter 7)

Speed of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	301.87	169.50	60.66	241.21	362.52	1.43	2.03	0.73	0.71	2.16
5	901.87	282.69	101.16	800.71	1003.03	10.07	7.90	2.83	7.24	12.90
10	1425.03	342.16	122.44	1302.59	1547.48	28.80	19.61	7.02	21.78	35.82
15	1596.67	405.46	145.09	1451.58	1741.76	37.10	15.83	5.66	31.44	42.76
20	1868.67	303.93	108.76	1759.91	1977.43	49.47	23.59	8.44	41.02	57.91
25	2025.23	357.18	127.81	1897.42	2153.05	63.67	24.36	8.72	54.95	72.38

Table C.23: Data Dropped vs Speed of Nodes (related to Figure 7.23 in Chapter 7)

Speed of nodes	DSDV					MDSDV				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to
1	138.80	74.99	26.83	111.97	165.63	1.47	1.76	0.63	0.84	2.10
5	113.60	71.53	25.60	88.00	139.20	1.10	1.40	0.50	0.60	1.60
10	133.80	96.75	34.62	99.18	168.42	1.13	1.53	0.55	0.59	1.68
15	134.90	116.18	41.58	93.32	176.48	1.97	7.43	2.66	-0.69	4.62
20	145.37	87.08	31.16	114.21	176.53	3.50	9.05	3.24	0.26	6.74
25	122.27	82.34	29.47	92.80	151.73	1.77	3.08	1.10	0.66	2.87

Table C.24: Data Dropped vs Speed of Nodes (related to Figure 7.24 in Chapter 7)

Appendix D

MDSDV, AODV, and DSR comparison Data

This appendix presents the data that are used to produce all the figures in Chapter 8. The data included in the appendix represents the quantitative comparison of the AODV, MDSDV, and DSR routing protocols. Results are based on simulation of 30 runs, and the error bars represent the 95% confidence interval of the mean.

We conduct four experiments to produce the data. We varied the number of sources in the first experiment (Appendix D.1), the network size is varied in the second experiment (Appendix D.2), the pause time is varied in the third experiment (Appendix D.3), and the speed of nodes is varied in the fourth experiment (Appendix D.4).

Each row in the table contains the parameter (i.e., number of sources, number of nodes, pause time, or speed of nodes), mean, Standard Deviation (StD), Confidence Interval (CoIn), and the Population Mean Range.

D.1 Varying Number of Sources)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	98.54	0.89	0.32	98.22	98.85	97.87	1.43	0.51	97.36	98.38	97.88	1.63	0.58	97.30	98.46
20	98.31	0.85	0.30	98.00	98.61	97.72	1.38	0.49	97.23	98.21	98.17	1.51	0.54	97.63	98.71
30	95.16	3.47	1.24	93.91	96.40	96.21	2.27	0.81	95.40	97.03	94.81	4.15	1.49	93.33	96.30
40	92.63	4.12	1.48	91.16	94.11	94.52	3.13	1.12	93.41	95.65	92.14	4.97	1.78	90.36	93.91
50	90.71	4.33	1.55	89.17	92.26	93.02	3.66	1.31	91.71	94.33	89.64	5.71	2.04	87.60	91.69

Table D.1: PDF vs Number of Sources (related to Figure 8.1 in Chapter 8)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	99.64	0.89	0.32	99.32	99.96	99.94	0.07	0.03	99.92	99.97	99.96	0.04	0.01	99.94	99.97
20	98.55	2.36	0.84	97.71	99.39	99.90	0.10	0.04	99.86	99.93	99.85	0.32	0.12	99.73	99.96
30	86.42	7.71	2.76	83.66	89.18	96.71	3.38	1.21	95.50	97.93	90.20	9.04	3.24	86.97	93.44
40	81.76	6.60	2.36	79.39	84.12	93.69	4.13	1.48	92.22	95.17	85.36	8.17	2.92	82.44	88.29
50	79.75	7.29	2.61	77.14	82.36	91.23	4.22	1.51	89.72	92.74	82.42	7.84	2.81	79.62	85.23

Table D.2: PDF vs Number of Sources (related to Figure 8.2 in Chapter 8)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	65.42	67.97	24.32	41.10	89.74	41.01	28.96	10.36	30.65	51.37	27.64	14.70	5.26	22.38	32.90
20	60.32	35.87	12.83	47.49	73.16	47.12	28.61	10.24	36.88	57.36	56.54	51.94	18.59	37.96	75.13
30	119.44	67.92	24.31	95.13	143.75	111.46	80.22	28.71	82.75	140.16	266.83	204.54	73.19	193.64	340.02
40	174.61	108.99	39.00	135.61	213.61	176.07	113.75	40.70	135.37	216.78	396.60	333.98	119.51	277.09	516.11
50	221.78	109.91	39.33	182.45	261.11	218.94	128.87	46.12	172.83	265.06	511.07	312.32	111.76	399.31	622.83

Table D.3: Average End to End Delay vs Number of Sources (related to Figure 8.3 in Chapter 8)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	42.70	53.01	18.97	23.73	61.67	14.60	2.37	0.85	13.76	15.45	16.69	4.11	1.47	15.22	18.16
20	53.78	43.91	15.71	38.06	69.49	20.09	4.24	1.52	18.57	21.61	25.28	13.04	4.67	20.62	29.95
30	293.13	206.86	74.02	219.11	367.16	170.53	186.76	66.83	103.70	237.36	516.61	529.77	189.58	327.03	706.19
40	369.41	169.73	60.74	308.68	430.15	260.54	204.03	73.01	187.52	333.55	747.08	529.57	189.50	557.58	936.58
50	456.64	221.99	79.44	377.21	536.08	417.15	262.08	93.78	323.37	510.93	875.38	536.97	192.15	683.23	1067.54

Table D.4: Average End to End Delay vs Number of Sources (related to Figure 8.4 in Chapter 8)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	1.0846	0.2855	0.1022	0.9825	1.1868	0.8933	0.0635	0.0227	0.8706	0.9161	0.3026	0.0987	0.0353	0.2672	0.3379
20	1.1005	0.2328	0.0833	1.0171	1.1838	0.5952	0.0400	0.0143	0.5809	0.6096	0.3377	0.1082	0.0387	0.2990	0.3765
30	1.3796	0.4139	0.1481	1.2315	1.5278	0.4554	0.0369	0.0132	0.4422	0.4686	0.4296	0.1463	0.0524	0.3773	0.4820
40	1.5382	0.4234	0.1515	1.3867	1.6897	0.4482	0.0394	0.0141	0.4341	0.4623	0.4608	0.1439	0.0515	0.4093	0.5122
50	1.6130	0.3858	0.1381	1.4749	1.7511	0.4453	0.0420	0.0150	0.4302	0.4603	0.5041	0.1476	0.0528	0.4513	0.5569

Table D.5: NRL vs Number of Sources (related to Figure 8.5 in Chapter 8)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	0.2855	0.0932	0.0334	0.2522	0.3189	0.2684	0.0033	0.0012	0.2672	0.2696	0.1226	0.0322	0.0115	0.1111	0.1341
20	0.6037	0.4828	0.1728	0.4309	0.7764	0.1793	0.0045	0.0016	0.1777	0.1809	0.1257	0.0433	0.0155	0.1102	0.1412
30	2.2560	1.1730	0.4197	1.8363	2.6758	0.1757	0.0394	0.0141	0.1616	0.1898	0.3929	0.2622	0.0938	0.2990	0.4867
40	2.6409	1.0576	0.3785	2.2624	3.0193	0.1884	0.0394	0.0141	0.1744	0.2025	0.4837	0.2396	0.0858	0.3980	0.5695
50	2.7384	1.2549	0.4491	2.2893	3.1874	0.1990	0.0385	0.0138	0.1852	0.2128	0.5240	0.2221	0.0795	0.4445	0.6034

Table D.6: NRL vs Number of Sources (related to Figure 8.6 in Chapter 8)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	32.83	19.29	6.90	25.93	39.73	44.33	29.84	10.68	33.65	55.01	42.17	34.73	12.43	29.74	54.60
20	56.23	27.22	9.74	46.49	65.97	63.57	34.21	12.24	51.32	75.81	45.17	33.49	11.98	33.18	57.15
30	185.50	135.05	48.33	137.17	233.83	108.20	56.46	20.21	87.99	128.41	104.87	64.34	23.02	81.84	127.89
40	275.73	157.45	56.34	219.39	332.07	131.47	73.33	26.24	105.23	157.71	156.30	117.31	41.98	114.32	198.28
50	340.53	164.42	58.84	281.70	399.37	161.37	90.71	32.46	128.91	193.83	188.40	121.37	43.43	144.97	231.83

Table D.7: Data Dropped vs Number of Sources (related to Figure 8.7 in Chapter 8)

Number of sources	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
10	7.63	19.74	7.07	0.57	14.70	0.73	0.98	0.35	0.38	1.08	0.40	0.67	0.24	0.16	0.64
20	43.70	72.45	25.92	17.78	69.62	1.70	2.53	0.91	0.79	2.61	1.50	2.36	0.84	0.66	2.34
30	528.00	311.74	111.55	416.45	639.55	63.40	81.37	29.12	34.28	92.52	164.50	196.09	70.17	94.33	234.67
40	707.13	293.99	105.20	601.93	812.34	109.83	99.56	35.63	74.21	145.46	280.67	232.07	83.05	197.62	363.71
50	800.77	326.06	116.68	684.09	917.45	178.47	124.49	44.55	133.92	223.01	356.33	232.31	83.13	273.20	439.46

Table D.8: Data Dropped vs Number of Sources (related to Figure 8.8 in Chapter 8)

D.2 Network Size (Varying Number of Nodes)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	96.71	2.79	1.00	95.71	97.71	94.18	5.91	2.11	92.07	96.30	96.26	4.11	1.47	94.79	97.74
40	93.15	4.57	1.64	91.51	94.79	92.00	6.16	2.20	89.80	94.20	91.46	6.48	2.32	89.15	93.78
50	95.16	3.47	1.24	93.91	96.40	96.22	2.27	0.81	95.40	97.03	94.82	4.15	1.49	93.33	96.30
60	92.23	5.19	1.86	90.37	94.09	93.75	5.42	1.94	91.81	95.68	90.69	7.03	2.52	88.18	93.21
70	90.52	6.04	2.16	88.36	92.68	93.48	4.57	1.64	91.85	95.12	89.32	7.46	2.67	86.65	91.99
80	89.35	5.43	1.94	87.41	91.29	92.94	5.34	1.91	91.03	94.85	87.57	8.83	3.16	84.41	90.73
90	88.11	5.91	2.11	86.00	90.23	92.05	5.53	1.98	90.07	94.03	81.84	7.86	2.81	79.03	84.65
100	85.62	6.81	2.44	83.18	88.06	90.50	6.20	2.22	88.28	92.72	78.38	13.48	4.82	73.56	83.21

Table D.9: PDF vs Number of Nodes (related to Figure 8.9 in Chapter 8)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	92.18	7.95	2.84	89.34	95.02	94.46	6.73	2.41	92.05	96.87	92.54	9.58	3.43	89.11	95.97
40	86.96	8.71	3.12	83.85	90.08	93.33	5.69	2.04	91.29	95.37	87.71	10.18	3.64	84.07	91.35
50	86.42	7.71	2.76	83.66	89.18	96.72	3.38	1.21	95.50	97.93	90.20	9.04	3.24	86.97	93.44
60	82.51	10.62	3.80	78.71	86.31	95.10	5.45	1.95	93.15	97.05	87.44	11.12	3.98	83.46	91.42
70	79.52	9.71	3.48	76.04	82.99	95.52	3.97	1.42	94.10	96.94	86.38	9.46	3.39	83.00	89.77
80	76.26	10.38	3.72	72.54	79.97	95.98	3.58	1.28	94.70	97.26	83.44	10.14	3.63	79.81	87.07
90	78.62	11.15	3.99	74.63	82.61	96.96	2.74	0.98	95.98	97.94	85.08	11.89	4.26	80.83	89.34
100	71.09	12.98	4.64	66.45	75.73	94.54	4.25	1.52	93.02	96.06	78.33	12.58	4.50	73.83	82.83

Table D.10: PDF vs Number of Nodes (related to Figure 8.10 in Chapter 8)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	85.76	87.80	31.42	54.34	117.18	207.39	343.83	123.04	84.35	330.42	191.69	204.54	73.19	118.50	264.88
40	150.97	101.84	36.44	114.52	187.41	384.36	399.12	142.82	241.53	527.18	614.07	508.70	182.04	432.04	796.11
50	119.44	67.92	24.31	95.13	143.75	111.46	80.22	28.71	82.75	140.16	266.83	204.54	73.19	193.64	340.02
60	187.71	107.86	38.60	149.11	226.31	244.67	230.51	82.49	162.19	327.16	503.86	394.46	141.16	362.70	645.02
70	255.77	162.19	58.04	197.73	313.81	232.37	195.33	69.90	162.48	302.27	524.08	408.29	146.10	377.97	670.18
80	295.77	124.88	44.69	251.08	340.46	356.17	372.43	133.27	222.90	489.44	707.49	606.87	217.17	490.32	924.66
90	335.14	151.43	54.19	280.96	389.33	348.32	300.23	107.43	240.89	455.75	1048.20	604.38	216.28	831.92	1264.47
100	422.33	232.40	83.16	339.17	505.50	408.00	356.32	127.51	280.50	535.51	1133.77	748.99	268.02	865.75	1401.79

Table D.11: Average End to End Delay vs Number of Nodes (related to Figure 8.11 in Chapter 8)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	117.82	122.38	43.79	74.03	161.62	147.52	212.10	75.90	71.62	223.42	327.38	462.18	165.39	161.99	492.77
40	239.03	157.19	56.25	182.78	295.28	301.57	320.26	114.60	186.97	416.18	723.13	659.25	235.91	487.22	959.04
50	293.13	206.86	74.02	219.11	367.16	170.53	186.76	66.83	103.70	237.36	516.61	529.77	189.58	327.03	706.19
60	406.28	317.06	113.46	292.83	519.74	278.48	415.98	148.86	129.62	427.33	790.97	848.42	303.60	487.36	1094.57
70	550.78	356.13	127.44	423.34	678.22	252.64	244.93	87.65	164.99	340.29	798.77	617.21	220.87	577.90	1019.63
80	680.35	374.96	134.18	546.17	814.53	233.49	241.39	86.38	147.10	319.87	1024.06	813.81	291.22	732.84	1315.27
90	611.01	394.07	141.02	469.99	752.02	167.06	174.78	62.54	104.51	229.60	860.42	844.79	302.30	558.12	1162.73
100	807.55	470.87	168.50	639.05	976.05	330.28	297.97	106.63	223.65	436.91	1314.17	855.92	306.29	1007.88	1620.46

Table D.12: Average End to End Delay vs Number of Nodes (related to Figure 8.12 in Chapter 8)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	0.7367	0.2711	0.0970	0.6397	0.8338	0.2501	0.0350	0.0125	0.2376	0.2626	0.2754	0.1111	0.0398	0.2356	0.3152
40	1.2902	0.4727	0.1692	1.1210	1.4593	0.3563	0.0537	0.0192	0.3371	0.3755	0.4428	0.1631	0.0584	0.3844	0.5012
50	1.3796	0.4139	0.1481	1.2315	1.5278	0.4554	0.0369	0.0132	0.4422	0.4686	0.4296	0.1463	0.0524	0.3773	0.4820
60	2.0237	0.7883	0.2821	1.7416	2.3058	0.6200	0.0690	0.0247	0.5954	0.6447	0.6132	0.2370	0.0848	0.5284	0.6980
70	2.5243	0.9214	0.3297	2.1946	2.8540	0.8107	0.0705	0.0252	0.7855	0.8359	0.7764	0.3351	0.1199	0.6565	0.8964
80	3.1449	0.9168	0.3281	2.8168	3.4730	1.0027	0.1159	0.0415	0.9612	1.0442	0.9554	0.4462	0.1597	0.7958	1.1151
90	3.6770	1.1033	0.3948	3.2822	4.0719	1.2481	0.1351	0.0483	1.1997	1.2964	1.4684	0.5633	0.2016	1.2668	1.6699
100	4.6128	1.5070	0.5393	4.0736	5.1521	1.5242	0.1519	0.0543	1.4698	1.5785	2.0734	1.3759	0.4924	1.5810	2.5658

