July 2002

UILU-ENG-02-2206
DC-204

DESIGN, IMPLEMENTATION AND TESTING OF ROUTING PROTOCOLS FOR MOBILE AD-HOC NETWORKS

Binita Gupta

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 2002	

4. TITLE AND SUBTITLE
Design, Implementation and Testing of Routing Protocols for Mobile for Ad-Hoc Networks

5. FUNDING NUMBERS

6. AUTHOR(S)

Binita Gupta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING RGANIZATION REPORT NUMBER
UILU-ENG-02-2206
(DC-204)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

An ad-hoc network is a collection of wireless mobile hosts dynamically forming a network without the use of any preexisting infrastructure or central coordinating entity. Multi-hop routing is used to enable communication between nodes in an ad-hoc network. Many routing protocols have been proposed and studied for ad-hoc networks. Most of these studies are based on simulation results. We believe that testing and validation of ad-hoc routing protocols on real test beds is necessary before they can be implemented in the real world.

This thesis addresses two routing protocols, a proactive routing protocol, DSDV, and a reactive routing protocol AODV. It provides implementation of DSDV and AODV routing protocols on the Linux kernel. It also provides a detailed specification of the DSDV routing protocol. An improved version of DSDV called *Adaptive DSDV* has been developed for highly dynamic networks. *Adaptive DSD V* is fully automatic in terms of configuring various DSDV parameters and completely obviates the need of hard coding their values. The thesis also conducts a detailed investigation of issues involved with on-demand routing in ad-hoc networks. An Ad-hoc Support Library (ASL) has been developed for implementing on-demand ad-hoc routing protocols. ASL is used to implement the AODV routing protocol without making any changes to the Linux kernel code. The resulting implementation is more efficient than many of the existing AODV implementations in terms of per-packet processing overhead. A real testbed of laptop computers has been used for the purpose of testing and performance evaluation of the implemented routing protocols. The thesis also presents a novel approach to enable inter-operability between proactive and reactive routing protocols in an hybrid ad-hoc network. The idea is to provide a set of services which can be utilized by ad-hoc routing protocols to achieve inter-operability.

14. SUBJECT TERMS

15. NUMBER OF PAGES
163

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

DESIGN, IMPLEMENTATION AND TESTING OF ROUTING PROTOCOLS FOR MOBILE AD-HOC NETWORKS

BY

BINITA GUPTA

B.Tech, Indian Institute of Technology, Kharagpur, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

ABSTRACT

An ad-hoc network is a collection of wireless mobile hosts dynamically forming a network without the use of any preexisting infrastructure or central coordinating entity. Multi-hop routing is used to enable communication between nodes in an ad-hoc network. Many routing protocols have been proposed and studied for ad-hoc networks. Most of these studies are based on simulation results. We believe that testing and validation of ad-hoc routing protocols on real test beds is necessary before they can be implemented in the real world.

This thesis addresses two routing protocols, a proactive routing protocol, DSDV, and a reactive routing protocol AODV. It provides implementation of DSDV and AODV routing protocols on the Linux kernel. It also provides a detailed specification of the DSDV routing protocol. An improved version of DSDV called *Adaptive DSDV* has been developed for highly dynamic networks. *Adaptive DSDV* is fully automatic in terms of configuring various DSDV parameters and completely obviates the need of hard coding their values. The thesis also conducts a detailed investigation of issues involved with on-demand routing in ad-hoc networks. An Ad-hoc Support Library (ASL) has been developed for implementing on-demand ad-hoc routing protocols. ASL is used to implement the AODV routing protocol without making any changes to the Linux kernel code. The resulting implementation is more efficient than many of the existing AODV implementations in terms of per-packet processing overhead. A real test-bed of laptop computers has been used for the purpose of testing and performance evaluation of the implemented routing protocols. The thesis also presents a novel approach to enable inter-operability between proactive and reactive routing protocols in an hybrid ad-hoc network. The idea is to provide a set of services which can be utilized by ad-hoc routing protocols to achieve inter-operability.

Design, implementation and testing of routing protocols for mobile ad-hoc networks

Approved by
Prof. P.R Kumar

_____

_____

Dedicated to...

my mother and father

ACKNOWLEDGMENTS

I am grateful to my advisor, Prof. P. R. Kumar, for his valuable guidance and enthusiastic support throughout the entire course of this thesis work. Ever since I have known him, he has been a great source of inspiration to me for doing innovative research work. Constant encouragement and motivation from him were keys to the successful completion of this thesis research. I would also like to thank Prof. R. S. Sreenivas for his useful suggestions at various stages of this thesis work.

I am indebted to my parents who have always inspired me to strive for excellence in every field I have worked in. It would have been impossible to complete this thesis work to my fullest satisfaction without their blessings and support.

I would also like to thank my colleague Vikas Kawadia for his numerous handy tips during the course of this project. Last but not the least, I would like to thank all my friends at UIUC who made my stay here enjoyable and memorable during the period of this thesis project.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

Introduction

1.1 Background

Since its emergence in the 1970's, wireless communication between mobile users in the form of mobile cellular systems has proliferated tremendously, while, on the other hand, interest in and attention to infrastructureless communication has greatly increased. This is due to the recent technological advances in cellular systems, notebook computers, personal digital assistants (PDAs) and wireless data communication devices such as wireless modems and wireless LANs. This has resulted in wireless mobile systems with decreasing prices and increasing data rates, which are the two main reasons for the rapid growth of wireless mobile computing.

There are several ways to enable wireless communication between mobile hosts. The first approach is to let the existing cellular network infrastructure carry both the data and voice traffic. The major problems which arise include the problem of hand-off, viz. how to smoothly hand over a connection from one base station to another without noticeable delay or packet loss. A restriction of cellular systems is that it is limited to places where such cellular network infrastructure exists.

Another approach is to use the existing Internet for communication between mobile hosts. One major difficulty that arises is due to the fact that the routing among the hosts in the Internet is based on the hierarchical addressing of these hosts. So, as soon as a host leaves the network it is configured for, it won't be able to receive any data traffic sent to it. To solve this

problem the Mobile IP protocol [1] has been developed to support mobility at the IP layer. This approach too requires infrastructure support for its proper functioning.

A third approach is to form an ad-hoc network among all the hosts wanting to communicate with each other. This requires hosts in the ad-hoc network to relay data traffic for others in the network. Thus, a host in an ad-hoc network also acts as a router. There are many advantages of ad-hoc networks as compared to infrastructured networks. These include setup on-demand, ease of deployment, fault tolerance, unconstrained connectivity and self-configurability.

Since ad-hoc networks do not rely on any pre-existing infrastructure, they can be deployed in places with no infrastructure. This is useful in disaster recovery situations or scenarios with non-existing or damaged communication infrastructure where rapid deployment of a communication network is needed [2]. Ad-hoc networks can also be useful in situations such as conference site where participants can form a temporary network without engaging the services of any pre-existing network. In addition ad-hoc networks also offer a very promising solution for personal area networks, sensor networks and home networking.

1.2 Infrastructured Wireless Networks

In this section we provide a brief overview of networks which require a preexisting infrastructure.