Table D.13: NRL vs Number of Nodes (related to Figure 8.13 in Chapter 8)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	0.8759	0.7808	0.2794	0.5964	1.1553	0.1294	0.0442	0.0158	0.1136	0.1452	0.2381	0.2168	0.0776	0.1605	0.3157
40	1.7182	1.0160	0.3636	1.3546	2.0818	0.1682	0.0506	0.0181	0.1501	0.1863	0.4005	0.2556	0.0915	0.3090	0.4920
50	2.2560	1.1730	0.4197	1.8363	2.6758	0.1757	0.0394	0.0141	0.1616	0.1898	0.3929	0.2622	0.0938	0.2990	0.4867
60	3.3359	1.8974	0.6790	2.6569	4.0149	0.2182	0.0588	0.0210	0.1971	0.2392	0.5510	0.3990	0.1428	0.4082	0.6938
70	4.1556	2.0144	0.7208	3.4348	4.8765	0.2430	0.0444	0.0159	0.2271	0.2589	0.7050	0.4045	0.1447	0.5602	0.8497
80	5.4082	2.7015	0.9667	4.4415	6.3749	0.2728	0.0467	0.0167	0.2561	0.2895	0.8971	0.5395	0.1931	0.7040	1.0901
90	5.1804	2.6575	0.9510	4.2295	6.1314	0.2891	0.0428	0.0153	0.2738	0.3044	0.9702	0.7244	0.2592	0.7110	1.2295
100	7.8822	4.2943	1.5367	6.3455	9.4189	0.3467	0.0558	0.0200	0.3267	0.3667	1.4956	0.9664	0.3458	1.1498	1.8415

Table D.14: NRL vs Number of Nodes (related to Figure 8.14 in Chapter 8)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	117.63	101.16	36.20	81.43	153.83	155.17	146.68	52.49	102.68	207.65	62.47	64.96	23.24	39.22	85.71
40	270.17	175.38	62.76	207.41	332.93	214.17	177.38	63.47	150.69	277.64	179.77	163.67	58.57	121.20	238.34
50	185.50	135.05	48.33	137.17	233.83	108.20	56.46	20.21	87.99	128.41	104.87	64.34	23.02	81.84	127.89
60	307.70	209.91	75.11	232.59	382.81	167.10	127.64	45.68	121.42	212.78	173.03	133.69	47.84	125.19	220.87
70	359.30	235.58	84.30	275.00	443.60	165.10	101.74	36.41	128.69	201.51	214.67	167.34	59.88	154.79	274.55
80	410.87	230.39	82.44	328.42	493.31	198.90	142.89	51.13	147.77	250.03	260.17	207.88	74.39	185.78	334.55
90	459.90	242.14	86.65	373.25	546.55	210.63	121.55	43.49	167.14	254.13	368.30	195.97	70.13	298.17	438.43
100	549.93	266.61	95.41	454.53	645.34	263.90	188.95	67.61	196.29	331.51	463.10	357.09	127.78	335.32	590.88

Table D.15: Data Dropped vs Number of Nodes (related to Figure 8.15 in Chapter 8)

Number of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
30	262.07	275.39	98.55	163.52	360.61	172.13	231.38	82.80	89.33	254.93	161.50	250.51	89.64	71.86	251.14
40	519.07	365.12	130.66	388.41	649.72	175.03	164.82	58.98	116.05	234.01	256.07	274.74	98.31	157.75	354.38
50	528.00	311.74	111.55	416.45	639.55	63.40	81.37	29.12	34.28	92.52	164.50	196.09	70.17	94.33	234.67
60	697.23	429.80	153.80	543.43	851.04	96.30	130.02	46.53	49.77	142.83	255.27	308.87	110.53	144.74	365.79
70	790.00	405.36	145.06	644.94	935.06	77.43	82.19	29.41	48.02	106.84	260.83	234.56	83.94	176.90	344.77
80	920.37	430.08	153.90	766.46	1074.27	72.10	87.15	31.19	40.91	103.29	356.63	277.17	99.18	257.45	455.82
90	819.80	454.33	162.58	657.22	982.38	48.23	49.59	17.75	30.49	65.98	304.43	335.30	119.99	184.45	424.42
100	1107.17	517.66	185.24	921.92	1292.41	103.33	102.87	36.81	66.52	140.14	432.00	305.29	109.25	322.75	541.25

Table D.16: Data Dropped vs Number of Nodes (related to Figure 8.16 in Chapter 8)

D.3 Varying Pause Time

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	92.96	6.95	2.49	90.47	95.45	98.60	2.00	0.71	97.88	99.31	95.96	5.89	2.11	93.86	98.07
25	87.95	7.92	2.83	85.12	90.79	96.87	4.02	1.44	95.43	98.31	92.41	8.25	2.95	89.46	95.36
50	86.96	6.86	2.46	84.51	89.42	96.52	2.95	1.06	95.47	97.58	90.36	8.08	2.89	87.46	93.25
75	83.94	10.46	3.74	80.19	87.68	94.38	4.66	1.67	92.71	96.05	85.87	10.00	3.58	82.30	89.45
100	89.74	5.92	2.12	87.62	91.85	97.17	3.04	1.09	96.08	98.26	93.22	5.96	2.13	91.09	95.35

Table D.17: PDF vs Pause Time (related to Figure 8.17 in Chapter 8)

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	95.16	3.47	1.24	93.91	96.40	96.22	2.27	0.81	95.40	97.03	94.82	4.15	1.49	93.33	96.30
25	91.42	5.98	2.14	89.28	93.56	93.89	5.69	2.04	91.86	95.93	89.99	7.92	2.83	87.15	92.82
50	93.79	3.96	1.42	92.37	95.20	96.73	2.34	0.84	95.89	97.57	94.98	4.40	1.58	93.40	96.55
75	86.97	6.27	2.24	84.73	89.22	92.14	5.38	1.93	90.21	94.06	88.48	8.73	3.12	85.36	91.61
100	86.42	7.71	2.76	83.66	89.18	96.72	3.38	1.21	95.50	97.93	90.20	9.04	3.24	86.97	93.44

Table D.18: PDF vs Pause Time (related to Figure 8.18 in Chapter 8)

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	201.75	198.65	71.09	130.67	272.84	80.25	98.95	35.41	44.84	115.66	259.85	377.97	135.26	124.59	395.10
25	264.70	203.35	72.77	191.93	337.47	174.90	268.62	96.12	78.77	271.02	427.17	500.98	179.27	247.89	606.44
50	265.60	144.48	51.70	213.90	317.30	190.84	157.44	56.34	134.50	247.18	649.37	664.03	237.62	411.75	886.99
75	399.72	316.35	113.20	286.51	512.92	300.21	301.48	107.88	192.33	408.10	839.24	636.79	227.87	611.37	1067.12
100	222.67	129.60	46.38	176.30	269.05	145.27	166.76	59.67	85.60	204.94	314.57	338.58	121.16	193.41	435.72

Table D.19: Average End to End Delay vs Pause Time (related to Figure 8.19 in Chapter 8)

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	119.44	67.92	24.31	95.13	143.75	111.46	80.22	28.71	82.75	140.16	266.83	204.54	73.19	193.64	340.02
25	187.55	114.35	40.92	146.63	228.46	172.01	156.09	55.86	116.16	227.87	478.51	475.86	170.28	308.23	648.79
50	156.75	95.49	34.17	122.57	190.92	103.89	79.38	28.40	75.48	132.29	268.20	239.46	85.69	182.51	353.89
75	318.66	196.53	70.33	248.33	388.98	250.03	220.50	78.90	171.12	328.93	642.18	657.88	235.42	406.76	877.60
100	293.13	206.86	74.02	219.11	367.16	170.53	186.76	66.83	103.70	237.36	516.61	529.77	189.58	327.03	706.19

Table D.20: Average End to End Delay vs Pause Time (related to Figure 8.20 in Chapter 8)

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	1.1783	0.9639	0.3449	0.8333	1.5232	0.1563	0.0273	0.0098	0.1465	0.1661	0.2236	0.1663	0.0595	0.1641	0.2831
25	1.8813	1.0594	0.3791	1.5022	2.2604	0.1719	0.0423	0.0152	0.1567	0.1871	0.3215	0.2476	0.0886	0.2329	0.4101
50	2.1613	1.0956	0.3921	1.7693	2.5534	0.1808	0.0332	0.0119	0.1690	0.1927	0.3841	0.2443	0.0874	0.2967	0.4715
75	2.5701	1.6038	0.5739	1.9961	3.1440	0.1896	0.0437	0.0156	0.1739	0.2052	0.5253	0.3086	0.1104	0.4149	0.6357
100	1.7146	0.8908	0.3188	1.3958	2.0333	0.1666	0.0296	0.0106	0.1560	0.1772	0.3144	0.1765	0.0632	0.2513	0.3776

Table D.21: NRL vs Pause Time (related to Figure 8.21 in Chapter 8)

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	1.3796	0.4139	0.1481	1.2315	1.5278	0.4554	0.0369	0.0132	0.4422	0.4686	0.4296	0.1463	0.0524	0.3773	0.4820
25	1.7202	0.6911	0.2473	1.4729	1.9675	0.3608	0.0580	0.0208	0.3400	0.3816	0.5100	0.2717	0.0972	0.4128	0.6072
50	1.3155	0.4696	0.1681	1.1475	1.4836	0.2405	0.0222	0.0080	0.2325	0.2485	0.3523	0.1144	0.0409	0.3114	0.3932
75	1.9307	0.7454	0.2667	1.6640	2.1975	0.1909	0.0360	0.0129	0.1780	0.2038	0.4774	0.2625	0.0939	0.3835	0.5713
100	2.2560	1.1730	0.4197	1.8363	2.6758	0.1757	0.0394	0.0141	0.1616	0.1898	0.3929	0.2622	0.0938	0.2990	0.4867

Table D.22: NRL vs Pause Time (related to Figure 8.22 in Chapter 8)

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	265.60	279.37	99.97	165.63	365.57	26.30	38.68	13.84	12.46	40.14	54.27	94.05	33.65	20.61	87.92
25	460.87	314.66	112.60	348.27	573.47	72.57	112.24	40.17	32.40	112.73	145.53	199.71	71.47	74.07	217.00
50	508.53	284.39	101.77	406.76	610.30	72.60	74.78	26.76	45.84	99.36	191.60	222.52	79.63	111.97	271.23
75	624.27	413.75	148.06	476.21	772.33	121.63	113.41	40.58	81.05	162.22	270.13	275.43	98.56	171.57	368.69
100	396.80	243.54	87.15	309.65	483.95	53.27	60.92	21.80	31.47	75.07	93.40	119.42	42.73	50.67	136.13

Table D.23: Data Dropped vs Pause Time (related to Figure 8.23 in Chapter 8)

Pause Time	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
0	185.50	135.05	48.33	137.17	233.83	108.20	56.46	20.21	87.99	128.41	104.87	64.34	23.02	81.84	127.89
25	330.70	232.32	83.13	247.57	413.83	147.80	182.80	65.42	82.38	213.22	180.50	219.09	78.40	102.10	258.90
50	235.67	147.67	52.84	182.82	288.51	81.73	52.55	18.81	62.93	100.54	75.77	64.73	23.16	52.60	98.93
75	509.67	265.78	95.11	414.56	604.78	177.47	136.85	48.97	128.50	226.44	227.50	254.66	91.13	136.37	318.63
100	528.00	311.74	111.55	416.45	639.55	63.40	81.37	29.12	34.28	92.52	164.50	196.09	70.17	94.33	234.67

Table D.24: Data Dropped vs Pause Time (related to Figure 8.24 in Chapter 8)

D.4 Varying Speed of Nodes

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	92.96	6.95	2.49	90.47	95.45	98.60	2.00	0.71	97.88	99.31	95.96	5.89	2.11	93.86	98.07
5	95.51	4.93	1.76	93.74	97.27	98.59	1.97	0.70	97.89	99.29	97.00	4.75	1.70	95.30	98.70
10	95.29	3.54	1.27	94.02	96.55	97.61	2.73	0.98	96.64	98.59	95.91	3.63	1.30	94.61	97.20
15	95.04	2.51	0.90	94.14	95.94	96.58	2.04	0.73	95.85	97.31	94.75	4.23	1.51	93.24	96.27
20	95.16	3.47	1.24	93.91	96.40	96.22	2.27	0.81	95.40	97.03	94.82	4.15	1.49	93.33	96.30

Table D.25: PDF vs Speed of Nodes (related to Figure 8.25 in Chapter 8)

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	89.74	5.92	2.12	87.62	91.85	97.17	3.04	1.09	96.08	98.26	93.22	5.96	2.13	91.09	95.35
5	86.05	7.55	2.70	83.35	88.75	95.67	4.39	1.57	94.10	97.24	89.46	8.51	3.04	86.42	92.51
10	85.72	9.24	3.31	82.41	89.02	96.04	4.44	1.59	94.45	97.63	90.77	9.43	3.37	87.40	94.15
15	86.37	7.86	2.81	83.56	89.18	95.36	3.74	1.34	94.02	96.70	89.30	8.01	2.86	86.44	92.17
20	86.42	7.71	2.76	83.66	89.18	96.72	3.38	1.21	95.50	97.93	90.20	9.04	3.24	86.97	93.44

Table D.26: PDF vs Speed of Nodes (related to Figure 8.26 in Chapter 8)

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	201.75	198.65	71.09	130.67	272.84	80.25	98.95	35.41	44.84	115.66	259.85	377.97	135.26	124.59	395.10
5	131.17	121.71	43.55	87.62	174.72	61.57	76.18	27.26	34.31	88.82	176.75	275.86	98.72	78.03	275.46
10	105.77	77.11	27.59	78.18	133.36	78.36	88.01	31.49	46.86	109.85	231.88	258.03	92.34	139.54	324.21
15	121.17	75.57	27.04	94.13	148.21	110.08	79.61	28.49	81.59	138.57	281.99	295.59	105.78	176.21	387.76
20	119.44	67.92	24.31	95.13	143.75	111.46	80.22	28.71	82.75	140.16	266.83	204.54	73.19	193.64	340.02

Table D.27: Average End to End Delay vs Speed of Nodes (related to Figure 8.27 in Chapter 8)

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	222.67	129.60	46.38	176.30	269.05	145.27	166.76	59.67	85.60	204.94	314.57	338.58	121.16	193.41	435.72
5	296.19	168.48	60.29	235.91	356.48	225.26	235.16	84.15	141.11	309.41	547.84	473.65	169.49	378.35	717.33
10	372.33	411.45	147.23	225.10	519.57	192.03	228.71	81.84	110.18	273.87	508.55	579.80	207.48	301.07	716.03
15	289.69	194.41	69.57	220.12	359.26	244.01	244.14	87.37	156.64	331.37	585.67	489.77	175.26	410.41	760.93
20	293.13	206.86	74.02	219.11	367.16	170.53	186.76	66.83	103.70	237.36	516.61	529.77	189.58	327.03	706.19

Table D.28: Average End to End Delay vs Speed of Nodes (related to Figure 8.28 in Chapter 8)

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	1.1783	0.9639	0.3449	0.8333	1.5232	0.1563	0.0273	0.0098	0.1465	0.1661	0.2236	0.1663	0.0595	0.1641	0.2831
5	0.9806	0.5410	0.1936	0.7870	1.1742	0.2126	0.0251	0.0090	0.2036	0.2216	0.2241	0.1447	0.0518	0.1723	0.2759
10	1.1022	0.3876	0.1387	0.9635	1.2409	0.3188	0.0299	0.0107	0.3081	0.3295	0.3135	0.1155	0.0413	0.2722	0.3548
15	1.3590	0.3513	0.1257	1.2333	1.4847	0.3955	0.0357	0.0128	0.3827	0.4082	0.3934	0.1246	0.0446	0.3488	0.4380
20	1.3796	0.4139	0.1481	1.2315	1.5278	0.4554	0.0369	0.0132	0.4422	0.4686	0.4296	0.1463	0.0524	0.3773	0.4820

Table D.29: NRL vs Speed of Nodes (related to Figure 8.29 in Chapter 8)

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	1.7146	0.8908	0.3188	1.3958	2.0333	0.1666	0.0296	0.0106	0.1560	0.1772	0.3144	0.1765	0.0632	0.2513	0.3776
5	2.1861	1.1302	0.4044	1.7816	2.5905	0.1814	0.0460	0.0165	0.1649	0.1978	0.4204	0.2859	0.1023	0.3181	0.5227
10	2.2637	1.4071	0.5035	1.7602	2.7672	0.1753	0.0400	0.0143	0.1610	0.1896	0.3509	0.2372	0.0849	0.2660	0.4358
15	2.1569	1.1473	0.4105	1.7463	2.5674	0.1834	0.0402	0.0144	0.1690	0.1978	0.4083	0.2278	0.0815	0.3267	0.4898
20	2.2560	1.1730	0.4197	1.8363	2.6758	0.1757	0.0394	0.0141	0.1616	0.1898	0.3929	0.2622	0.0938	0.2990	0.4867

Table D.30: NRL vs Speed of Nodes (related to Figure 8.30 in Chapter 8)

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	265.60	279.37	99.97	165.63	365.57	26.30	38.68	13.84	12.46	40.14	54.27	94.05	33.65	20.61	87.92
5	162.87	186.72	66.82	96.05	229.68	30.07	45.01	16.11	13.96	46.17	48.27	88.34	31.61	16.65	79.88
10	173.93	133.59	47.81	126.13	221.74	64.63	88.22	31.57	33.07	96.20	71.23	62.79	22.47	48.76	93.70
15	190.20	88.75	31.76	158.44	221.96	83.80	48.48	17.35	66.45	101.15	95.63	76.43	27.35	68.28	122.99
20	185.50	135.05	48.33	137.17	233.83	108.20	56.46	20.21	87.99	128.41	104.87	64.34	23.02	81.84	127.89

Table D.31: Data Dropped vs Speed of Nodes (related to Figure 8.31 in Chapter 8)

Speed of nodes	AODV					MDSDV					DSR				
	Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range		Mean	StD	CoIn	Population Mean Range	
				from	to				from	to				from	to
1	396.80	243.54	87.15	309.65	483.95	53.27	60.92	21.80	31.47	75.07	93.40	119.42	42.73	50.67	136.13
5	534.93	302.13	108.11	426.82	643.05	89.70	108.96	38.99	50.71	128.69	194.63	224.55	80.36	114.28	274.99
10	559.53	369.35	132.17	427.36	691.70	87.30	112.24	40.16	47.14	127.46	178.67	256.75	91.88	86.79	270.54
15	535.33	323.93	115.92	419.42	651.25	95.93	96.72	34.61	61.32	130.54	207.77	212.26	75.96	131.81	283.72
20	528.00	311.74	111.55	416.45	639.55	63.40	81.37	29.12	34.28	92.52	164.50	196.09	70.17	94.33	234.67

Table D.32: Data Dropped vs Speed of Nodes (related to Figure 8.32 in Chapter 8)

Appendix E

NS2 simulator modification

This appendix presents the NS2 simulator model code. The code is also available at: http://www.macs.hw.ac.uk/∼etorban/.

E.1 NS2 modified Files

File 1 packet.h

```
53  #define HDR_CMN(p)      (hdr_cmn::access(p))
       ..
       ..
61  #define HDR_IP(p)       (hdr_ip::access(p))
       ..
       ..
75  enum packet_t {
76    PT_TCP,
77    PT_UDP,
78    PT_CBR,
       ..
       ..
96    PT_HELLO,    //
97    PT_AVAILABLE,
98    PT_FULL_DUMP,   //
99    PT_UPDATE_PACKET,  //
100   PT_ERROR_PACKET,  //
101   PT_FAILURE,    //
       ..
       ..
123   /* CMU/Monarch's extnsions */
124   PT_ARP,
125   PT_MAC,
126   PT_TORA,
127   PT_DSR,
128   PT_AODV,
129   PT_MDSDV,
130   PT_IMEP,
       ..
       ..
185  };

186  class p_info {
187  public:
188      p_info() {
189        name_[PT_TCP]= "tcp";
190        name_[PT_UDP]= "udp";
191        name_[PT_CBR]= "cbr";
              ..
              ..
209        name_[PT_HELLO] = "HELLO";               // 20
210        name_[PT_AVAILABLE] = "AVAILABLE";       // 21
211        name_[PT_FULL_DUMP] = "FULL_DUMP";       // 22
212        name_[PT_UPDATE_PACKET] = "UPDATE_P";    // 23
213        name_[PT_ERROR_PACKET] = "ERROR_P";      // 24
214        name_[PT_FAILURE] = "FAILURE_P";         // 25
            ..
            ..
232        name_[PT_MAC]= "MAC";
233        name_[PT_TORA]= "TORA";
234        name_[PT_DSR]= "DSR";
235        name_[PT_AODV]= "AODV";
236        name_[PT_MDSDV]= "MDSDV";
237        name_[PT_IMEP]= "IMEP";
            ..
            ..
            ..
470   struct hdr_cmn {
```

```
471     enum dir_t { DOWN= -1, NONE= 0, UP= 1 };
472     packet_t ptype_;        // packet type
473     int      size_;         // simulated packet size
474     int      uid_;          // unique id
475     int      error_;        // error flag
476     int      errbitcnt_;    // # of corrupted bits jahn
477     int      fecsize_;
478     double   ts_;           // timestamp: for q-delay measurement
479     int      iface_;        // receiving interface (label)
480     dir_t    direction_;    // direction: 0=none, 1=up, -1=down
481     // source routing
482     char src_rt_valid;
483     double ts_arr_;         // Required by Marker of JOBS
484
485     //Monarch extn begins
486     nsaddr_t prev_hop_;     // IP addr of forwarding hop
487     nsaddr_t next_hop_; // next hop for this packet
488     int      addr_type_;    // type of next_hop_ addr
489     nsaddr_t last_hop_;     // for tracing on multi-user channels

490     //MDSDV extn begins
491     nsaddr_t current_node_  // Ip address of the node that is dealing with the packet.
492     nsaddr_t first_node_    // the neigbour that the source used to send the packet.
493     nsaddr_t forwarded_node_  // the second hop field of the chosen entry.
         ..
         ..
541 }
         ..
         ..
         ..
```

File 2 ns-lib.tcl

```
119  proc delay_parse { spec } {
120        return [time_parse $spec]
121  }

122  # Create the core OTcl class called "Simulator".
123  # This is the principal interface to the simulation engine.
124  #
125  #Class Simulator

126  #
127  # XXX Whenever you modify the source list below, please also change the
128  # OTcl script dependency list in Makefile.in
129  #
130  source ns-autoconf.tcl
131  source ns-address.tcl
         ..
         ..
190  source ../mobility/dsdv.tcl
191  source ../mobility/mdsdv.tcl
192  source ../mobility/dsr.tcl
         ..
         ..
597     # create node instance
598     set node [eval $self create-node-instance $args]

599     # basestation address setting
600     if { [info exist wiredRouting_] && $wiredRouting_ == "ON" } {
601             $node base-station [AddrParams addr2id [$node node-addr]]
602     }
603     switch -exact $routingAgent_ {
```

```
604         DSDV {
605                 set ragent [$self create-dsdv-agent $node]
606         }
607         MDSDV {
608                 set ragent [$self create-mdsdv-agent $node]
609         }
610         DSR {
611                 $self at 0.0 "$node start-dsr"
612         }
613         AODV {
614                 set ragent [$self create-aodv-agent $node]
615         }
616         TORA {
617                 Simulator set IMEPFlag_ ON
618                 set ragent [$self create-tora-agent $node]
619         }
          ..
          ..
801  Simulator instproc create-mdsdv-agent { node } {
802        # Create a mdsdv routing agent for this node
803        set ragent [new Agent/MDSDV]
804        # Setup address (supports hier-addr) for mdsdv agent
805        # and mobilenode
806        set addr [$node node-addr]
807        $ragent addr $addr
808        $ragent node $node
809        if [Simulator set mobile_ip_] {
810            $ragent port-dmux [$node demux]
811        }
812        $node addr $addr
813        $node set ragent_ $ragent
814        $self at 0.0 "$ragent start-mdsdv"    ;# start updates
815        return $ragent
}
          ..
          ..
          ..
```

File 3 mdsdv.tcl

```
38  # ======================================================================
39  # Default Script Options
40  # ======================================================================
41  Agent/MDSDV set sport_        0
42  Agent/MDSDV set dport_        0
43  Agent/MDSDV set wst0_         6        ;# As specified by Pravin
44  Agent/MDSDV set perup_       10        ;# update period
45  Agent/MDSDV set use_mac_      0        ;# Performance suffers with this on
46  Agent/MDSDV set be_random_    1        ;# Flavor the performance numbers
47  Agent/MDSDV set alpha_        0.875    ;# 7/8, as in RIP(?)
48  Agent/MDSDV set min_update_periods_ 3  ;# Missing perups before linkbreak
49  Agent/MDSDV set verbose_      0        ;#
50  Agent/MDSDV set trace_wst_    0        ;#
51
52
53  set opt(ragent)             Agent/MDSDV
          ..
          ..

65  Agent/MDSDV instproc init args {
66           eval $self next $args
```

```
67  }

73  proc create-MDSDV-routing-agent { node id } {
74      global ns_ ragent_ tracefd opt
75
76      #  Create the Routing Agent and attach it to port 255.
77      set ragent_($id) [new $opt(ragent)]
78      set ragent $ragent_($id)
79
80      ## setup address (supports hier-addr) for MDSDV agent and mobilenode.
81      set addr [$node node-addr]
          ..
          ..
          ..
114 }

117  proc mdsdv-create-mobile-node { id args } {
118     global ns ns_ chan prop topo tracefd opt node_
119     global chan prop tracefd topo opt
120
121     set ns_ [Simulator instance]
          ..
          ..
          ..
185     return $node
186  }
```

File 4 Makefile

```
159  OBJ_CC = \
160       mdsdv/mdsdv.o mdsdv/rtable02.o \
161       tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
162       common/scheduler.o common/object.o common/packet.o \
          ..
          ..
          ..
437  tcl/mobility/dsdv.tcl \
438  tcl/mobility/mdsdv.tcl \
439  tcl/mobility/dsr.tcl \
          ..
          ..
          ..
          ..
          ..
```

E.2 Header Files added

File 5 mdsdv.h

```
#ifndef cmu_dsdv_h_
#define cmu_dsdv_h_

#include "config.h"
```

```
#include "agent.h"
#include "ip.h"
#include "delay.h"
#include "scheduler.h"
#include "queue.h"
#include "trace.h"
#include "arp.h"
#include "ll.h"
#include "mac.h"
#include "priqueue.h"

#include "rtable.h"

#if defined(WIN32) && !defined(snprintf)
#define snprintf _snprintf
#endif /* WIN32 && !snprintf */

typedef double Time;

#define MAX_QUEUE_LENGTH 5
#define ROUTER_PORT      0xff

class MDSDV_Helper;

class MDSDV_Agent : public Agent {
  friend class MDSDV_Helper;
public:
  MDSDV_Agent();
  virtual int command(int argc, const char * const * argv);
  void lost_link(Packet *p);

protected:
  void helper_callback(Event *e);
  Packet* rtable(int);
  virtual void Recv_Packet(Packet *, Handler *);
  void trace(char* fmt, ...);
  void tracepkt(Packet *, double, int, const char *);

  Packet * Generate_Hello_Message(int new_seq);
  Packet * Make_Fulldump(nsaddr_t dst);
  Packet * Generate_Update_Packet(int change_count);
  Packet * Generate_Failure_Packet(nsaddr_t src, nsaddr_t dst,nsaddr_t first);

  // update old_rte in routing table to to new_rte
  int updateRoute(rtable_ent *old_rte, rtable_ent *new_rte);
  void Receive_Control_Packet (Packet * p);
  void Receive_Hello_Message (Packet * p);
  void Receive_Fulldump (Packet * p);
  void Receive_Updatepacket (Packet * p);
  void Receive_Error_packet (Packet * p);
  void Receive_Failure_Packet (Packet * p);
  void addNewNeighbour(nsaddr_t dst, int seq);
  void Forward_Packet (Packet * p);
  void startUp();
  int diff_subnet(int dst);
  void sendOutBCastPkt(Packet *p);
  void addNewEntry(nsaddr_t dst, int new_seq);

  Trace *tracetarget;         // Trace Target
  MDSDV_Helper  *helper_;   // MDSDV Helper, handles callbacks
  RoutingTable *table_;     // Routing Table
  NeighbourTable *ntable_;  // neighbours table
  PriQueue *ll_queue;         // link level output queue
  int seqno_;                 // Sequence number to advertise with...
  int myaddr_;                // My address...
  int linkno_;                // link id added by ali
```

```
  // Extensions for mixed type simulations using wired and wireless nodes
  char *subnet_;              // My subnet
  MobileNode *node_;          // My node
  // for debugging
  char *address;
  NsObject *port_dmux_;       // my port dmux

  Event *periodic_callback_;  // notify for periodic update
  Event *periodic_update_;

  // Randomness/MAC/logging parameters
  int be_random_;
  int use_mac_;
  int verbose_;
  int trace_wst_;

  // last time a periodic update was sent...
  double lasttup_;              // time of last triggered update
  double next_tup;              // time of next triggered update

  // MDSDV constants:
  double alpha_;  // 0.875
  double wst0_;   // 6 (secs)
  double perup_;  // 10 (secs)  period between updates
  int    min_update_periods_;    // 3 we must hear an update from a neighbor every
                                 // min_update_periods or we declare them unreachable

  void output_rte(const char *prefix, rtable_ent *prte, MDSDV_Agent *a);

};

class MDSDV_Helper : public Handler {
  public:
    MDSDV_Helper(MDSDV_Agent *a_) { a = a_; }
    virtual void handle(Event *e) { a->helper_callback(e); }

  private:
    MDSDV_Agent *a;
};

#endif
```