1.2.1 Cellular Networks

In a cellular network [3], the geographic area is split into cells. Such networks are characterized by a backbone of stationary nodes called base station (BS) or access point (AP). Each cell is managed by one base station. Adjacent base stations typically operate on different channels to prevent interference. Activities among different base stations are coordinated by Mobile Switching Centers (MSCs). Mobile nodes within a cell communicate with their nearest base station. A registration is performed with the nearest base station every time a mobile node is first switched on. Seamless connectivity is ensured by the process of hand-off when a mobile host moves from one cell to another. The base station continuously monitors the mobile host's

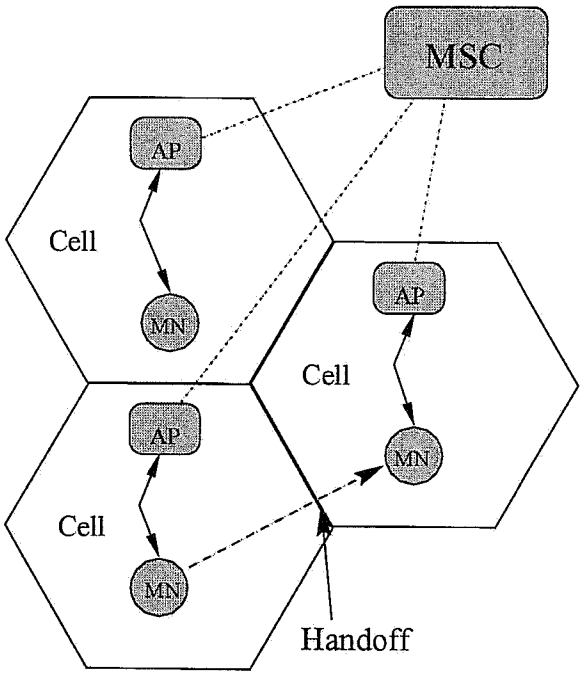


Figure 1.1 Hand-off in a Cellular Network

signal strength and passes this information to the MSC. When this signal strength drops below a certain threshold, the MSC instructs the base station to perform a hand-off. This hands over the management of the mobile host to another base station receiving a stronger signal. Figure 1.1 depicts a simple cellular network illustrating the process of hand-off between two access points. Such wireless network infrastructures are also commonly used inside buildings and on campuses to establish wireless local area networks.

1.2.2 Mobile Internet

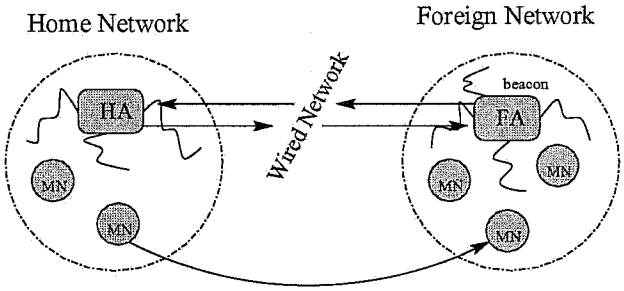
The Internet Protocol (IP) was designed with two assumptions in mind. First, a node's point of attachment is assumed to remain fixed, and, second, a node's IP address identifies the network to which it is attached. If a node could move around on the Internet without changing its IP address, it would no longer be possible to correctly route datagrams to it based solely on its IP address. To overcome this problem, the Mobile Internet Protocol [1, 4] was developed. There are three basic entities defined in the protocol. The Mobile Node (MN), the Home Agent

(HA), and the Foreign Agent (FA). A Mobile Node is a host or a router that changes its point of attachment from one network or subnetwork to another without changing its IP address. The network for which the mobile node is originally configured for, is called the home network. A Home Agent is a router on the mobile node's home network. It maintains a repository of current locations of all its mobile nodes, which it uses to relay datagrams to these nodes. A Foreign Agent is a router on the foreign network that takes care of a Mobile Node while the Mobile Node is away from its home network.

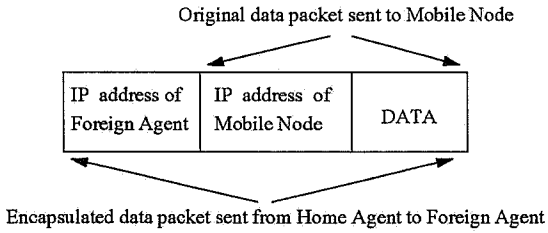
Mobility agents (Home Agents and Foreign Agents) advertise their presence by periodic Agent Advertisements or beacons (Figure 1.2a). A "beacon" is a control message which is used by a node to announce its own presence to the rest of the nodes in the network. A Mobile Node receives these advertisements and determines whether it is on the home network or the foreign network. If it detects that it has moved to a foreign network, it obtains a *care-of-address* on the foreign network. The Foreign Agent then contacts the Mobile Node's Home Agent to register the new care-of address for the Mobile Node. Packets sent to the Mobile Node's home address are received by the Home Agent. These packets are then relayed from the Home Agent to the Mobile Node using *IP in IP encapsulation* or *tunneling* (Figure 1.2b). The data packet at the Home Agent is encapsulated with another IP header with destination field set to the care-of-address of the Mobile Node. Upon receiving the packet, the Foreign Agent strips the extra header off of the packet, and then forwards it to the mobile node. In the reverse direction, packets initiated by the mobile node are directly sent to the intended recipienti, as routinely done in IP.

1.3 Infrastructure-less Wireless Networks

Infrastructure-less networks, e.g., mobile ad-hoc networks allow a collection of mobile nodes (MNs) to communicate with each other without the need of any preexisting infrastructure. Each of the nodes has a wireless interface for communication over radio or infrared links. Every node in an ad-hoc network serves as a router, and the network is said to be "peer-to-peer"



(a) Mobile IP protocol



(b) IP in IP encapsulation

Figure 1.2 Mobile IP

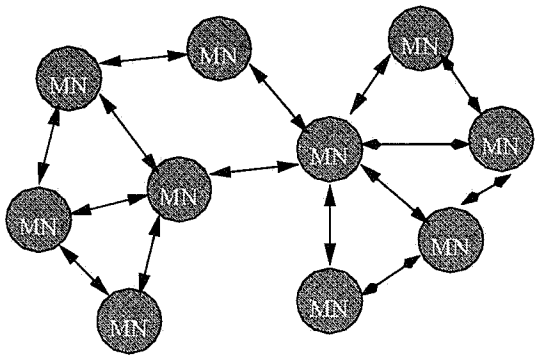


Figure 1.3 Example of an Ad-hoc Network

network. Multiple hops may be used to connect a given source and destination. Figure 1.3 illustrates an ad-hoc network.

Mobile ad-hoc networks give rise to a number of interesting challenges due to possibly higher transmission error rates, variable capacity links, limited communication bandwidth, broadcast nature of communication, dynamically changing topologies and limited battery lifetime. Three main issues which needs to be considered for correct and efficient functioning of mobile ad-hoc networks are:

- Media Access Control
- Power Control
- Routing

1.3.1 Media Access Control

The wireless medium distinguishes itself from the wired medium by the fact that it is a shared medium. Hence, transmissions can interfere with each other, causing collisions. The Media Access Control (MAC) problem is to schedule transmissions online in a distributed asynchronous manner so that packets reach their intended one hop neighbor recipients. The standard MAC schemes from the wired world (e.g. Carrier Sense Medium Access with Collision Detection (CSMA/CD) [5]) can not be employed in ad-hoc networks because of the well known

hidden and exposed terminal problems [6]. Thus, more elaborate schemes are needed for media access control in ad-hoc networks. IEEE 802.11 protocol [7, 8] uses a four-way handshake in the form of RTS, CTS, DATA and ACK packets for each and every data packet. The SEEDEX MAC protocol [9] employs a random schedule, driven by pseudo-random number generators, to reduce collisions without making explicit reservations for each and every data packet.

1.3.2 Power Control

Power control problem is the problem of determining the power level at which a packet should be transmitted in an ad-hoc network. Transmit power control is important in wireless ad-hoc networks because it can impact the battery lifetime of mobile nodes, as well as the traffic carrying capacity of the entire network. There are a number of protocols which have been proposed for power control in ad-hoc networks. These can be loosely classified into three categories. The first class comprises of strategies to find an optimal transmit power to control the connectivity properties of the network or a part of it, which could be power per node, per link or a single power level for the entire network. The second class of approaches could be called power aware routing. Most schemes use the distributed Bellman-Ford algorithm with power as the metric. The third class of approaches aim at modifying the MAC layer for achieving power control. A more comprehensive discussion of power control can be found in [10].

1.3.3 Routing

Routing in an ad-hoc network is complicated by the fact that there are unique characteristics of these networks that make traditional routing protocols inapplicable. Ad-hoc networks are characterized by dynamic interconnection topology caused by node mobility. Additionally it is required that such networks be self-configurable. Thus, a routing protocol is needed which can dynamically discover the routes, while necessitating minimal configuration effort. Although mobile computers in ad-hoc networks can be modeled as routers, existing routing protocols would place too heavy a computational burden on these mobile hosts. Moreover, the convergence characteristics of wire-line routing protocols are not good enough for the needs of ad-hoc

networks. The wireless access medium, the radio environment, also has special characteristics that must be considered when designing routing protocols for ad-hoc networks. One example of this is that unidirectional links may form when two nodes use different strengths for their transmissions, allowing only one of the hosts to hear the other. Unidirectional links can also result from the surrounding interference. Limited CPU, storage and battery capacity of the mobile nodes are other important factors which should be taken into account while designing an ad-hoc routing protocol.

A number of routing protocols have been proposed for Mobile Ad-hoc Networks (MANET) [11], but very few of them have actually been implemented and tested on real systems.

Most of the studies on ad-hoc routing protocols are currently based on simulation data. We believe that testing and validation of these routing protocols on real test beds is necessary to understand the actual performance of these protocols. In view of this, this thesis provides implementation of two existing ad-hoc routing protocols, DSDV and AODV, on the Linux kernel. It details a fully automatic and self-configurable version of DSDV, called Adaptive DSDV, for highly dynamic ad-hoc networks. This thesis also conducts an in-depth exploration of issues involving on-demand ad-hoc routing protocols. An Ad-hoc Support Library (ASL) has been developed to provide services for implementing on-demand ad-hoc routing protocols. The thesis also presents a novel approach to enable inter-operability between proactive and reactive routing protocols in an hybrid ad-hoc network.

1.4 Thesis Layout

The rest of this thesis is organized as follows. Chapter 2 provides a basic overview of various routing mechanisms, and a summary of existing routing protocols in the Internet. Chapter 3 describes different ad-hoc routing protocols. Chapter 4 is an in-depth description of the DSDV ad-hoc routing protocol. The implementation details for the DSDV protocol are described in Chapter 5. Chapter 6 presents the design and implementation of the Adaptive DSDV (A-DSDV) routing protocol. Chapter 7 discusses the various issues involved with on-demand routing in ad-hoc networks and the Ad-hoc Support Library (ASL) in detail. Details about AODV

routing protocol are provided in Chapter 8. Chapter 9 provides a detailed description of the implementation of AODV. Chapter 10 presents a novel idea to enable inter-operability between proactive and reactive routing protocols in hybrid ad-hoc networks. Chapter 11 presents the testing and experimentation results and their analysis. Chapter 12 concludes this thesis.

CHAPTER 2

An Overview of Routing Protocols

A routing protocol is needed whenever a packet needs to travel multiple hops from its source to its intended destination. A routing protocol has two main functions: discovery of routes for various source-destination pairs, and the delivery of messages to their correct destinations. While the first function is fairly complicated, the second function can be achieved in a straight forward manner using the routing table data structure maintained inside the kernel. The more difficult task is to populate this data structure with correct and relevant routing information. There are a number of basic schemes to carry out this functionality.

2.1 The Basic Routing Schemes

There are three basic routing schemes which are employed in traditional routing protocols. These are link state routing, distance vector routing and source routing [12].

2.1.1 Distance Vector Routing

The "distance vector" at a node can be regarded as a list of all the nodes in the network along with the number of hops needed to reach the node, and the next node on the path. The distance vector routing algorithm is also known as "Distributed Bellman-Ford" [13] algorithm based on the names of its inventors. A distance-vector algorithm is a completely distributed algorithm. Every node keeps a list of all known routes in a table called routing table. The routing table contains entries for all the reachable destinations in the network. Each table

entry specifies values for how far away that destination is (the distance or cost), and what node is the next hop on the route to that destination (the vector). Each node monitors the cost of all of its outgoing links, but instead of broadcasting this information to all the nodes, it periodically broadcasts to each of its neighbors an estimate of the shortest distance to every other node in the network. This information is used by the receiving node to recalculate its routing table. A "broadcast" of routing information means transmitting it on the IP limited broadcast address, "255.255.255.255".

Distance vector algorithms are computationally efficient, easy to implement, and require much less storage space. However, these algorithms do not scale well because routing update messages contain an entry for every node in the network. There are two main problems associated with distance-vector protocols. First, a distance-vector algorithm works fine only in a static environment. If links or routers fail, the distance-vector routing approach converges slowly due to the *count to infinity* problem [12, 14], in which inconsistencies arise because routing update messages propagate slowly across the network. Because of its slow convergence, the protocol might not stabilize rapidly enough in a network where routes change rapidly. Second, a distance vector protocol can result in the formation of both short-lived and long-lived routing loops. The primary cause for this is that the nodes choose their next hops in a completely distributed manner based on information that can be stale. Many distance vector algorithms use techniques called *split horizon, poison reverse, triggered updates* and *hold down* [12] to handle routing loops, though not always successfully.

2.1.2 Link State

The "link state" is basically a snapshot of the graph of the complete network. In a link state routing protocol, each node maintains a view of the complete network topology along with a cost for each link. To keep these views consistent, each node periodically transmits the link costs of its outgoing links to all other nodes in the network. This is done through flooding the network with link status control messages. The "flooding" [13] ensures that all the nodes participating in the routing protocol get a copy of the link-state information from all

the other nodes in the network. When any node receives this information, it updates its view of the network topology and applies a shortest path algorithm to choose the next hop for each destination.

One of the chief advantages of a link state algorithm is that each node computes routes independently using the link state it has. Since the routes are computed locally, they are internally consistent and so a link state protocol is guaranteed to converge. Also, because link status messages carry information only about the direct links from a single node, the size of these messages does not depend on the number of nodes in the network. This makes link state protocols scale better than distance-vector routing protocols [14]. Some link costs in a node's view can be incorrect because of long propagation delays, partitioned networks, etc. Such inconsistent network topology views can lead to the formation of routing loops. These loops are however short-lived, because they disappear in the time it takes a message to traverse the diameter of the network.