File 6 rtable.h

```
#ifndef cmu_rtable_h_
#define cmu_rtable_h_

#include "config.h"
#include "scheduler.h"
#include "queue.h"

#define BIG   250

#define NEW_ROUTE_SUCCESS_NEWENT       0
#define NEW_ROUTE_SUCCESS_OLDENT       1
#define NEW_ROUTE_METRIC_TOO_HIGH      2
#define NEW_ROUTE_ILLEGAL_CANCELLATION 3
#define NEW_ROUTE_INTERNAL_ERROR       4

#ifndef uint
typedef unsigned int uint;
```

```
#endif // !uint


class rtable_ent {
public:
  rtable_ent() { bzero(this, sizeof(rtable_ent));}
  nsaddr_t   dst;            // destination
  nsaddr_t   hop;            // next hop
  nsaddr_t   shop;           // second hop
  uint       metric;         // distance
  uint       seqnum;         // last sequence number we saw
  uint       link_no;        // link id
  double     changed_at;     // when route last changed
  double     new_seqnum_at;  // when we last heard a new seq number
  double     wst;            // running wst info
};


class RoutingTable {
  public:
    RoutingTable();

    int Add_Entry(const rtable_ent &ent,nsaddr_t *id);
    int Sort_Entry(const rtable_ent &ent,int first_entry_position, nsaddr_t nod_id);
    int RemainingLoop();
    void InitLoop();
    int number_of_elts();
    void local_rep(nsaddr_t nod_id);
    void stale_routes(nsaddr_t nod_id);

    rtable_ent *NextLoop();
    rtable_ent *GetEntry1(nsaddr_t dest, nsaddr_t via,nsaddr_t *myaddr );
    rtable_ent *GetEntry2(nsaddr_t dest, nsaddr_t via, nsaddr_t previous,nsaddr_t myaddr );

  private:
    rtable_ent *rtab;

    int    maxelts;
    int    elts;
    int    ctr;
};



//*****************  dealing with the neighbours table  ******************
class ntable_ent {
public:
  ntable_ent() { bzero(this, sizeof(ntable_ent));}
  nsaddr_t     dst;              // destination
  uint         neighbour;        // is it a neighbour or no (1 means a neighbour)
  uint         metric;           // distance

  uint         seqnum;           // last sequence number we saw
  uint         link_no;          // link id
  double       changed_at;       // when route last changed
  Event        *timeout_event;   // event used to schedule timeout action
  PacketQueue  *q;               //pkts queued for dst
  bool         active;           // is it an active
};


class NeighbourTable {
  public:
    NeighbourTable();

    // functions belongs to ntable..
    void nInitLoop();
```

```
    void nAddEntry(const ntable_ent &ent);
    int nRemainingLoop();
    ntable_ent *nNextLoop();
    ntable_ent *nGetEntry(nsaddr_t dest);


  private:
    ntable_ent *ntab;

    int    nmaxelts;
    int    nelts;
    int    nctr;

};



//*****************  dealing with the Queue table  ******************
class qtable_ent {
public:
  qtable_ent() { bzero(this, sizeof(qtable_ent));}
  qsaddr_t     dst;             // destination
  PacketQueue  *q;              //pkts queued for dst
};


class QueueTable {
  public:
    QueueTable();

    // functions belongs to qtable..
    void qAddEntry(const ntable_ent &ent);
    ntable_ent *qGetEntry(nsaddr_t dest);


  private:
    ntable_ent *qtab;

    int    nmaxelts;
    int    nelts;
    int    nctr;

};

#endif
```

E.3 Functions

Function 1 startUp

```
void MDSDV_Agent::startUp()
{
  // Each node starts by creating an entry belongs to itself in its routing table
  Time now = Scheduler::instance().clock();
  subnet_ = Address::instance().get_subnetaddr(myaddr_);

  address = Address::instance().print_nodeaddr(myaddr_);
```

```
  rtable_ent rte;
  bzero(&rte, sizeof(rte));

  rte.dst = myaddr_;
  rte.hop = myaddr_;
  rte.shop = -99;
  rte.metric = 0;
  rte.seqnum = seqno_;
  rte.link_no = myaddr_*10000+myaddr_;
  seqno_ += 2;

  rte.advertise_ok_at = 0.0;
  rte.advert_seqnum = true;
  rte.advert_metric = true;
  rte.changed_at = now;
  rte.new_seqnum_at = now;
  rte.wst = 0;
  rte.timeout_event = 0;

  rte.q = 0;

  nsaddr_t *my_ip = &myaddr_;
  table_->Add_Entry (rte, my_ip);

  periodic_callback_ = new Event ();
  Scheduler::instance ().schedule (helper_, periodic_callback_,
                                   jitter (DSDV_STARTUP_JITTER, be_random_));
}
```

Function 2 MDSDV_Agent

```
MDSDV_Agent::MDSDV_Agent (): Agent (PT_MESSAGE), ll_queue (0), seqno_ (0), linkno_ (0),
  myaddr_ (0), subnet_ (0), node_ (0), port_dmux_(0),
  periodic_callback_ (0), be_random_ (1),
  use_mac_ (0), verbose_ (1), trace_wst_ (0), lasttup_ (-10),
  alpha_ (0.875),  wst0_ (2), perup_ (10),
  periodic_update_ (0), min_update_periods_ (3) // constants
{
  table_ = new RoutingTable ();
  ntable_ = new NeighbourTable ();
  helper_ = new MDSDV_Helper (this);

  bind_time ("wst0_", &wst0_);
  bind_time ("perup_", &perup_);

  bind ("use_mac_", &use_mac_);
  bind ("be_random_", &be_random_);
  bind ("alpha_", &alpha_);
  bind ("min_update_periods_", &min_update_periods_);
  bind ("verbose_", &verbose_);
  bind ("trace_wst_", &trace_wst_);

  address = 0;
}
```

Function 3 Recv_Packet

```
void
MDSDV_Agent::Recv_Packet (Packet * p, Handler *)
{
  Scheduler & s = Scheduler::instance ();
  hdr_ip *iph = HDR_IP(p);
  hdr_cmn *cmh = HDR_CMN(p);
  int src = Address::instance().get_nodeaddr(iph->saddr());
  int dst = Address::instance().get_nodeaddr(iph->daddr());

  Time now = Scheduler::instance().clock();

  if(src == myaddr_ && cmh->num_forwards() == 0)
  {
    cmh->size() += IP_HDR_LEN;
    iph->ttl_ = IP_DEF_TTL;
  }

  else if(src == myaddr_)
  {
    drop(p, DROP_RTR_ROUTE_LOOP);
    return;
  }

  else
  {
    // If the Time To Live (TTL) is over, drop the packet.
    if(--iph->ttl_ == 0)
    {
      drop(p, DROP_RTR_TTL);
      return;
    }
  }

  if ((src != myaddr_) && (iph->dport() == ROUTER_PORT))
    Receive_Control_Packet(p);
  else if (iph->daddr() == ((int)IP_BROADCAST) && (iph->dport() != ROUTER_PORT))
  {
    if (src == myaddr_)
      sendOutBCastPkt(p);     // handle to broadcast the packet
    else
      port_dmux_->Recv_Packet(p, (Handler*)0);    // hand it over to the port-demux.
  }
  else
    Forward_Packet(p);     // forward the packet.
}
```

Function 4 Forward_Packet

```
void MDSDV_Agent::Forward_Packet (Packet * p)
{
  hdr_ip *iph = HDR_IP(p);
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();
  hdr_cmn *hdrc = HDR_CMN (p);

  rtable_ent *prte;          // pointer to an entry in Routing Table (NT)
  ntable_ent *nprte;         // pointer to an entry in Neighbours Table (NT)
```

```
hdrc->direction() = hdr_cmn::DOWN;
int src = Address::instance().get_nodeaddr(iph->saddr()); // get the source's address.
int dst = Address::instance().get_nodeaddr(iph->daddr()); // get the destination's address.

if (diff_subnet(iph->daddr()))
{
  // get the best route to the destination...
  prte = table_->GetEntry2 (dst,-99, -99,myaddr_);
  if (prte && prte->metric != BIG)
    goto send;          // a valid route is found, send the packet

  dst = node_->base_stn();
  prte = table_->GetEntry2 (dst,-99, -99,myaddr_);
  if (prte && prte->metric != BIG)
    goto send;
  else
  {
    //drop the packet with warning
    fprintf(stderr, "warning: Route to base_stn not known: dropping pkt\n");
    Packet::free(p);
    return;
  }
}

if (hdrc->num_forwards() == 0)    // This means the source needs to send a data...
{
  // Search for the best route to the destination. If a valid is route is found,
  // forward the packet, Otherwise queue it.
  prte = table_->GetEntry2 (dst,-99, hdrc->prev_hop_,myaddr_);
  if (prte && prte->metric != BIG)
    goto send;                  // a valid route is found, send the packet.
  else
  {
    // There is no route to the destination in the routing table. Queue the packet.
    // Find the entry in the QT that belongs to the destination.

    qtable_ent *qprte = qtable_->qGetEntry (dst);
    if (qprte)
    {
      // The entry is found. Queue the packet.
      if (!qprte->q)
        qprte->q = new PacketQueue ();

      qprte->q->enque(p);

      if (verbose_)
        trace ("VBP %.5f _%d_ %d:%d -> %d:%d", now, myaddr_, iph->saddr(), iph->sport(),
               iph->daddr(), iph->dport());

      while (qprte->q->length () > MAX_QUEUE_LENGTH)
        drop (qprte->q->deque (), DROP_RTR_QFULL);

      return;
    }
    else
    {
      // The entry is not found... create a new entry and queue the packet
      qtable_ent qrte;

      bzero(&qrte, sizeof(qrte));
      qrte.dst = dst;
      qrte.q = new PacketQueue();
      qrte.q->enque(p);

      assert (qrte.q->length() == 1 && 1 <= MAX_QUEUE_LENGTH);
      qtable_->qAddEntry(qrte);
```

```
      if (verbose_)
        trace ("VBP %.5f _%d_ %d:%d -> %d:%d", now, myaddr_, iph->saddr(), iph->sport(),
               iph->daddr(), iph->dport());

      return;
    }
  }
}
else  // This means an intermediate node plans to forward the packet....
{
  nsaddr_t  via = hdrc->forwarded_node_;
  if (hdrc->forwarded_node_ == -99)
    via = dst;

  // Search for a route to the destination through the node its address is specified
  // in hdrc->forwarded_node_ field...
  prte = table_->GetEntry1 (dst, via, &myaddr_);
  if (prte && prte->metric != BIG)
    goto send;                    // a valid route is found, send the packet
  else
  {
    // The intermediate node did not find the specified route, it should generate and
    // send a Failure packet to the source node, and try to find an alternative route.

    Generate_Failure_Packet(src, dst, hdrc->prev_hop_);

    prte = table_->GetEntry2 (dst,-99,hdrc->prev_hop_,myaddr_);
    if (prte && prte->metric != BIG)
      goto send;                // an alternative route is found, send the packet.
    else
    {
      // No alternative route is found. So, must queue the packet..

      qtable_ent *qprte = qtable_->qGetEntry (dst); // find entry belongs to the destination in NT.
      if (qprte)
      {
        // The entry is found. Queue the packet.
        if (!qprte->q)
          qprte->q = new PacketQueue ();

        qprte->q->enque(p);

        if (verbose_)
          trace ("VBP %.5f _%d_ %d:%d -> %d:%d", now, myaddr_, iph->saddr(), iph->sport(),
                 iph->daddr(), iph->dport());

        while (qprte->q->length () > MAX_QUEUE_LENGTH)
          drop (qprte->q->deque (), DROP_RTR_QFULL);

        return;
      }
      else
      {
      // The entry is not found... create a new entry and queue the packet
        qtable_ent qrte;

        bzero(&qrte, sizeof(qrte));
        qrte.dst = dst;
        qrte.q = new PacketQueue();
        qrte.q->enque(p);

        assert (qrte.q->length() == 1 && 1 <= MAX_QUEUE_LENGTH);
        qtable_->qAddEntry(qrte);

        if (verbose_)
          trace ("VBP %.5f _%d_ %d:%d -> %d:%d", now, myaddr_, iph->saddr(), iph->sport(),
```

```
                    iph->daddr(), iph->dport());

          return;
        }
      }
    }
}
send:

hdrc->addr_type_ = NS_AF_INET;
hdrc->xmit_failure_ = mac_callback;
hdrc->xmit_failure_data_ = this;

if (prte->metric > 1)
  hdrc->next_hop_ = prte->hop;
else
  hdrc->next_hop_ = dst;

if (hdrc->num_forwards() == 0)
  hdrc->first_node_ = hdrc->next_hop_;   // to inform the source regarding a link failure.

// hdrc->forwarded_node_ is used to force the receiver to forward the packet to this node.
// hdrc->current_node_ is used to ask the receiver not to forward the packet pack to me.
hdrc->forwarded_node_ = prte->shop;
hdrc->current_node_ = myaddr_;

if (verbose_)
{
  trace ("Routing pkts outside domain: \ VFP %.5f _%d_ %d:%d -> %d:%d", now, myaddr_,
   iph->saddr(), iph->sport(), iph->daddr(), iph->dport());
}

assert (!HDR_CMN (p)->xmit_failure_ || HDR_CMN (p)->xmit_failure_ == mac_callback);
target_->recv(p, (Handler *)0);

return;
}
```

Function 5 sendOutBCastPkt

```
void MDSDV_Agent::sendOutBCastPkt(Packet *p)
{
  Scheduler & s = Scheduler::instance ();
  // send out bcast pkt with jitter to avoid sync
  s.schedule (target_, p, jitter(DSDV_BROADCAST_JITTER, be_random_));

  s.cancel(periodic_callback_);    // cancel the next periodic callback
  s.schedule (helper_, periodic_callback_, 0);   // reschedule the next peridic update
}
```

Function 6 Receive_Control_Packet

```
void MDSDV_Agent::Receive_Control_Packet (Packet * p)
{
  int HELLO = 20;
  int FULL_DUMP = 22;
  int UPDATE_P = 23;
```

```
  int ERROR_P = 24;
  int FAILURE_P = 25;

  hdr_ip *iph = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN (p);

 unsigned char *d = p->accessdata ();  // A pointer to the begining of the received packet.
 unsigned char *w = d + 1;             // A pointer to the next byte of the received packet.

 // What kind of routing packet is received?
 if ( hdrc->ptype() == HELLO)         // is it a hello packet?
   Receive_Hello_Message(p);

 if (hdrc->ptype() == FULL_DUMP)      // is it a Full dump ?
   Receive_Fulldump(p);

 if (hdrc->ptype() == UPDATE_P)       // is it an update packet?
   Receive_Updatepacket(p);

 if (hdrc->ptype() == ERROR_P)        // is it an Errore packet?
   Receive_Error_packet(p);

 if (hdrc->ptype() == FAILURE_P)       // is it a Failure packet?
   Receive_Failure_Packet(p);
}
```

Function 7 Receive_Hello_Message

```
void MDSDV_Agent::Receive_Hello_Message (Packet * p)
{
  hdr_ip *iph = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN (p);

  int src = Address::instance().get_nodeaddr(iph->saddr());

  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  unsigned char *d = p->accessdata ();  // A pointer to the begining of the received packet.
  unsigned char *w = d + 1;             // A pointer to the next byte of the received packet.

  int change_count = *d;

  rtable_ent *prte;          // pointer to an entry in the Routing Table (RT)
  prte = NULL;

  ntable_ent *nprte;         // pointer to an entry in the Neighbours Table (NT)
  nprte = NULL;

  qtable_ent *qprte;         // pointer to an entry in the Queue Table (QT)
  qprte = NULL;

  // extracting the sequence number from the Update packet....
  int new_sequence = *(w++);
  new_sequence = new_sequence << 8 | *(w++);
  new_sequence = new_sequence << 8 | *(w++);
  new_sequence = new_sequence << 8 | *(w++);

  // Check if there is an entry belongs to the Hello Message sender in the NT.
  nprte = ntable_->nGetEntry (src);
  if (!nprte)
    addNewNeighbour(src, new_sequence);      // To add new entry in the NT.
```

```
else
{
  if (nprte->metric != BIG)
  {
    nprte->seqnum = new_sequence;
    Scheduler::instance ().cancel (nprte->timeout_event);
  }
  else
  {
    nprte->timeout_event = new Event ();
    nprte->metric = 1;
    nprte->changed_at = now;
    nprte->seqnum = new_sequence;
    nprte->link_no = myaddr_*10000+src;

    Packet *p = Make_Fulldump(src);
    if (p) {
      assert (!HDR_CMN (p)->xmit_failure_);
      s.schedule (target_, p, jitter(DSDV_BROADCAST_JITTER, be_random_));
    }
  }

  // reschedule the expected periodic time
  s.schedule (helper_, nprte->timeout_event, min_update_periods_ * perup_);

  if (verbose_)
  {
    trace ("VPC %.5f _%d_", now, myaddr_);
    tracepkt (p, now, myaddr_, "PU");
  }
}

// Check if there is a route to the Hello Message sender in the RT....
prte = table_->GetEntry1 (src, src,&myaddr_);
if (!prte)                          // the route is not found, create a route ...
  addNewEntry(src, new_sequence);
else
{
  if (prte->metric == BIG)
  {
    // an invalid route is found ... activate it
    prte->metric = 1;
    prte->seqnum = new_sequence;
    prte->changed_at = now;
  }
  else
  {
    // an active route is found ... update the sequence number and time
    prte->seqnum = new_sequence;
    prte->changed_at = now;
  }
}

// Check if there are queed packets in the QT belongs to the Hello Message sender.
qprte = qtable_->qGetEntry (src);
if (qprte && qprte->q)
{
  qtable_ent qrte;
  bzero(&qrte, sizeof(qrte));         //initialisaion of new entry

  int num_of_q_packets = 0;           //counter for the number of qued packets....

  Packet *queued_p;
  while ((queued_p = qprte->q->deque()) && num_of_q_packets < 7)
  {
    num_of_q_packets++;
    recv(queued_p, 0); // give the packets to ourselves to forward
```

```
    }

    delete qprte->q;

    qprte->q = 0;
    bcopy(qprte, &qrte,sizeof(qtable_ent));
  }
  lasttup_ = now;
  return;
}
```