2.1.3 Source Routing

By source routing [15] we mean that each packet must carry the complete path that the packet should take through the network from source to destination. The routing decision is therefore made at the source node. The advantage of this approach is that it is easy to avoid routing loops. The disadvantage is that each packet requires an overhead. Source routing is not efficient for high mobility scenarios since the source supplied route may be inaccurate.

2.2 A Classification of Routing Protocols

Routing protocols can be further classified into several different ways.

- Centralized or Distributed
- Static or Adaptive
- Reactive or Proactive or Hybrid.

One way to categorize the routing protocols is to divide them into centralized and distributed algorithms. In centralized routing algorithms, all routing decisions are made at a central node, while in distributed algorithms, the route computation task is shared by all the nodes in the network.

Another classification of routing protocols relates to whether they change routes in response to the traffic input patterns. In static algorithms, the routes used by source-destination pairs are fixed regardless of traffic conditions. They can only change in response to node or link failures. This type of algorithm can not achieve high throughput under a broad variety of traffic input patterns. Most major networks use some form of adaptive routing where the routes used may change in response to congestion.

A third classification that is more related to ad-hoc networks is to classify the routing algorithms as proactive, reactive or hybrid. Proactive protocols attempt to continually evaluate the routes within the network, so that when a packet needs to be forwarded, the route is already known and can be immediately used. The family of distance-vector protocols is an example of a proactive scheme. Reactive protocols, on the other hand, invoke a route determination procedure only on-demand. Thus, when a route is needed, some sort of global search procedure is employed. The family of classical flooding algorithms belong to the reactive group. Hybrid methods make use of both these schemes to provide a more efficient routing algorithm.

The advantage of the proactive schemes is that, when a route is needed, there is no delay incurred in route determination. In reactive protocols, because route information may not be available at the time a route request is received, a delay is incurred to determine a route, which may be quite significant. Furthermore, the global search procedure of the reactive protocols requires significant control traffic. Pure reactive routing protocols may not be applicable in real-time applications.

2.3 Routing Protocols Used in the Internet

The Internet comprises of a large number of interconnected autonomous systems (ASs) each of which constitutes a distinct routing domain. Such autonomous systems are usually run

by a single organization or administrative entity such as a company or university. The basic idea behind autonomous systems is to provide an additional way to hierarchically aggregate routing information in the large Internet, thus improving scalability. The routing problem can then be divided into two parts: routing within a single autonomous system (intradomain routing), and routing among autonomous systems (interdomain routing). Routing protocols used for intradomain routing are known as interior gateway protocols (IGPs). Exchange of routing information among autonomous systems is achieved through exterior gateway protocols (EGPs). In addition to improving scalability, the AS model decouples the intradomain routing that takes place in one AS from that taking place in another.

The two most common interior gateway protocols for the Internet are Routing Information Protocol (RIP) [16, 14] and Open Shortest Path First (OSPF) [16, 14]. The first one is a distance-vector protocol, while the later is based on the idea of link state routing. The Border Gateway Protocol (BGP-4) [17] is the exterior gateway protocol currently used in the Internet.

2.3.1 The Routing Information Protocol (RIP)

Of all the interior gateway routing protocols, RIP is probably the most widely used. RIP was ported to TCP/IP when LANs first appeared in early 80s. The underlying RIP protocol is a straightforward implementation of distance-vector routing. It partitions participants into *active* and *passive* machines. Active routers advertise their routes to others; passive machines listen and update their routes based on advertisements, but do not advertise themselves. A router running RIP in active mode broadcasts routing update message every 30 seconds. Each message consists of pairs, where each pair contains an IP network address and the distance to that network. RIP uses the hop-count as the metric to measure the distance to a destination. The hop count along a path from a given source to a given destination refers to the number of routers that a datagram encounters along that path. All routes in RIP have timeouts associated with them and are deleted if routing updates are not received for those destination networks before the timeout.

Being a distance vector protocol, RIP does not guarantee freedom from routing loops. It employs various techniques to reduce (but not eliminate) routing loops. These include *split horizon, poison reverse, triggered updates* and *hold down.* While these techniques solve looping for certain scenarios, they do not make RIP loop free for all possible network topologies.

2.3.2 The Open Shortest Path First (OSPF) Protocol

Open Shortest Path First (OSPF) [16] is a link state routing protocol with a complex set of options and features. Sanctioned by the Internet Engineering Tast Force (IETF) [18], it is intended to become Internet's preferred interior gateway routing protocol. Some of the salient features and advantages of OSPF are:

- Changes in an OSPF network are propagated quickly.
- OSPF is hierarchical, using area 0 as the top of the hierarchy. These areas can be used to logically segment the OSPF network to decrease the size of routing tables.
- OSPF can fully support sub-netting, including Variable Length Subnet Mask (VLSM) and non-contiguous subnets.
- After initialization, OSPF only sends updates of the routing table sections which have changed; it does not send the entire routing table. OSPF periodically sends out "hello" packets to check the status of other routers.

In general, *Hello* packets are used by the routers to announce their presence to the rest of the network as well as to obtain routing information from their neighboring routers. A hello packet typically contains the identity of the originating node, along with some other informations. OSPF uses *Dijkstra's Shortest Path* algorithm [19] to compute the shortest paths to all the other nodes in the network. It uses the process of *relaxation* to compute the shortest path from a single source node to all other destination nodes in the network.

Some of the disadvantages of OSPF include its complexity and its demand on memory and computation. Another disadvantage of OSPF arises in a scenario where an entire network

is running OSPF, but one link within it is "bouncing" every few seconds. In such a case OSPF updates would dominate the network by informing every other router every time the link changes state.

2.3.3 The Border Gateway Protocol (BGP 4)

The Border Gateway Protocol (BGP) [17, 16] is the exterior gateway protocol used in the Internet. It is a very robust and scalable routing protocol. BGP uses TCP as its transport protocol. BGP neighbors exchange full routing information when the TCP connection between neighbors is first established. When changes to the routing tables are detected, the BGP routers send to their neighbors information about only those routes that have changed. BGP routers do not send periodic routing updates, and BGP routing updates advertise only the "preferred" path to a destination network. BGP uses a number of attributes to determine the best route when multiple paths exist to a particular destination.

2.4 Routing in Ad-hoc Networks

Traditional routing protocols are not suitable for ad-hoc wireless networks for the following reasons:

- These protocols are designed for a static topology and would have problem converging to a steady state in an ad-hoc network with a frequently changing topology.
- Traditional link state and distance-vector routing protocols are highly dependent on periodic control messages. This requires frequent exchange of large control packets among the network nodes. This is not very desirable for ad-hoc networks because of scarce wireless transmission bandwidth. These protocols also maintain routes to every node in the network, which consume a lot of available resources.
- Traditional routing protocols are too computationally intensive to be useful for ad-hoc networks where nodes have limited CPU, memory and battery capacity.

- These protocols assume existence of bidirectional links for their functionality. In the wireless radio environment this might not always be the case.

The routing protocols designed for ad-hoc wireless networks should possess certain properties to meet their specific needs [20]. Following are some of the properties which are desirable for ad-hoc routing protocols:

Distributed Operation: The routing protocol must be distributed in order to increase reliability. In a dynamic topology, as in ad-hoc networks, each node must be intelligent enough to make routing decisions using other collaborative nodes.