Function 8 Receive_Fulldump

```
void MDSDV_Agent::Receive_Fulldump (Packet * p)
{
  hdr_ip *iph = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN (p);

  int src = Address::instance().get_nodeaddr(iph->saddr());

  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  int i;
  unsigned char *d = p->accessdata ();  // A pointer to the begining of the received packet.
  unsigned char *w = d + 1;             // A pointer to the next byte of the received packet.

  rtable_ent rte;
  rtable_ent *prte;      // pointer to an entry in the Routing Table (RT)

  nsaddr_t dst;

  prte = NULL;

  int kk = *d;
  int elements;
  rtable_ent *pr2;

  // extracting the sequence number from the Full Dump....
  int new_sequence = *(w++);
  new_sequence = new_sequence << 8 | *(w++);
  new_sequence = new_sequence << 8 | *(w++);
  new_sequence = new_sequence << 8 | *(w++);

  table_->stale_routes(myaddr_);   // calling a function to delete stale routes from the RT...

  int change_count = 0;  // To store number of entries to be included in the Update packet.
  int modify_rt = 0;     // To store number of overwritten or added entries in the RT...

  ntable_ent *nprte;      // pointer to an entry in the Neighbours Table (NT)
  nprte = NULL;

  qtable_ent *qprte;      // pointer to an entry in the Queue Table (QT)
  qprte = NULL;

  // Check if there is an entry belongs to the Full dump sender in the NT...
  nprte = ntable_->nGetEntry (src);
  if (!nprte)
    addNewNeighbour(src, new_sequence);     // add new entry in the NT.
  else
  {
    if (nprte->metric != BIG)
```

```
  {
    nprte->seqnum = new_sequence;
    Scheduler::instance ().cancel (nprte->timeout_event);
  }
  else
  {
    nprte->timeout_event = new Event ();
    nprte->metric = 1;
    nprte->changed_at = now;
    nprte->seqnum = new_sequence;
    nprte->link_no = myaddr_*10000+src;
  }
  // reschedule the expected periodic time
  s.schedule (helper_, nprte->timeout_event, min_update_periods_ * perup_);

  if (verbose_)
  {
    trace ("VPC %.5f _%d_", now, myaddr_);
    tracepkt (p, now, myaddr_, "PU");
  }
}

// Check if there is a direct route to the Full dump sender in the RT....
prte = table_->GetEntry1 (src, src,&myaddr_);
if (!prte)                          // the route is not found, create a route ...
  addNewEntry(src, new_sequence);
else if (prte->metric == BIG)
{
  // an invalid route is found, activate it.
  prte->metric = 1;
  prte->seqnum = new_sequence;
  prte->changed_at = now;
}
else
{
  // an active route is found, update it.
  prte->seqnum = new_sequence;
  prte->changed_at = now;
}


// Check if there are queed packets in the queue table belongs to the Full Dump sender.
qprte = qtable_->qGetEntry (src);
if (qprte && qprte->q)
{
  qtable_ent qrte;
  bzero(&qrte, sizeof(qrte));         //initialisaion of new entry

  int num_of_q_packets = 0;           //counter for the number of qued packets....

  Packet *queued_p;
  while ((queued_p = qprte->q->deque()) && num_of_q_packets < 7)
  {
    num_of_q_packets++;
    recv(queued_p, 0); // give the packets to ourselves to forward
  }

  delete qprte->q;

  qprte->q = 0;
  bcopy(qprte, &qrte,sizeof(qtable_ent));
}

// Dealing with the Full Dump entry by entry...
for (i = *d; i > 0; i--)
{
  bzero(&rte, sizeof(rte));      // initialisaion of new entry
```

```
// extracting data from the Full Dump....
dst = *(w++);                                   // destination
dst = dst << 8 | *(w++);
dst = dst << 8 | *(w++);
dst = dst << 8 | *(w++);

rte.dst = dst;

rte.hop = *(w++);                               // the first hop
rte.hop =rte.hop << 8 | *(w++);
rte.hop =rte.hop << 8 | *(w++);
rte.hop =rte.hop << 8 | *(w++);

rte.shop = *(w++);                              // the second hop
rte.shop =rte.shop << 8 | *(w++);
rte.shop =rte.shop << 8 | *(w++);
rte.shop =rte.shop << 8 | *(w++);

rte.metric = *(w++);                            // number of hops

rte.link_no = *(w++);                           // the link id
rte.link_no = rte.link_no << 8 | *(w++);
rte.link_no = rte.link_no << 8 | *(w++);
rte.link_no = rte.link_no << 8 | *(w++);

rte.seqnum = *(w++);                            // the sequence number
rte.seqnum = rte.seqnum << 8 | *(w++);
rte.seqnum = rte.seqnum << 8 | *(w++);
rte.seqnum = rte.seqnum << 8 | *(w++);

if(myaddr_== rte.dst || myaddr_== rte.hop || myaddr_== rte.shop)
  continue;

if (rte.metric == 0)
  continue;

rte.shop=rte.hop;
rte.hop = src;

if (rte.metric != BIG) rte.metric += 1;
  rte.changed_at = now;

// check if there is a route to this destination through Full dump sender...
prte = table_->GetEntry1 (rte.dst, src,&myaddr_);

/********** decide whether to update the routing table *********/
if (!prte)
{
  int write = updateRoute(NULL, &rte);
  if (write == 1)
    modify_rt++;  // Number of entries that have been overwritten or added to the RT.
}
else if (rte.metric <= prte->metric)      // the route is found ... choose the best
     {
       int write = updateRoute(prte, &rte);
       if (write == 1)
         modify_rt++;
     }
     else
       continue;


qtable_ent *qprte;         // pointer to an entry in the Queue table (QT)
qprte = NULL;

// Check if there are queed packets in the queue table belongs to the Destination.
```

```
    qprte = qtable_->qGetEntry (rte.dst);
    if (qprte && qprte->q)
    {
      qtable_ent qrte;
      bzero(&qrte, sizeof(qrte));         //initialisaion of new entry

      int num_of_q_packets = 0;           //counter for the number of qued packets....

      Packet *queued_p;
      while ((queued_p = qprte->q->deque()) && num_of_q_packets < 7)
      {
        num_of_q_packets++;
        recv(queued_p, 0); // give the packets to ourselves to forward
      }

      delete qprte->q;

      qprte->q = 0;
      bcopy(qprte, &qrte,sizeof(qtable_ent));
    }
  }
}
```

Function 9 Receive_Updatepacket

```
void MDSDV_Agent::Receive_Updatepacket (Packet * p)
{
  hdr_ip *iph = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN (p);

  int src = Address::instance().get_nodeaddr(iph->saddr());
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  int i;
  unsigned char *d = p->accessdata ();  // A pointer to the begining of the received packet.
  unsigned char *w = d + 1;             // A pointer to the next byte of the received packet.


  rtable_ent rte;
  rtable_ent *prte;               // pointer to an entry in the Routing Table (RT)
  prte = NULL;

  nsaddr_t dst;


  int kk = *d;
  int elements;
  rtable_ent *pr2;

  int new_sequence = *(w++);
  new_sequence = new_sequence << 8 | *(w++);
  new_sequence = new_sequence << 8 | *(w++);
  new_sequence = new_sequence << 8 | *(w++);

  table_->stale_routes(myaddr_);   // call a function to delete the stale routes....

  int change_count = 0;       // count of entries to be included in the Update packet.
  int modify_rt = 0;          // number of entries that have been overwritten or added in the RT

  ntable_ent *nprte;   // pointer to an entry in neighbours table (NT)
  nprte = NULL;
```

```
qtable_ent *qprte;      // pointer to an entry in the Queue Table (QT)
qprte = NULL;

// Check if there is an entry belongs to the Update packet sender in the NT...
nprte = ntable_->nGetEntry (src);
if (!nprte)
  addNewNeighbour(src, new_sequence);
else
{
  if (nprte->metric != BIG)
  {
    nprte->seqnum = new_sequence;
    Scheduler::instance ().cancel (nprte->timeout_event);
  }
  else
  {
    nprte->timeout_event = new Event ();
    nprte->metric = 1;
    nprte->changed_at = now;
    nprte->seqnum = new_sequence;
    nprte->link_no = myaddr_*10000+src;

    // call a function to generate a full dump and unicast it to the Update packet sender...
    Packet *p = Make_Fulldump(src);
    if (p)
    {
      assert (!HDR_CMN (p)->xmit_failure_);
      s.schedule (target_, p, jitter(DSDV_BROADCAST_JITTER, be_random_));
    }
  }

  // reschedule the expected periodic time
  s.schedule (helper_, nprte->timeout_event, min_update_periods_ * perup_);

  if (verbose_)
  {
    trace ("VPC %.5f _%d_", now, myaddr_);
    tracepkt (p, now, myaddr_, "PU");
  }
}

// check if you have a dirct route to the update packet sender.
prte = table_->GetEntry1 (src, src,&myaddr_);
if (!prte)                 // the route is not found, create a route ...
  addNewEntry(src, new_sequence);
else if (prte->metric ==BIG)
  {
    // an invalid route is found, activate it.
    prte->metric = 1;
    prte->seqnum = new_sequence;
    prte->changed_at = now;
  }
  else
  {
    // an active route is found, update it.
    prte->seqnum = new_sequence;
    prte->changed_at = now;
  }


  // Check if there are queed packets in the queue table belongs to the Full Dump sender.
  qprte = qtable_->qGetEntry (src);
  if (qprte && qprte->q)
  {
    qtable_ent qrte;
    bzero(&qrte, sizeof(qrte));         //initialisaion of new entry
```

```
  int num_of_q_packets = 0;           //counter for the number of qued packets....

  Packet *queued_p;
  while ((queued_p = qprte->q->deque()) && num_of_q_packets < 7)
  {
    num_of_q_packets++;
    recv(queued_p, 0); // give the packets to ourselves to forward
  }

  delete qprte->q;

  qprte->q = 0;
  bcopy(qprte, &qrte,sizeof(qtable_ent));
}

// Dealing with the received Update packet entry by entry...
for (i = *d; i > 0; i--)
{
  bzero(&rte, sizeof(rte));     //initialisaion of new entry

  // extracting data from the update packet....
  dst = *(w++);                              // destination
  dst = dst << 8 | *(w++);
  dst = dst << 8 | *(w++);
  dst = dst << 8 | *(w++);

  rte.dst = dst;

  rte.hop = *(w++);                          // the first hop
  rte.hop =rte.hop << 8 | *(w++);
  rte.hop =rte.hop << 8 | *(w++);
  rte.hop =rte.hop << 8 | *(w++);

  rte.shop = *(w++);                         // the second hop
  rte.shop =rte.shop << 8 | *(w++);
  rte.shop =rte.shop << 8 | *(w++);
  rte.shop =rte.shop << 8 | *(w++);

  rte.metric = *(w++);                       // number of hops

  rte.link_no = *(w++);                      // the link id
  rte.link_no = rte.link_no << 8 | *(w++);
  rte.link_no = rte.link_no << 8 | *(w++);
  rte.link_no = rte.link_no << 8 | *(w++);

  rte.seqnum = *(w++);                       // the sequence number
  rte.seqnum = rte.seqnum << 8 | *(w++);
  rte.seqnum = rte.seqnum << 8 | *(w++);
  rte.seqnum = rte.seqnum << 8 | *(w++);

  if(myaddr_== rte.dst || myaddr_== rte.hop || myaddr_== rte.shop || rte.metric == 0)
    continue;    // Ignore the entry..

  rte.shop=rte.hop;
  rte.hop = src;
  if (rte.metric != BIG) rte.metric += 1;
    rte.changed_at = now;


   // check if there is a route to the destination through The Update packet sender...
   prte = table_->GetEntry1 (rte.dst, src,&myaddr_);

   /*********** decide whether to update our routing table *********/
   if (!prte)
   {
     int write = updateRoute(NULL, &rte);
```

```
        if (write == 1)
          modify_rt++;      // Number of entries that have been overwritten or added to the RT.
       }
       else if (rte.metric <= prte->metric)   // the route is found ... choose the best
          {
            int write = updateRoute(prte, &rte);
            if (write == 1)
              modify_rt++;
          }
          else
            continue;


      qtable_ent *qprte;         // pointer to an entry in the Queue table (QT)
      qprte = NULL;

      // Check if there are queed packets in the queue table belongs to the Destination.
      qprte = qtable_->qGetEntry (rte.dst);
      if (qprte && qprte->q)
      {
        qtable_ent qrte;
        bzero(&qrte, sizeof(qrte));         //initialisaion of new entry

        int num_of_q_packets = 0;           //counter for the number of qued packets....

        Packet *queued_p;
        while ((queued_p = qprte->q->deque()) && num_of_q_packets < 7)
        {
          num_of_q_packets++;
          recv(queued_p, 0); // give the packets to ourselves to forward
        }

        delete qprte->q;

        qprte->q = 0;
        bcopy(qprte, &qrte,sizeof(qtable_ent));
      }
    }
}
```