Loop Free: The routing protocol should supply loop-free routes. This improves overall performance by avoiding waste of transmission bandwidth or CPU consumption.

Demand Based Operation: The routing algorithm should adapt to the generated traffic only on-demand. The protocol should only react when needed and periodic broadcast of control information should be avoided for efficient utilization of available resources. The obvious drawback of this is increased latency.

Proactive Operation: For certain systems, additional latency incurred due to on-demand based routing might not be acceptable. Thus, if enough resources are available, a proactive mode of operation must be supported.

Unidirectional Link Support: The radio environment can cause the formation of unidirectional links. The designed routing protocol should be able to detect and utilize such links.

Power Conservation: The nodes in an ad-hoc network can be laptops or thin clients, such as PDAs, that are very limited in battery power, and therefore use some sort of stand-by mode to save power. It is important that ad-hoc routing protocols have support for these sleep modes.

Security: The radio environment is especially vulnerable to impersonation attacks. So, to ensure the desired behavior from the routing protocol, some sort of preventive security measures are required. Authentication and encryption is probably one alternative. The problem here lies in the distribution of keys among the nodes in the ad-hoc network.

Multiple Routes: Multiple routes could be used to increase the traffic carried by the network, and also to reduce the number of reactions to topology changes and congestion. If one route has become invalid, it is possible that another stored route is still valid. This saves the routing protocol from initiating a new route discovery.

Quality of Service Support: Some sort of Quality of Service support is probably necessary to incorporate into these routing protocols. This depends on what these networks will be used for, e.g., real time traffic.

None of the proposed routing protocols from MANET posses all of these properties. Since many routing protocols are still under development, it is expected that they will support extended functionality in the future. Various MANET routing protocols are described in the next chapter.

CHAPTER 3

Routing Protocols for Ad-hoc Networks

There are three categories of unicast ad-hoc routing protocols: proactive, reactive and hybrid [21]. Proactive or table-driven protocols attempt to maintain, at each node, consistent, up-to-date information for all destinations. Examples are DSDV [22], CGSR [23], WRP [24], FSR [25], GSR [26] etc. Reactive or on-demand protocols attempt to minimize overhead by discovering routes only on-demand. Examples include AODV [27], TORA [28], DSR [29], SSR [30] etc. Hybrid protocols combine aspects of proactive and reactive protocols. An examples is ZRP [31]. These protocols can be subdivided into two further categories, Flat and Hierarchical. In flat routing schemes, each node maintains a routing table with entries for all the nodes in the network. Such schemes do not scale well to large networks.

Hierarchical schemes scale well and can be used in large networks. In a hierarchical routing scheme, a network consists of two kinds of nodes, endpoints and switches, or cluster heads. Only endpoints can be sources or destinations of data traffic, and only cluster heads can perform routing functions. To form the lowest level partitions in the hierarchy, endpoints choose the most convenient switches to which they will associate by checking radio link quality. Autonomously, they group themselves into cells around these switches. This is called "cell formation." Each endpoint is within one hop of the switch with which it is affiliated. The switches, in turn, organize themselves hierarchically into clusters, each of which functions as a multihop packet-radio network. First level cluster heads organize to form higher level clusters, and so on. This procedure is called "hierarchical clustering."

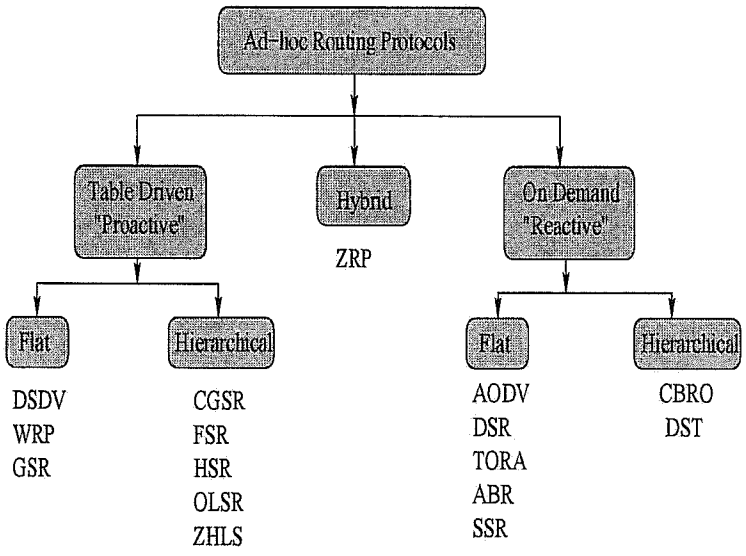


Figure 3.1 Categorization of Unicast Ad-hoc Routing Protocols

The major advantage of hierarchical routing is the efficient utilization of radio channel resources and the drastic reduction of routing table storage and transmission and processing overhead. Figure 3.1 depicts the categorization of unicast ad-hoc routing protocols.

3.1 Proactive (Table-Driven) Protocols

These protocols require mobile nodes to maintain (one or more) routing tables containing routing information to every other node in the network. Nodes respond to changes in the network topology by propagating routing updates throughout the network in order to maintain a consistent view of the network. These protocols differ in the method by which the topology change information is distributed across the network, and the number of necessary routing related tables.

3.1.1 Destination Sequenced Distance Vector (DSDV)

Destination Sequenced Distance Vector (DSDV) [22] is a distance vector routing protocol where each node maintains a routing table and periodically broadcasts routing updates. The

advantage with DSDV over traditional distance vector protocols is that DSDV guarantees freedom from routing loops by the use of sequence numbering associated with every destination. Details of DSDV are presented in Chapter 4.

3.1.2 Clusterhead Gateway Switch Routing (CGSR)

Clusterhead Gateway Switch Routing (CGSR) [23] uses as its basis the DSDV routing algorithm. It differs from the DSDV routing protocol in the type of addressing and network organization scheme employed. Instead of a flat network, CGSR is a clustered multihop mobile wireless network with several heuristic routing schemes. A clusterhead is responsible for controlling a group of ad-hoc nodes. A distributed clusterhead election algorithm is used to elect a clusterhead. To avoid adversely affecting routing protocol performance by frequent cluster head changes, a Least Cluster Change (LCC) clustering algorithm is used. Using LCC, cluster heads only change when two cluster heads come into contact, or when a node moves out of contact of all other cluster heads.

Since CGSR uses DSDV as the underlying scheme, it has much of the same overhead as DSDV. However, it modifies DSDV by using a hierarchical cluster head-to-gateway routing approach to route traffic from source to destination. Gateway nodes are nodes which are within communication range of two or more cluster heads. A packet sent by a node is first routed to its cluster head, and then the packet is routed from the cluster head to a gateway to another cluster head, and so on until the cluster head of the destination node is reached. The packet is then transmitted to the destination. Figure 3.2 illustrates an example of CGSR scheme. Each node must keep a cluster member table to store the destination cluster head for each mobile node in the network. These tables are periodically broadcasted using DSDV algorithm. Each node also maintains a routing table which is used to determine the next hop in order to reach the destination. On receiving a packet, a node consults its cluster member table and routing table to determine the nearest cluster head along the route to the destination. Next the node checks its routing table to determine the next-hop node to reach the selected cluster head. It then transmits the packet to this node.