Function 10 Receive_Error_packet

```
void MDSDV_Agent::Receive_Error_packet (Packet * p)
{
  hdr_ip *iph = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN (p);

  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  unsigned char *d = p->accessdata ();  // A pointer to the begining of the received packet.
  unsigned char *w = d + 1;             // A pointer to the next byte of the received packet.

  // extracting data from the Error packet....
  int flag = 0;

  int packet_sender = *(w++);
  packet_sender = packet_sender << 8 | *(w++);
  packet_sender = packet_sender << 8 | *(w++);
  packet_sender = packet_sender << 8 | *(w++);

  int destination = *(w++);
```

```
destination = destination << 8 | *(w++);
destination = destination << 8 | *(w++);
destination = destination << 8 | *(w++);

int link_no = *(w++);
link_no = link_no << 8 | *(w++);
link_no = link_no << 8 | *(w++);
link_no = link_no << 8 | *(w++);

int new_sequence = *(w++);
new_sequence = new_sequence << 8 | *(w++);
new_sequence = new_sequence << 8 | *(w++);
new_sequence = new_sequence << 8 | *(w++);

// To check if I am the Errror packet sender....
if (myaddr_ ==  packet_sender)
{
  Packet::free(p); // do nothing just free the packet...
  return;
}

rtable_ent rte;          // new rte learned from the Update packet being processed
rtable_ent *prte;        // pointer to an entry in the Routing table (RT)

rtable_ent *pr2;

ntable_ent *nprte;       // pointer to an entry in the Neighbours Table (NT)
nprte = NULL;

qtable_ent *qprte;       // pointer to an entry in the Queue table (QT)
qprte = NULL;


// Check if there is an entry belongs to the Error packet sender in the NT...
nprte = ntable_->nGetEntry (hdrc->prev_hop_);
if (!nprte)
  addNewNeighbour(hdrc->prev_hop_, new_sequence);   // insert a new neighbour in the NT
else
{
  if (nprte->metric != BIG)
  {
    // the Error packet is received from an old neighbour.. updating is needed.
    nprte->seqnum = new_sequence;
    Scheduler::instance ().cancel (nprte->timeout_event);
  }
  else
  {
    // the neighbour is not active.. activate it
    nprte->timeout_event = new Event ();
    nprte->metric = 1;
    nprte->changed_at = now;
    nprte->seqnum = new_sequence;
    nprte->link_no = myaddr_*10000+hdrc->prev_hop_;
  }

  // reschedule the expected periodic time
  s.schedule (helper_, nprte->timeout_event, min_update_periods_ * perup_);

  if (verbose_)
  {
    trace ("VPC %.5f _%d_", now, myaddr_);
    tracepkt (p, now, myaddr_, "PU");
  }
}

prte = NULL;
```

```
// Check if there is a direct route to the Error packet sender in the RT....
prte = table_->GetEntry1 (hdrc->prev_hop_, hdrc->prev_hop_,&myaddr_);
if (!prte)                      // the route is not found, create a route ...
  addNewEntry(hdrc->prev_hop_, new_sequence);
else if (prte->metric == BIG)
     {
       // an invalid route is found, activate it.
       prte->metric = 1;
       prte->seqnum = new_sequence;
       prte->changed_at = now;
     }
     else
     {
       // an active route is found, update it.
       prte->seqnum = new_sequence;
       prte->changed_at = now;
     }

     // Check if there are queed packets in the queue table belongs to the Error packet sender.
     qprte = qtable_->qGetEntry (src);
     if (qprte && qprte->q)
     {
       qtable_ent qrte;
       bzero(&qrte, sizeof(qrte));         //initialisaion of new entry

       int num_of_q_packets = 0;           //counter for the number of qued packets....

       Packet *queued_p;
       while ((queued_p = qprte->q->deque()) && num_of_q_packets < 7)
       {
         num_of_q_packets++;
         recv(queued_p, 0); // give the packets to ourselves to forward
       }

       delete qprte->q;

       qprte->q = 0;
       bcopy(qprte, &qrte,sizeof(qtable_ent));
     }

     prte = NULL;

     if (verbose_)
       trace ("VTO %.5f _%d_ %d->%d", now, myaddr_, myaddr_, prte->dst);

     if (myaddr_ ==  destination && packet_sender==hdrc->prev_hop_)
       return;

     int Flags=0;

     // Check that I am neither the destination nor the sender in the Error packet....
     if ((myaddr_ !=  destination) && (myaddr_ !=  packet_sender))
     {
       int link_no1 = packet_sender * 10000 + destination;
       int link_no2 = destination * 10000 + packet_sender;
       int bad2 = 0;     // counter of the bad routes

       // Check the Routing table and delete any entry that contains the same link number included
       for (table_->InitLoop (); (pr2 = table_->NextLoop ()); )
       {
         if ((pr2->link_no  == link_no1 || pr2->link_no  == link_no2) && pr2->metric != BIG)
         {
           Flags=1;
           if (verbose_)
             trace ("VTO %.5f _%d_ marking %d", now, myaddr_, pr2->dst);

             pr2->metric = BIG;     // assign this route as an invalid route
```

```
                pr2->changed_at = now;
                pr2->seqnum = 0;
                bad2++;
              }
            }

            table_->local_rep(myaddr_);
          }

return;

}
```

Function 11 Receive_Failure_Packet

```
void MDSDV_Agent::Receive_Failure_Packet (Packet * p)
{
  hdr_ip *iph = HDR_IP(p);
  hdr_cmn *hdrc = HDR_CMN (p);

  int src = Address::instance().get_nodeaddr(iph->saddr());
  int dst = Address::instance().get_nodeaddr(iph->daddr());

  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  unsigned char *d = p->accessdata ();  // A pointer to the begining of the received packet.
  unsigned char *w = d + 1;             // A pointer to the next byte of the received packet.


  rtable_ent *prte;       // pointer to an entry in the Rrouting Table (RT)

  // extracting data from the Failure packet....
  int source = *(w++);          // the source node address in the Failure packet...
  source = source << 8 | *(w++);
  source = source << 8 | *(w++);
  source = source << 8 | *(w++);

  int destination = *(w++);    // the destination node address in the Failure packet...
  destination = destination << 8 | *(w++);
  destination = destination << 8 | *(w++);
  destination = destination << 8 | *(w++);

  int first_hop = *(w++);  // the neighbour node that the source node used to send the data packet.
  first_hop = first_hop << 8 | *(w++);
  first_hop = first_hop << 8 | *(w++);
  first_hop = first_hop << 8 | *(w++);

  if (myaddr_ == source)
  {
    // this means that I am the source of the data packet....
    if (verbose_)
      trace ("VTO %.5f _%d_ %d->%d", now, myaddr_, myaddr_, prte->dst);

    // Check If there is a route to the destination through the first hop mentioned in
    // the failure Packet...
    prte = NULL;
    prte = table_->GetEntry1 (destination, first_hop, &myaddr_);

    if (prte)
    {
      // the route is found, and should be assigned as an invalid route.
```

```
      prte->metric = BIG;
      prte->changed_at = now;
      prte->seqnum = 0;

      table_->local_rep(myaddr_);     // call this function to delete stale routes...
    }
  }
  else
  {
    // I am not the source. The Failure packet should be forwarded to the source node...
    iph->saddr() = myaddr_;
    iph->daddr() = source;
    hdrc->addr_type() = NS_AF_INET;;
    hdrc->direction() = hdr_cmn::DOWN;       //important: change the packet's direction

    // forward the Failure Packet, jitter to avoid sync...
    Scheduler::instance ().cancel (p);
    s.schedule (target_, p, jitter(DSDV_BROADCAST_JITTER, be_random_));
  }
}
```

Function 12 Stale_Routes

```
void RoutingTable::Stale_Routes(nsaddr_t nod_id)
{
  //Function to aasigin some routes as invalid...
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  int kk = 0;
  rtable_ent *krte = NULL;
  rtable_ent ent;

  for (int max=0;max<elts;max++)
  {
    if (rtab[max].metric != 250 && rtab[max].metric > 1 && (now - rtab[max].changed_at > 25.0))
      rtab[max].metric = BIG;     // assign this route as an invalid route
  }
}
```

Function 13 Local_repaire

```
void RoutingTable::Local_repaire(nsaddr_t nod_id)
{
  // Function to remove invalid routes from the Routing Table
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  int kk = 0;
  int remove_entry = 0;

  rtable_ent *krte = NULL;
  rtable_ent ent;

  for (int max=0;max<elts;max++)
  {
```

```
    // searech for invalid routes
    if (rtab[max].metric == 250 && rtab[max].dst != rtab[max].hop && !rtab[max].q)
    {
      remove_entry = 1;
      break;
    }
  }

  if (remove_entry == 1)
  {
    int i = max;
    int k = max++;
    while (k < elts)
    {
      if ((rtab[k].metric != 250) || (rtab[k].q) || (rtab[k].metric == 250 && rtab[k].dst == rtab[k].
      {
        rtab[i] = rtab[k];
        i++;
      }
      k++;
    }

    for (ctr = i; (krte = NextLoop ()); )
    {
      if (krte->timeout_event)
        krte->timeout_event = 0;
    }

    elts = i++;

    rtable_ent *tmp = rtab;
    assert(temp);

    rtab = new rtable_ent[maxelts];
    assert(rtab);
    bcopy(tmp, rtab, elts*sizeof(rtable_ent));
    delete tmp;
  }
}
```

Function 14 Generate_Hello_Message

```
Packet *MDSDV_Agent::Generate_Hello_Message(int new_seq)
{
  double now = Scheduler::instance ().clock ();

  Packet *p = allocpkt ();
  hdr_ip *iph = hdr_ip::access(p);
  hdr_cmn *hdrc = HDR_CMN (p);


  // The packet we send wants to be broadcast
  hdrc->next_hop_ = IP_BROADCAST;
  hdrc->addr_type_ = NS_AF_INET;
  hdrc->ptype() = PT_HELLO;   // packet type is a hello message
  hdrc->error() = 0;
  hdrc->prev_hop_ = myaddr_;

  iph->saddr() = myaddr_;
  iph->sport() = ROUTER_PORT;
  iph->daddr() = IP_BROADCAST << Address::instance().nodeshift();
```

```
  iph->dport() = ROUTER_PORT;

  int change_count = 0;

  //hdrc->size_ = change_count * 12 + IP_HDR_LEN; // MDSDV + IP
  hdrc->size_ = IP_HDR_LEN;

  unsigned char *walk;

  // Allocate 5 bytes (1B for the Sequence number and 1B for the number of entries).
  p->allocdata(5);

  walk = p->accessdata ();

  *(walk++) = change_count;

  *(walk++) = new_seq >> 24;
  *(walk++) = (new_seq >> 16) & 0xFF;
  *(walk++) = (new_seq >> 8) & 0xFF;
  *(walk++) = (new_seq >> 0) & 0xFF;

  return p;
}
```

Function 15 Make_Fulldump

```
Packet * MDSDV_Agent::Make_Fulldump(nsaddr_t dst)
{
  rtable_ent *prte;
  ntable_ent *nprte;

  seqno_ += 2;     // increment my sequence number.

  int change_count;          // To store number of entries in the Full Dump.
  int rtbl_sz;               // To store the total entries in the Routing Table (RT).
  int unadvertiseable;       // To store the number of routes we can't advertise yet

  change_count = 0;
  rtbl_sz = 0;
  unadvertiseable = 0;

  double old_time;
  int old_seqnum = 0;

  int defualt =-88;
  nsaddr_t old_dst = defualt;
  uint old_metric = 255;

  // Check the Routing Table to decide how many entries should be in Full Dump.
  for (table_->InitLoop (); (prte = table_->NextLoop ()); )
  {
    if (prte->metric == 250)    // Exclude invalid routes.
      continue;

    if (prte->dst != old_dst)   // Only one route for each destination is considered.
    {
      change_count++;
      old_dst = prte->dst;
    }
    continue;
```

```
}

defualt =-88;
old_dst = defualt;
old_metric = 255;

// Allocating the packet.
Packet *p1 = allocpkt ();
hdr_ip *iph = hdr_ip::access(p1);
hdr_cmn *hdrc = HDR_CMN (p1);

double now = Scheduler::instance ().clock ();
unsigned char *walk;
unsigned char *last_walk=walk;        // to keep last position

// The packet we send wants to be unicast.
hdrc->next_hop_ = dst;
hdrc->addr_type_ = NS_AF_INET;
hdrc->ptype() = PT_FULL_DUMP;   // packet type is a Full dump.
iph->daddr() = dst;
iph->dport() = ROUTER_PORT;

hdrc->prev_hop_ = myaddr_;

//Allocate 21 Bytes for each entry + 1 Byte for cheange_count + 4 Bytes for packet sender
p1->allocdata((change_count * 21) + 5);
walk = p1->accessdata ();

*(walk++) = change_count;           // Number of entries in the Full Dump.

*(walk++) = seqno_ >> 24;           // New sequence number.
*(walk++) = (seqno_ >> 16) & 0xFF;
*(walk++) = (seqno_ >> 8) & 0xFF;
*(walk++) = (seqno_ >> 0) & 0xFF;

int num_of_entries = 0;

// Include the entries in Full Dump.
for (table_->InitLoop (); (prte = table_->NextLoop ()); )
{
  if (prte->metric == 250)        // Exclude invalid routes.
    continue;

  // Only one route is included for each destination. The route should be the best route.
  // The shortest route (with least number of hops) is chosen. If two routes have the
  // same number of hops, the one with highest sequence number is chosen.

  if (prte->dst != old_dst || ((prte->dst == old_dst) && (prte->metric < old_metric)) ||
    ((prte->dst == old_dst) && (prte->metric == old_metric) && (prte->seqnum > old_seqnum)))

  {
    if (prte->dst == old_dst)
    {
      walk=last_walk;
      change_count++;
      num_of_entries--;
    }
    last_walk=walk;     // To store the last position.

    // include this entry in the udate packet.
    *(walk++) = prte->dst >> 24;                 // destination
    *(walk++) = (prte->dst >> 16) & 0xFF;
    *(walk++) = (prte->dst >> 8) & 0xFF;
    *(walk++) = (prte->dst >> 0) & 0xFF;

    *(walk++) = prte->hop >> 24;                  // first hop
    *(walk++) = (prte->hop >> 16) & 0xFF;
```

```
      *(walk++) = (prte->hop >> 8) & 0xFF;
      *(walk++) = (prte->hop >> 0) & 0xFF;

      *(walk++) = prte->shop >> 24;                 // second hop
      *(walk++) = (prte->shop >> 16) & 0xFF;
      *(walk++) = (prte->shop >> 8) & 0xFF;
      *(walk++) = (prte->shop >> 0) & 0xFF;

      *(walk++) = prte->metric;                     // numbaer of hops

      *(walk++) = (prte->link_no) >> 24;            // link number
      *(walk++) = ((prte->link_no) >> 16) & 0xFF;
      *(walk++) = ((prte->link_no) >> 8) & 0xFF;
      *(walk++) = (prte->link_no) & 0xFF;

      *(walk++) = (prte->seqnum) >> 24;             // Sequence number
      *(walk++) = ((prte->seqnum) >> 16) & 0xFF;
      *(walk++) = ((prte->seqnum) >> 8) & 0xFF;
      *(walk++) = (prte->seqnum) & 0xFF;

      change_count--;

      old_dst = prte->dst;
      old_metric = prte->metric;
      old_time = prte->changed_at;
      old_seqnum = prte->seqnum;
    }

    prte->advertise = false;           // no need to advertise this entry again.
  }

  assert(change_count == 0);
  return p1;
}
```

Function 16 Generate_Update_Packet

```
Packet *MDSDV_Agent::Generate_Update_Packet(int change_count)
{
  //===============  Function to generate an Update Packet  ======================
  rtable_ent *prte; // pointer to an entry in the Routing Table (RT)
  ntable_ent *nprte; // pointer to an entry in the Neighbours Table (NT)

  Packet *p1 = allocpkt ();
  hdr_ip *iph = hdr_ip::access(p1);
  hdr_cmn *hdrc = HDR_CMN (p1);

  double now = Scheduler::instance ().clock ();
  unsigned char *walk;
  unsigned char *last_walk=walk;    // to store last position.

  // The packet we send wants to be broadcast
  hdrc->next_hop_ = IP_BROADCAST;
  hdrc->addr_type_ = NS_AF_INET;
  hdrc->ptype() = PT_UPDATE_PACKET;   // Packet type is an Update Packet.
  iph->daddr() = IP_BROADCAST << Address::instance().nodeshift();
  iph->dport() = ROUTER_PORT;

  // Allocate 17 Bytes for each entry + 1 Byte for cheange_count + 4 Bytes for packet sender.
  p1->allocdata((change_count * 21) + 5);
  walk = p1->accessdata ();
```

```
  *(walk++) = change_count;          // Include the number of entries of the Update packet.

  *(walk++) = seqno_ >> 24;          // Include the sender address in the Updat packet
  *(walk++) = (seqno_ >> 16) & 0xFF;
  *(walk++) = (seqno_ >> 8) & 0xFF;
  *(walk++) = (seqno_ >> 0) & 0xFF;

  int defualt =-88;
  nsaddr_t old_dst = defualt;
  uint old_metric = 255;

  int num_of_entries = 0;
  int old_seqnum = 0;
  double old_time;

  // Filling the Update Packet entries....
  for (table_->InitLoop (); (prte = table_->NextLoop ()); )
  {
    // Exclude invalid routes...
    if (prte->error || prte->metric < 1 || prte->metric == 250)
      continue;

      // include this entry in the udate packet.
      *(walk++) = prte->dst >> 24;          // save 4 Bytes for the destination in update packet
      *(walk++) = (prte->dst >> 16) & 0xFF;
      *(walk++) = (prte->dst >> 8) & 0xFF;
      *(walk++) = (prte->dst >> 0) & 0xFF;

      *(walk++) = prte->hop >> 24;         // save 4 Bytes for the first hop in update packet
      *(walk++) = (prte->hop >> 16) & 0xFF;
      *(walk++) = (prte->hop >> 8) & 0xFF;
      *(walk++) = (prte->hop >> 0) & 0xFF;

      *(walk++) = prte->shop >> 24;            // save 4 Bytes for the second hop in update packet
      *(walk++) = (prte->shop >> 16) & 0xFF;
      *(walk++) = (prte->shop >> 8) & 0xFF;
      *(walk++) = (prte->shop >> 0) & 0xFF;

      *(walk++) = prte->metric;            // save 1 Byte for the numbaer of hops in update packet

      *(walk++) = (prte->link_no) >> 24;      // save 4 Bytes for the link number in update packet
      *(walk++) = ((prte->link_no) >> 16) & 0xFF;
      *(walk++) = ((prte->link_no) >> 8) & 0xFF;
      *(walk++) = (prte->link_no) & 0xFF;

      *(walk++) = (prte->seqnum) >> 24;        // save 4 Bytes for the seq number in update packet
      *(walk++) = ((prte->seqnum) >> 16) & 0xFF;
      *(walk++) = ((prte->seqnum) >> 8) & 0xFF;
      *(walk++) = (prte->seqnum) & 0xFF;

      change_count--;

      old_dst = prte->dst;
      old_metric = prte->metric;
      old_time = prte->changed_at;
      old_seqnum = prte->seqnum;

    prte->advertise = false;             // no need to advertise this entry again
  }
  assert(change_count == 0);
  return p1;
}
```