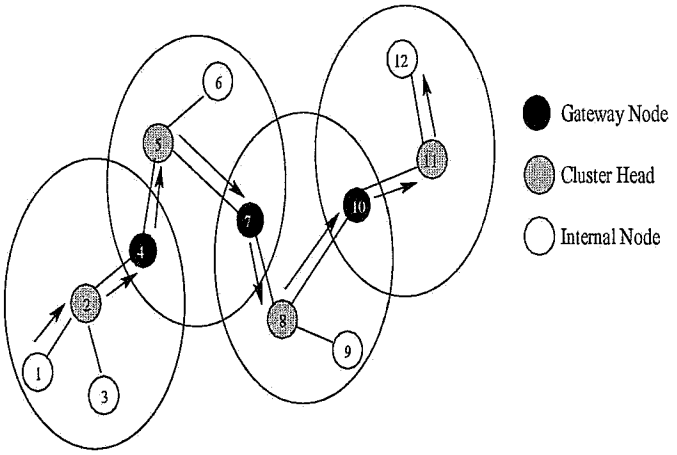


Figure 3.2 Example of CGSR routing from source node 1 to destination node 12

3.1.3 Wireless Routing Protocol (WRP)

The Wireless Routing Protocol (WRP) [24] is a path finding algorithm, and is based on distance vector routing. To avoid routing loops, each node in WRP communicates the distance and second-to-last hop (predecessor) information for each destination in the wireless network. At each node a consistency check of the received predecessor information is done to prevent routing loops. Each node in WRP maintains four tables: (a) distance table, (b) routing table, (c) link-cost table, and (d) message retransmission list (MRL) table. Each entry of the MRL contains the sequence number of the update message, a retransmission counter, an acknowledgement-required flag vector with one entry per neighbor, and a list of updates sent in the update message. The MRL records which updates in an update message need to be retransmitted and which neighbors should acknowledge the retransmissions.

Mobiles inform each other of link changes through the use of update messages. An update message is sent only between neighboring nodes and contains a list of updates (the destination, the distance to the destination, and the predecessor of the destination), as well as a list of responses indicating which mobile nodes should acknowledge the update. Mobile nodes send update messages after processing updates from neighbors or detecting a change in links to one of the neighbors. Upon receiving an update message, a node updates its distance table and

checks for new possible paths through other nodes. Any new paths are relayed back to the original node. WRP uses hello messages for connectivity if there are no other messages to indicate connectivity.

3.1.4 Global State Routing (GSR)

Global State Routing (GSR) [26] takes the idea of link state routing but improves it by avoiding flooding of routing messages. In this protocol, each mobile node has knowledge of the full network topology and maintains a Neighbor list, a Topology table, a Next Hop table and a Distance table. The Neighbor list of a node contains the list of all its neighbors. For each destination node, the Topology table contains the link state information as reported by the destination and the timestamp indicating the time the destination generated this information. For each destination, the Next Hop table contains the next hop to which the packet for this destination must be forwarded. The Distance table contains the shortest distance to each destination node. GRP uses a weight function to compute the distance of a link.

At the beginning, each node starts with an empty neighbor list and empty topology table. The node learns about its neighbors by examining the sender field of each packet in the inbound queue. Upon receiving a routing message, a node updates its topology table if the sequence number of the message is newer than the sequence number stored in the table. The concept of sequence number is borrowed from DSDV (see Chapter 4) and is used to distinguish stale links from the new ones. The updated topology table is used to compute the shortest path from this node to all other nodes in the network using Dijkstra's algorithm [13, 19]. The mobile node then rebuilds the routing table based on the newly computed topology and broadcasts the new information to its neighbors. Since the global topology is maintained at each node, preventing routing loops is easier.

3.1.5 Fisheye State Routing (FSR)

Fisheye State Routing (FSR) [25], an implicit hierarchical protocol, is an improvement over GSR. It saves considerable amount of network bandwidth by using a "fisheye" technique to

reduce the size of update messages without seriously affecting routing accuracy. In routing, the fisheye approach translates to maintaining accurate distance and path quality information about the immediate neighborhood of a node, with progressively less details as the distance increases.

In FSR, each node maintains a link state table based on the up-to-date information received from neighboring nodes, and periodically exchanges it with its local neighbors only (no flooding). While processing routing messages, entries with higher sequence numbers replace the ones with smaller sequence numbers. Nodes use a fisheye approach while exchanging routing information. Figure 3.3 illustrates the application of fisheye in a mobile, wireless network. The circles with different shades of gray define the fisheye scopes with respect to the center node (node 11). The fisheye scope is defined as the set of nodes that can be reached with a given number of hops. Figure 3.3 shows three scopes for 1, 2 and >2 hops. Nodes are color coded as gray, black, and white accordingly.

In FSR, entries in a routing table corresponding to nodes within the smallest scope are propagated to the neighbors with the highest frequency. The rest of the entries are sent out with lower frequency. Thus, by using different exchange periods for different entries in the routing table, routing overhead is reduced. This strategy produces timely updates from near stations but creates large latencies from stations afar. Even though the nodes do not have accurate information about the distant nodes, the packet is routed correctly, because, when a packet approaches its destination, it finds increasingly more accurate routing information as it enters sectors with a higher fresh rate.

3.1.6 Hierarchical State Routing (HSR)

Hierarchical State Routing (HSR) [25] is a hierarchical link state based routing protocol. The characteristic features of HSR are multilevel clustering and logical partitioning of mobile nodes. The network is partitioned into clusters and a cluster head is elected as in cluster-based algorithm. The Least Cluster Change (LCC) algorithm is adopted to reduce the number of times the cluster head changes.

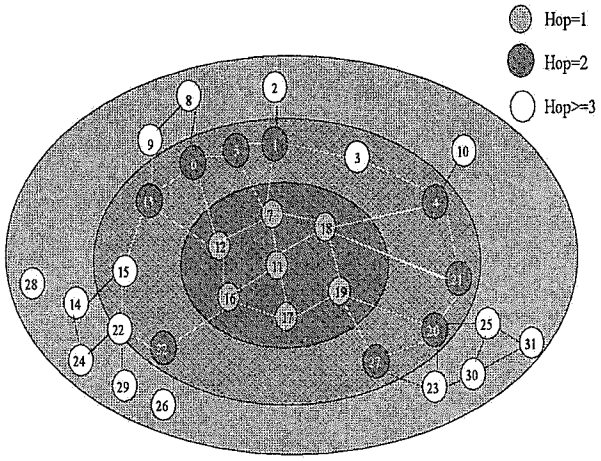


Figure 3.3 Scope of Fisheye in an ad-hoc network

Within a physical cluster, each node monitors the link status to each neighbor and broadcasts it within the cluster. The cluster head summarizes this information within its cluster and propagates it to neighboring cluster heads via gateways (nodes lying within more than one cluster). The cluster heads at lower levels again organize themselves into clusters and so on. Only the clusters at the lowest level consist of physical links. At higher levels only virtual links exist. These virtual links can be viewed as a tunnel implemented through lower level physical links. At higher levels the link state information of the virtual links is exchanged. A node at each level floods to its lower level the information that it obtains after the algorithm has run at that level. So the lower levels have hierarchical topology information. Each node has a hierarchical address called hierarchical ID (HID) which is defined as the sequence of addresses from the top hierarchy to the node itself. Figure 3.4 provides an example of clustering in HSR. A gateway can be reached from the root via more than one path, so gateway nodes can have more than one HID. A hierarchical address is enough to ensure delivery from anywhere in the network to the host.

In addition nodes are also partitioned into logical subnetworks and each node is assigned a logical address <subnet, host>. Each subnet has a location management server (LMS). All the nodes of that subnet register their logical address with the LMS. These LMSs advertise

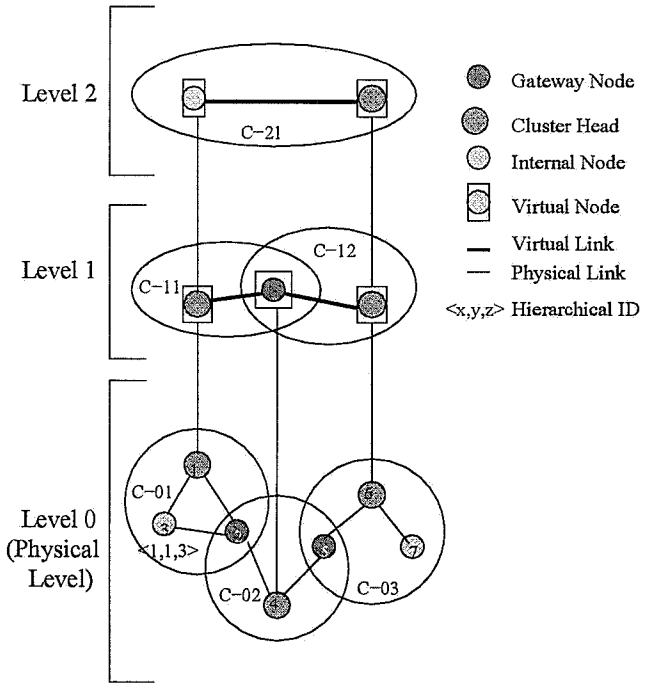


Figure 3.4 An example of clustering in HSR

their hierarchical addresses to the top levels and the upper level information is sent down to lower LMSs. The transport layer sends a packet to the network layer with the logical address of the destination. The network layer finds the hierarchical address of the destination LMS from its LMS and then sends the packet to it. The destination LMS forwards the packet to the destination. Once the source and destination know each other's hierarchical address, they can bypass the LMS and communicate directly. Since the logical address/hierarchical address is used for routing, the protocol is highly adaptable to network changes.

3.1.7 Optimized Link State Routing (OLSR)

Optimized Link State Routing (OLSR) [32] is an optimized version of link state routing protocol for ad-hoc networks. It improves performance by reducing the amount of information sent in control messages. In a pure link state protocol, all the links with neighbor nodes are

declared and flooded throughout the entire network. OLSR uses a multipoint relaying technique to efficiently and economically flood its control messages.

Each node in the network selects a set of nodes in its neighborhood, which retransmit its packets. This set of selected neighboring nodes is called the multipoint relays (MPRs) of that node (Figure 3.5). Nodes select their MPR set in such a way that the set covers (in terms of radio range) all the nodes that are 2-hops away and the number of MPRs is minimized. Each node also maintains a set of its neighbors which are called the *MPR Selectors* of the node, which is the set of nodes which have chosen this node as MPR. Every broadcast message coming from these *MPR Selectors* of a node is assumed to be retransmitted by that node.

Each node periodically transmits a Hello Message 2.3.2, which contains the list of neighboring nodes to which a bi-directional or a uni-directional link exist. These messages permit each node to learn the knowledge of its neighbors up to two hops. Using this information, each node then selects of its MPRs. These selected MPRs are indicated in future Hello messages with the link status as MPR. This information helps nodes to build their *MPR Selector* table. Each node also keeps a Neighbor table where it records the information about its one hop neighbors, the link status with these neighbors, and a list of two hop neighbors to which these one hop neighbors give access. This table is tagged by a sequence number which is incremented any time the node updates its MPR set.

Each node also broadcasts specific control messages called *Topology Control (TC)* messages. A TC messages are forwarded like usual broadcast messages in the entire network. TC message sent by a node contains the *MPR Selector* set of that node. This information is used by the nodes to build their topology table. The topology table is in turn used to compute the routing table for all the nodes in the network. In route calculation, MPRs are used as the intermediate nodes to form the route from a given node to any destination in the network.

3.1.8 Zone-based Hierarchical Link State Routing (ZHLS)

In Zone-based Hierarchical Link State Routing Protocol (ZHLS) [33], the network is divided into non-overlapping zones. Unlike other hierarchical protocols, there is no zone-head. ZHLS

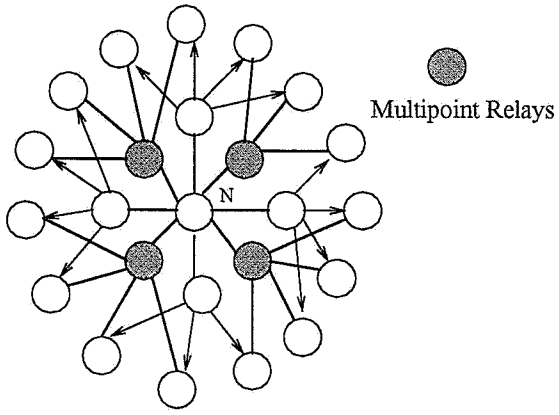


Figure 3.5 Multipoint Relays in OLSR

defines two levels of topologies - node level and zone level. A node level topology tells how nodes of a zone are connected to each other physically. A virtual link between two zones exists if at least one node of a zone is physically connected to some node of the other zone. Zone level topology tells how zones are connected together. ZHLS specifies two types of Link State Packets (LSP) - node LSP and zone LSP. A node LSP of a node contains its neighbor node information and is propagated within the zone, whereas a zone LSP contains the zone information and is propagated globally. So each node has full node connectivity knowledge about the nodes in its own zone and only zone connectivity information about other zones in the network. So, given the zone id and the node id of a destination, the packet is routed based on the zone id till it reaches the correct zone. Then, in that zone, it is routed based on the node id. Since a <zone id, node is> of the destination is sufficient for routing, the protocol is adaptable to changing topologies.

3.2 Reactive (On-Demand) Protocols

These protocols take a "lazy" approach to routing. In contrast to table-driven routing protocols, a list of all up-to-date routes is not maintained at every node. Instead the routes are created as and when required. When a node requires a route to a destination, it invokes a

route discovery mechanisms to find a path to the destination. This process is completed once a route is found, or a route could not be discovered after a maximum number of retries. Once a route has been established, it is maintained by some form of route maintenance procedure until either the destination becomes inaccessible along every path from the source, or until the route is no longer desired.

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

The Ad-hoc On-demand Distance Vector (AODV) [27] routing is a variation of the DSDV routing protocol. It minimizes the number of required broadcasts by creating routes on an on-demand basis, as opposed to maintaining a complete list of routes as in the DSDV algorithm. AODV uses the procedures of route discovery and route maintenance to discover and maintain routes to desired destinations in the network. AODV also applies a number of optimizations to improve the packet latency and network throughput. The AODV protocol is explained in detail in Chapter 8.