Function 17 Generate_Failure_Packet

```
Packet *MDSDV_Agent::Generate_Failure_Packet(nsaddr_t src, nsaddr_t dst,nsaddr_t first_hop)
{
  // Generate and unicast a Failure Packet...
  Packet *p = allocpkt ();
  hdr_ip *iph = hdr_ip::access(p);
  hdr_cmn *hdrc = HDR_CMN (p);

  unsigned char *walk;

  hdrc->ptype() = PT_Failure;   // packet type is a Failure packet.

  // The packet we send wants to be unicast to a specific destination which the source node.
  hdrc->next_hop_ = src;          // Unicast to the source of the data packet.
  hdrc->addr_type_ = NS_AF_INET;
  iph->daddr() = src;      // Unicast
  iph->dport() = ROUTER_PORT;

  hdrc->error() = 0;
  iph->saddr() = myaddr_;         // the source of the Failure packet
  iph->sport() = ROUTER_PORT;

  hdrc->prev_hop_ = myaddr_;

  int change_count = 1;        // Only one entry to be included in the Failure Packet.

  p->allocdata((change_count * 12) + 1);
  walk = p->accessdata ();

  *(walk++) = change_count;

  *(walk++) = src >> 24;          // the data packet sender address.
  *(walk++) = (src >> 16) & 0xFF;
  *(walk++) = (src >> 8) & 0xFF;
  *(walk++) = (src >> 0) & 0xFF;

  *(walk++) = dst >> 24;         // the destination address .
  *(walk++) = (dst >> 16) & 0xFF;
  *(walk++) = (dst >> 8) & 0xFF;
  *(walk++) = (dst >> 0) & 0xFF;

  *(walk++) = (first_hop) >> 24;   // the first node that the source used to send the data packet.
  *(walk++) = ((first_hop) >> 16) & 0xFF;
  *(walk++) = ((first_hop) >> 8) & 0xFF;
  *(walk++) = (first_hop) & 0xFF;

  assert (!HDR_CMN (p)->xmit_failure_);
  s.schedule (target_, p, 0);
}
```

Function 18 helper_callback

```
void MDSDV_Agent::helper_callback (Event * e)
{
  // This function is called in two cases: periodic callback or timeout...
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();
  rtable_ent *prte;
  rtable_ent *pr2;
```

```
ntable_ent *nprte;

Packet *p;

// Check for periodic callback
if (periodic_callback_ && e == periodic_callback_)
{
  seqno_ += 2;

  prte = table_->GetEntry1 (myaddr_, myaddr_,&myaddr_);
  if (prte)
    prte->seqnum = seqno_;

  int change_count = 0;
  int defualt =-88;
  nsaddr_t old_dst = defualt;
  uint old_metric = 255;

  // Check the Routing Table (RT) to find valid routes.
  for (table_->InitLoop (); (prte = table_->NextLoop ()); )
  {
    // Exclude the invalid routes...
    if (prte->error || prte->metric < 1 || prte->metric == 250)
      continue;

    if (prte->dst != old_dst)
    {
      change_count++;
      old_dst = prte->dst;
    }
    continue;
  }

  // If no valid routes found in the RT, broadcast a Hello Message, otherwise broadcast an Update P
  if (change_count > 0)
    p = Generate_Update_Packet(change_count);
  else
    p = Generate_Hello_Message(seqno_);

  if (verbose_)
  {
    trace ("VPC %.5f _%d_", now, myaddr_);
    tracepkt (p, now, myaddr_, "PU");
  }

  if (p)
  { // Broadcast the packet.
    assert (!HDR_CMN (p)->xmit_failure_);
    s.schedule (target_, p, jitter(DSDV_BROADCAST_JITTER, be_random_));
  }

  s.schedule (helper_, periodic_callback_, perup_ * (0.75 + jitter (0.25, be_random_)));
  lasttup_ = now;

  return;
}

// If it is a timeout, check the Neighbours Table (NT)
for (ntable_->nInitLoop (); (nprte = ntable_->nNextLoop ());)
  if (nprte->timeout_event && (nprte->timeout_event == e))
    break;

if (nprte)
{
  if (verbose_)
    trace ("VTO %.5f _%d_ %d->%d", now, myaddr_, myaddr_, prte->dst);
```

```
if (nprte->timeout_event)
  Scheduler::instance ().cancel (nprte->timeout_event);

nprte->timeout_event = 0;
nprte->neighbour = 0;
nprte->metric = BIG;     // assign this route as an invalid route

nprte->changed_at = now;
nprte->seqnum = 0;
nprte->active = false;
seqno_ += 2;

//================  Generating and broadcasting an Error Packet  ===============
Packet *p = allocpkt ();
hdr_ip *iph = hdr_ip::access(p);
hdr_cmn *hdrc = HDR_CMN (p);

double now = Scheduler::instance ().clock ();
unsigned char *walk;
int change_count = 1;           // Number of entries in the Error Packet.

// The packet we send wants to be broadcast
hdrc->prev_hop_ = myaddr_;
hdrc->next_hop_ = IP_BROADCAST;
hdrc->addr_type_ = NS_AF_INET;
hdrc->ptype() = PT_ERROR_PACKET;   // Packet type is an Error packet
iph->daddr() = IP_BROADCAST << Address::instance().nodeshift();
iph->dport() = ROUTER_PORT;

hdrc->iface() = -2;
hdrc->error() = 0;
iph->saddr() = myaddr_;
iph->sport() = ROUTER_PORT;

p->allocdata((change_count * 16) + 1);
walk = p->accessdata ();

*(walk++) = change_count;

*(walk++) = myaddr_ >> 24;                  // the sender in Error packet
*(walk++) = (myaddr_ >> 16) & 0xFF;
*(walk++) = (myaddr_ >> 8) & 0xFF;
*(walk++) = (myaddr_ >> 0) & 0xFF;

*(walk++) = nprte->dst >> 24;               // the destination in Error packet
*(walk++) = (nprte->dst >> 16) & 0xFF;
*(walk++) = (nprte->dst >> 8) & 0xFF;
*(walk++) = (nprte->dst >> 0) & 0xFF;

*(walk++) = nprte->link_no >> 24;           // link number in Error packet
*(walk++) = (nprte->link_no >> 16) & 0xFF;
*(walk++) = (nprte->link_no >> 8) & 0xFF;
*(walk++) = (nprte->link_no >> 0) & 0xFF;

*(walk++) = seqno_ >> 24;                   // link number in Error packet
*(walk++) = (seqno_ >> 16) & 0xFF;
*(walk++) = (seqno_ >> 8) & 0xFF;
*(walk++) = (seqno_ >> 0) & 0xFF;

//I've discovedred lost link so. Broadcasting an Error packet immediatelly.
s.schedule (target_, p, 0);

//checking the Routing Table (RT)
for (table_->InitLoop (); (pr2 = table_->NextLoop ()); )
{
  if (pr2->hop == nprte->dst && pr2->metric != BIG)
  {
```

```
        if (verbose_)
          trace ("VTO %.5f _%d_ marking %d", now, myaddr_, pr2->dst);

        pr2->metric = BIG;     // assign this route as an invalid route
        pr2->changed_at = now;
        pr2->seqnum = 0;
      }
    }
    table_->local_rep(myaddr_);
  }
  else
  {
    // unknown event on  queue
    fprintf(stderr,"DFU: unknown queue event\n");
    abort();
  }
}
```

Function 19 lost_link

```
void MDSDV_Agent::lost_link (Packet *p)
{
  double now = Scheduler::instance ().clock ();

  hdr_ip *iph = hdr_ip::access(p);
  hdr_cmn *hdrc = HDR_CMN (p);

  int src = Address::instance().get_nodeaddr(iph->saddr());
  int dst = Address::instance().get_nodeaddr(iph->daddr());

  ntable_ent *nprte = ntable_->nGetEntry (hdrc->next_hop_);

  if(use_mac_ == 0)
  {
    if (nprte && nprte->timeout_event)
    {
      Scheduler::instance ().cancel (nprte->timeout_event);
      helper_callback (nprte->timeout_event);  // report lost link (timeout event) immediatly.
    }

    Generate_Failure_Packet(src, dst, hdrc->prev_hop_);  // generate and unicast a Failure packet.

    rtable_ent *prte = table_->GetEntry2(dst,-99, -99,myaddr_); // Find an alternative path.
    if (!prte || prte->metric == BIG)
    {
      // No alternative path found. Must queue the packet in the QT.
      qtable_ent *qprte = qtable_->qGetEntry (dst);  // find the entry belongs to the destination in
      if (qprte)
      {
        if (!qprte->q)
          qprte->q = new PacketQueue ();

        qprte->q->enque(p);

        if (verbose_)
          trace ("VBP %.5f _%d_ %d:%d -> %d:%d", now, myaddr_, iph->saddr(), iph->sport(),
                 iph->daddr(), iph->dport());

        while (qprte->q->length () > MAX_QUEUE_LENGTH)
          drop (qprte->q->deque (), DROP_RTR_QFULL);

        return;
```

```
      }
      else
      {
        // Brand new destination
        qtable_ent qrte;

        bzero(&qrte, sizeof(qrte));
        qrte.dst = dst;
        qrte.q = new PacketQueue();
        qrte.q->enque(p);

        assert (qrte.q->length() == 1 && 1 <= MAX_QUEUE_LENGTH);
        qtable_->qAddEntry(qrte);

        if (verbose_)
          trace ("VBP %.5f _%d_ %d:%d -> %d:%d", now, myaddr_, iph->saddr(), iph->sport(),
                 iph->daddr(), iph->dport());

        return;
      }
    }
    else
      recv(p, 0);
  }
  return;
}
```

Function 20 mac_callback

```
static void mac_callback (Packet * p, void *arg)
{
  ((MDSDV_Agent *) arg)->lost_link (p);
}
```

Function 21 updateRoute

```
int MDSDV_Agent::updateRoute(rtable_ent *old_rte, rtable_ent *new_rte)
{
  int negvalue = -1;
  assert(new_rte);

  Time now = Scheduler::instance().clock();
  new_rte->changed_at = now;
  char buf[1024];

  snprintf (buf, 1024, "%c %.5f _%d_ (%d,%d->%d,%d->%d,%d->%d,%f)",
  (new_rte->metric != BIG && (!old_rte || old_rte->metric != BIG)) ? 'D' : 'U',
  now, myaddr_, new_rte->dst, old_rte ? old_rte->metric : negvalue, new_rte->metric,
  old_rte ? old_rte->seqnum : negvalue,  new_rte->seqnum, old_rte ? old_rte->hop : -1,
  new_rte->hop, new_rte->advertise_ok_at);

  rtable_ent *prte;

  nsaddr_t *my_ip = &myaddr_;
```

```
  if (new_rte->metric > 0)
    new_rte->advertise = true;

  if (new_rte->metric > 1)
    new_rte->timeout_event = 0;

  int write =table_->Add_Entry (*new_rte, my_ip);

  if (trace_wst_)
    trace ("VWST %.12lf frm %d to %d wst %.12lf nxthp %d [of %d]", now, myaddr_,
     new_rte->dst, new_rte->wst, new_rte->hop, new_rte->metric);

  if (verbose_)
    trace ("VS%s", buf);

  return(write);
}
```

Function 22 addNewEntry

```
void MDSDV_Agent::addNewEntry(nsaddr_t dst, int new_seq)
{
  // This function is used to prepare a new entry to be inserted in the routing table.
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  rtable_ent rte;
  bzero(&rte, sizeof(rte));

  rte.dst = dst;
  rte.hop = dst;
  rte.shop = -99;
  rte.metric = 1;
  rte.seqnum = new_seq;
  rte.link_no = myaddr_ * 10000 + dst;

  rte.changed_at = now;
  rte.new_seqnum_at = now;
  rte.wst = 0;

  nsaddr_t *my_ip = &myaddr_;
  table_->Add_Entry (rte, my_ip);   // call this function to add the entry.
}
```

Function 23 addNewNeighbour

```
void MDSDV_Agent::addNewNeighbour (nsaddr_t dst, int seq)
{
  // This function is used to create an entry for a new neighbour.
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  ntable_ent rte;
  bzero(&rte, sizeof(rte));
```

```
  rte.dst = dst;
  rte.neighbour = 1;
  rte.metric = 1;
  rte.seqnum = seq;
  rte.link_no = myaddr_ * 10000 + dst;

  rte.changed_at = now;
  rte.timeout_event = 0;
  rte.timeout_event = new Event ();
  s.schedule (helper_, rte.timeout_event, min_update_periods_ * perup_);

  rte.active = true;
  rte.q = 0;

  // call nAddEntry() function to add a new neighbour in the Neigbours table.
  ntable_->nAddEntry (rte);
}
```

Function 24 Add_Entry

```
int RoutingTable::Add_Entry(const rtable_ent &ent,nsaddr_t *my_ip1)
{
nsaddr_t nod_id1 =*my_ip1;
rtable_ent *it;

assert(ent.metric <= BIG);

Scheduler & s = Scheduler::instance ();
double now = s.clock ();

int kk=0;

  int c=2;
  int N=0;   //the smallest section
  int elts1=0;
  int min1=0;
  int max1=0;
  int first_entry_position;
  int first_entry;

  int found1=0;
  int found2=0;
  int found3=0;
  int return_or_no;

  if (elts > 0)
  {
    max1=elts-1;
    if(max1==0)
      elts1=0;
    else
      elts1=round(max1/2.0);

    do
    {
      c = check1(&ent, &rtab[elts1], *my_ip1);

     if(c==0)
     {
        found1=0;
        found2=0;
```

```
found3=0;

int entry_no=0;

int position1=-99;
int position2=-99;
int position3=-99;

//  move to the 1st entry belongs to this destination
while(rtab[elts1].dst==ent.dst && elts1 >=0)
{
  elts1--;
  if (elts1 < 0)
    break;
}

elts1++;    // this is the 1st entry belongs to this destination
first_entry_position=elts1;

entry_no = elts1;

while(rtab[elts1].dst==ent.dst)
{
  if ((rtab[elts1].dst==ent.dst)&&(rtab[elts1].hop==ent.hop)&&
     (rtab[elts1].shop==ent.shop)&&(rtab[elts1].metric==ent.metric)&&
     (rtab[elts1].link_no==ent.link_no)&& elts1 >= 0)
  {
    // Just modify the sequence number and Changed_at fields
    rtab[elts1].seqnum=ent.seqnum;
    rtab[elts1].changed_at=now;
    return(0);
  }

  if ((rtab[elts1].dst==ent.dst)&&(rtab[elts1].hop==ent.hop)&&
     (rtab[elts1].metric<ent.metric) &&(rtab[elts1].metric!=BIG)&& elts1 >= 0)
    // Discard the received entry. Same destination, same first hop, and
    // greater number of hops
    return(0);

  if ((rtab[elts1].dst==ent.dst) && (rtab[elts1].hop!=ent.hop) &&
     ((rtab[elts1].hop==ent.shop)||(rtab[elts1].shop==ent.hop)) &&
     (ent.shop!=ent.dst) && (rtab[elts1].metric<=ent.metric) &&
     (rtab[elts1].metric!=BIG) && elts1 >= 0)
     // Discard the received entry. Same destination, same second hop, greater
     // number of hops, and first hop = second hop of an entry in the RT

     return(0);

  if ((rtab[elts1].dst==ent.dst) && (rtab[elts1].hop!=ent.hop) &&
     (rtab[elts1].shop==ent.shop) && (ent.shop!=ent.dst) &&
     (rtab[elts1].metric!=BIG) && elts1 >= 0)
  {
    // two entries with the same destination , different first hop, same second
    // hop and destination <> second hop
    if (rtab[elts1].seqnum>ent.seqnum || rtab[elts1].metric<ent.metric)
      // Discard the received entry. The sequence number is smaller or the number
      // of hops is greater
      return(0);
  }


  if ((rtab[elts1].dst==ent.dst)&& (rtab[elts1].link_no==ent.link_no) &&
     (rtab[elts1].metric<ent.metric) && (rtab[elts1].metric!=BIG)&& elts1 >= 0)
    // Discard the received entry. Same destination, same link no, and the number
    // of hops is greater
    return(0);
```

```
    if ((rtab[elts1].dst==ent.dst) && ((rtab[elts1].hop!=ent.hop) &&
       (rtab[elts1].shop==ent.shop)&&(rtab[elts1].dst!=ent.shop))&& elts1 >= 0)
    {
      found3=1;              // Lowest Periority
      entry_no=elts1;
      position3=elts1;
    }

    if ((rtab[elts1].dst==ent.dst)&&(rtab[elts1].link_no==ent.link_no)&& elts1 >= 0)
    {
      found2=1;             //Midium Periority
      entry_no=elts1;
      position2=elts1;
    }

    if ((rtab[elts1].dst==ent.dst) && (rtab[elts1].hop==ent.hop) &&
       (rtab[elts1].hop==BIG) && elts1 >= 0)
    {
      found1=1;            // Highest Periority
      entry_no=elts1;

      position1=elts1;
    }

    elts1++;
  }

  if (found1+found2+found3==1)
    elts1=entry_no;  // Only one entry is similar to the received entry , overwrite.

  if (found1+found2+found3==2)
  {
    if (position1==position2 || position1==position3 || position2==position3)
      elts1=entry_no;
    else
      return(0);
  }

  if (found1+found2+found3>2)
  {
    // Discard the received entry, because more than one entry similar to the received
    // entry are found.
    elts1=entry_no;
    return(0);
  }

  if (found1+found2+found3==0)
  {
    elts1=first_entry_position;
    while(rtab[elts1].dst==ent.dst)
    {
      if ((rtab[elts1].dst==ent.dst)&&(rtab[elts1].hop>ent.hop))
        break;
      elts1++;
    }

    kk=elts1;
    goto insert_entry;
  }
}

kk=elts1;
if ((c==0)||(elts1==0)||(min1==max1))
  break;

if(c==1)                //  not found , and it is could be up side part
  max1=elts1-1;
```

```
    if(max1<min1)
      max1=min1;

    if(c== 2)              // not found , and it is could be down side part
      min1=elts1+1;

    if(min1>max1)
      min1=max1;

    elts1=round((min1+max1)/2.0) ;// executed in both c==1 or c==2

  } while(elts1>=N);

}

if (c == 0 && elts != 0)
{
  it=&rtab[kk];   // A pointer to an entry (kk) in the routing table
  int F_sort=0;   // flag for swapping: zero means no need to swap two entries in the RT

  // check if we need to sort the routing table
  if((rtab[kk].dst == ent.dst) && (rtab[kk].hop != ent.hop))
    F_sort=1;    // to check if the new entry (ent) is overwritten in the right position

  if ((rtab[kk].metric > ent.metric) || (rtab[kk].seqnum < ent.seqnum) || ent.metric == BIG)
  {
    if (rtab[kk].dst == rtab[kk].hop && ent.dst != ent.hop)
       return(0);

    bcopy(&ent,it,sizeof(rtable_ent));  //  overwirte the new entry (&ent)

    nsaddr_t nod_id =*my_ip1;
    if (F_sort==1)
      // Check if the new entry (ent) is overwritten in a wrong position. If so,
      // swapping is needed
      int z= Sort_Entry(ent,first_entry_position, nod_id);

    return(1);      // return 1 when the enry has been overwritten.
  }

  return(0);       // return 0 when the enry has not been overwritten.

}

insert_entry:

// Check the RT's size. If it is full, double its size
if (elts == maxelts)
 {
   rtable_ent *tmp = rtab;
   assert(temp);

   maxelts *= 2;
   rtab = new rtable_ent[maxelts];
   assert(rtab);
   bcopy(tmp, rtab, elts*sizeof(rtable_ent));
   delete tmp;
 }


 if((c==2) && (elts!=0))
   kk++;

 int max=kk;

  // moving the entries of the RT, starting from the end of the table, one by one till
```

```
    // reaching the position where the new entry should be inserted.
   int i = elts-1;
   while (i >= max)
   {
     rtab[i+1] = rtab[i];
     i--;
   }


   if (max < 0)
     max = 0;

   //insert the new entry in the righ position (max).
   bcopy(&ent, &rtab[max], sizeof(rtable_ent));
   if ((c==1)|| (c=2 && ent.dst == rtab[max].dst))
     rtab[max].trigger_event = 0;

   elts++;     //increament elts counter

   return(1);
}
```

Function 25 Sort_Entry

```
int RoutingTable::Sort_Entry(const rtable_ent &ent,int first_entry_position, nsaddr_t nod_id)
{
  // this function is used only in one case; if a new entry (ent) is overwritten because of
  // similarity. In this case; sometimes the new entry is inserted in a wrong position

  int sort = 1;
  rtable_ent rte1;
  bzero(&rte1, sizeof(rte1));

  do
  {
    sort = 1;   // flag to continue or to stop sorting

    int kk = first_entry_position;  // first entry belongs to the destination in the RT

    // sort only a specific part of the RT (where the new entry is inserted)
    while((rtab[kk].dst == ent.dst) && (rtab[kk+1].dst == ent.dst))
    {
      // Each time check two entries. If they are not sorted, swap them
      if ((rtab[kk].hop > rtab[kk+1].hop) && (kk+1 < elts))
      {
        sort=0;

        //=======================( SWAP )=================================
        bcopy(&rtab[kk],&rte1,sizeof(rtable_ent));
        rtab[kk] = rtab[kk+1];
        bcopy(&rte1,&rtab[kk+1], sizeof(rtable_ent));
        //=======================( End of SWAP )==========================
      }
      kk++;
    }
  }while (sort == 0);

  return 0;
}
```

Function 26 GetEntry1

```
rtable_ent * RoutingTable::GetEntry1(nsaddr_t dest, nsaddr_t via, nsaddr_t *myaddr )
{
  rtable_ent ent; // a new empty entry
  ent.dst = dest; // assign new entry destination
  ent.hop = via ;   // assign new entry hop

  rtable_ent *prte;

  int c = 2;
  int N = 0;   //the smallest section
  int elts1 = 0;
  int min1 = 0;
  int max1 = 0;
  if (elts > 0)
  {
    max1=elts-1;
    if(max1 == 0)
      elts1 = 0;
    else
      elts1=round(max1/2.0);

    do
    {
      c = check2(&ent, &rtab[elts1], *myaddr);
      if ((c==0) || (elts1 == 0) || (min1 == max1))
         break;

      if(c == 1)                //  the entry is not found, could be in the upper part
        max1 = elts1-1;

      if(max1 < min1)
        max1 = min1;

      if(c == 2)                 // the entry is not found, could be in the lower part
        min1=elts1+1;

      if(min1 > max1)
        min1 = max1;

      elts1 = round((min1+max1)/2.0);     // executed in both cases (c=1 or c=2)

    } while(elts1>=N);

    if(c == 0)
    {
      // The entry is found, return it
      prte=&rtab[elts1];
      return prte;
    }

    return 0;
  }
}
```

Function 27 GetEntry2

```
rtable_ent *
RoutingTable::GetEntry2(nsaddr_t dest, nsaddr_t via, nsaddr_t from_node, nsaddr_t myaddr)
```

```
{
  //function  to get the best route
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  // Create new entry
  rtable_ent ent;       // a new empty entry
  ent.dst = dest;         // destination
  ent.hop = via ;         // First hop
  ent.metric = 250;       // we suppose the worst metric ( to get a better one)
  ent.changed_at = 0;     // we suppose the worst time ( to get a better one)
  ent.seqnum = 0;

  rtable_ent *it;

  int c = 2;
  int N = 0;
  int elts1 = 0;
  int min1 = 0;
  int max1 = 0;

  if (elts > 0)
  {
    max1=elts-1;
    if(max1 == 0)
      elts1 =0 ;
    else
      elts1 = round(max1/2.0);

    do
    {
      c = check1(&ent, &rtab[elts1], myaddr);
      if(c == 0)
      {
        int entry_no = 0;
        while(rtab[elts1].dst == ent.dst && elts1 >= 0)
        {
          elts1--;
        }
        elts1++;         // the position of the first entry belongs to the destination

        it = &rtab[elts1];

        while(rtab[elts1].dst == ent.dst)
        {
          if ((rtab[elts1].metric < ent.metric && rtab[elts1].hop != from_node) ||
              (rtab[elts1].metric == ent.metric && rtab[elts1].metric < BIG &&
              rtab[elts1].seqnum > ent.seqnum && rtab[elts1].hop != from_node))
          {
            int j=elts1;
            ent.metric = rtab[elts1].metric;
            ent.changed_at = rtab[elts1].changed_at;
            ent.seqnum = rtab[elts1].seqnum;

            it=&rtab[elts1];
          }
          elts1++;
        }
      }

      if ((c==0) || (elts1==0) || (min1 == max1))
        break;

      if(c == 1)                //  the entry is not found, could be in the upper part
        max1 = elts1-1;

      if(max1 < min1)
```

```
        max1 = min1;

      if(c == 2)              // the entry is not found, could be in the lower part
        min1 = elts1 + 1;

      if(min1 > max1)
        min1 = max1;

      elts1 = round((min1 + max1)/2.0) ;   // executed in both cases (c=1 or c=2)

    } while(elts1 >= N);

    if(c == 0 && (it->dst >= 0 && it->dst <= 250))
      return it;            // return the entry.

  }
  return 0;
}
```

Function 28 check1

```
static int check1(const void *a, const void *b,nsaddr_t nod_id)
{
  nsaddr_t dst_a = ((const rtable_ent *) a)->dst;
  nsaddr_t dst_b = ((const rtable_ent *) b)->dst;

  if (dst_a == dst_b)
    return 0;          // the entry is found

  if (dst_a < dst_b)
    return 1;          // the entry is not found continue search up

  if (dst_a > dst_b)
    return 2;          // the entry is not found continue search down
}
```

Function 29 check2

```
static int check2(const void *a, const void *b,nsaddr_t nod_id)
{
  nsaddr_t ia1 = ((const rtable_ent *) a)->dst;
  nsaddr_t ib1 = ((const rtable_ent *) b)->dst;
  nsaddr_t ia2 = ((const rtable_ent *) a)->hop;
  nsaddr_t ib2 = ((const rtable_ent *) b)->hop;

  if ((ia1 == ib1)&& (ia2 == ib2))
    return 0;          // the entry is found

  if ((ia1 < ib1) || ((ia1 == ib1) && (ia2 < ib2)))
    return 1;      // the entry is not found continue search up

  if ((ia1 > ib1) || ((ia1 == ib1)&& (ia2 > ib2)))
    return 2;      // the entry is not found continue search down
}
```

Function 30 local_rep

```
void RoutingTable::local_rep(nsaddr_t nod_id)
{
  // This function is used to remove the invalid routes from the node's routing table.
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  int kk = 0;
  rtable_ent *krte = NULL;

  rtable_ent ent;

  int max;
  int invalid_route = 0;

  for (max=0; max<elts; max++)
  {
    // check for an invalid route.
    if (rtab[max].metric == 250 && rtab[max].dst != rtab[max].hop && !rtab[max].q)
    {
      // an invalid route is found
      invalid_route = 1;
      break;
    }
  }

  if (invalid_route == 1)
  {
    int i = max;
    int k = max++;
    while (k < elts)
    {
      if ((rtab[k].metric != 250) || (rtab[k].q) ||
         (rtab[k].metric == 250 && rtab[k].dst == rtab[k].hop))
      {
        // overwrite the entry by the next one (shifting all the rest entries)
        rtab[i] = rtab[k];
        i++;
      }
      k++;
    }

    int counter1 = 0;

    for (ctr = i; (krte = NextLoop ()); )
    {
      counter1++;

      if (krte->timeout_event)
        krte->timeout_event = 0;
    }

    elts = i++;

    rtable_ent *tmp = rtab;
    assert(temp);

    rtab = new rtable_ent[maxelts];
    assert(rtab);
    bcopy(tmp, rtab, elts*sizeof(rtable_ent));
    delete tmp;
  }
}
```

Function 31 stale_routes

```
void RoutingTable::stale_routes(nsaddr_t nod_id)
{
  // This function is used to assign stale routes as invalid routes
  Scheduler & s = Scheduler::instance ();
  double now = s.clock ();

  rtable_ent *krte = NULL;
  rtable_ent ent;
  int counter1 = 0;

  for (int max = 0; max <elts; max++)
  {
    if (rtab[max].metric != 250 && rtab[max].metric > 1 && (now - rtab[max].changed_at > 25.0))
    {
      rtab[max].metric = BIG;     // assign this route as an invalid route
      counter1++;
    }
  }
}
```

Function 32 nAddEntry

```
void NeighbourTable::nAddEntry(const ntable_ent &ent)
{
  // This function is to create an entry for a new neigbour
  ntable_ent *it;
  assert(ent.metric <= BIG);

  // search for an entry belongs to the neighbour
  if ((it = (ntable_ent*) bsearch(&ent, ntab, nelts, sizeof(ntable_ent), rtent_trich)))
  {
    // the entry is found
    bcopy(&ent,it,sizeof(ntable_ent));
    return;
  }

  // check the size of the Neighbours table. If it is full, double it
  if (nelts == nmaxelts)
  {
    ntable_ent *tmp = ntab;
    nmaxelts *= 2;
    ntab = new ntable_ent[nmaxelts];
    bcopy(tmp, ntab, nelts*sizeof(ntable_ent));
    delete tmp;
  }

  int max;
  for (max=0;max<nelts;max++)
  {
    if (ent.dst < ntab[max].dst)
      break;
  }
```

```
  // shift the entries strating from the end of the table to insert the new entry in the
  // right position
  int i = nelts-1;
  while (i >= max)
  {
    ntab[i+1] = ntab[i];
    i--;
  }

  bcopy(&ent, &ntab[max], sizeof(ntable_ent));  // insert the entry
  nelts++;

  return;
}
```

Function 33 nGetEntry

```
ntable_ent * NeighbourTable::nGetEntry(nsaddr_t dest)
{
  // This function is to check if the node (dest) is a neighbour
  ntable_ent ent;
  ent.dst = dest;
  return (ntable_ent *) bsearch(&ent, ntab, nelts, sizeof(ntable_ent), rtent_trich);
}
```