3.2.2 Dynamic Source Routing (DSR)

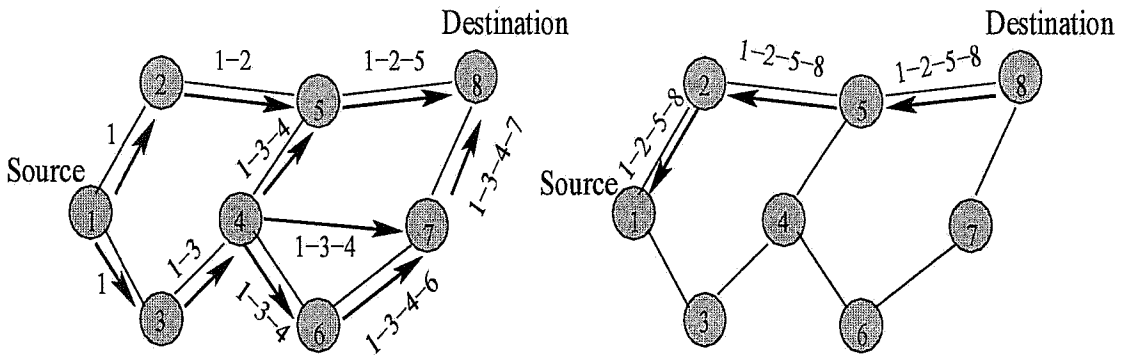
The Dynamic Source Routing (DSR) [29, 34] is an on-demand routing protocol in which the source specifies the complete route. Mobile nodes maintain route caches containing the source routes that they are aware of. A node updates entries in its route-cache as and when it learns about new routes.

The two major phases of the protocol are: Route Discovery and Route Maintenance. When the source node wants to send a packet to a destination, it looks up its route cache to determine if it already contains a route to the destination. If it finds that an unexpired route to the destination exists, then it uses this route to send the packet. But if the node does not have such a route, then it initiates a route discovery process by broadcasting a route request packet. The route request packet contains the address of the source and the destination, and a unique identification number. Each intermediate node checks whether it knows of a route to the destination. If it does not, it appends its address to the route record of the packet and forwards

the packet to its neighbors. To limit the number of route requests propagated, a node processes the route request packet only if it has not already seen the packet, and its address is not already in the route record of the packet. Nodes can also operate their network interface in promiscuous mode, disabling the interface address filtering and causing the network protocol to receive all packets that the interface overhears. These packets are scanned for useful source routes or route error messages and then discarded.

A route reply is generated when either the destination or an intermediate node with current information on a route to the destination receives the route request packet. A route request packet reaching such a node already contains, in its route record, the sequence of hops taken from the source to this node. Figure 3.6a shows the formation of the route record as the route request travels through the network. If the route reply is generated by the destination, it places the route record from the route request packet into the route reply packet. On the other hand, if the node generating a route reply is an intermediate node, it appends its cached route to the destination to the route record of the route request and puts that into the route reply. To send the route reply packet, the responding node must have a route to the source. If it has a route to the source in its route cache, it can use that. The reverse of route records can be used if symmetric links are supported. If none of these cases is true, the node can initiate a route discovery to the source and piggy back the route reply on this new route request. Figure 3.6b shows propagation of route reply containing the route record from destination to the source node.

In DSR, route maintenance is accomplished through the use of route error packets and acknowledgements. Route error packets are generated at a node when the data link layer encounters a fatal transmission problem. When a route error packet is received, the node in the route error is removed from the route cache and all routes containing that node are truncated at that point. Acknowledgement packets are used to verify the correct operation of the route links. This also includes passive acknowledgements in which a node hears the next hop forwarding the packet along the route.



(a) Building of Route Record during Route Discovery

(b) Propogation of the Route Reply with the Route Record

Figure 3.6 Route Discovery in DSR

3.2.3 Temporally Ordered Routing Algorithm (TORA)

The Temporally Ordered Routing Algorithm (TORA) [28] is an adaptive, efficient and scalable distributed routing algorithm based on the concept of link reversal. It is source initiated and provides multiple routes for any desired source-destination pair. It provides only the routing mechanism and depends on Internet MANET Encapsulation Protocol (IMEP) for other underlying functions. The key design concept of TORA is that the control messages are localized to a very small set of nodes near the occurrence of a topological change. To achieve this, nodes maintain routing information about adjacent (1-hop) nodes. The protocol has three basic functions: (i) Route creation, (ii) Route maintenance, and (iii) Route erasure. Each node maintains the following quintuple:

- Logical time of a link failure
- The unique ID of the node that defined the new reference level
- A reflection indicator bit
- A propagation ordering parameter or "height"

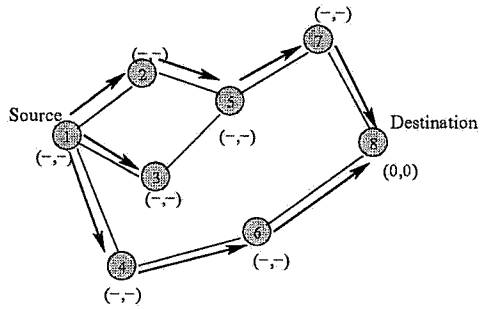
- The unique ID of the node

The first three elements collectively represent the reference level. A new reference level is defined each time a node loses its last downstream link due to a link failure. The last two values define the difference with respect to the reference level.

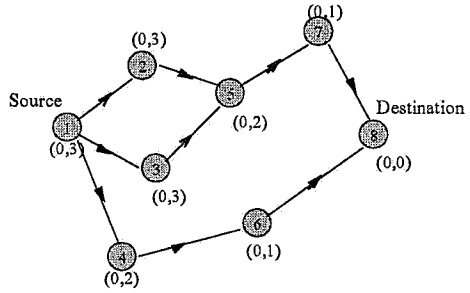
Route creation is done using QRY and UPD packets. The route creation algorithm starts with the height of the destination set to 0, and all other node's height set to NULL (i.e. undefined). The source broadcasts a QRY packet with the destination node's ID in it. A node with a non-NULL height responds with a UPD packet that has its height in it. A node receiving a UPD packet sets its height to one more than that of the node that generated the UPD packet. A node with higher height is considered upstream and a node with lower height downstream. In this way a directed acyclic graph (DAG) is constructed from the source to the destination. Figure 3.7 illustrates the route creation process in TORA. In Figure 3.7a, node 5 does not propagate QRY from node 3 as it has already seen and propagated the QRY message from node 2. In Figure 3.7b, the source node may have received a UPD each from node 2 and 3 but, since node 4 gives it lesser height, it retains that height.

When a node moves, the DAG route is broken and route maintenance is needed to reestablish a DAG for the same destination. When the last downstream link of a node fails, it generates a new reference level. This results in the propagation of that reference level by neighboring nodes, as shown in Figure 3.8. Links are reversed to reflect the changes in adapting to the new reference level. This has the same effect as reversing the direction of one or more links when a node has no downstream links. In the route erasure phase, TORA floods a broadcast clear packet (CLR) throughout the network to erase invalid routes.

In TORA there is a potential for oscillations to occur, especially when multiple sets of co-ordinating nodes are concurrently detecting partitions, erasing routes, and building new routes based on each other. Because TORA uses inter-nodal coordination, its instability problem is similar to the "count-to-infinity" [12] problem in distance-vector routing protocols, except that such oscillations are temporary and route convergence will ultimately occur.



(a) Propogation of QRY message through the network



(b) Height of each node updated as a result of UPD message

Figure 3.7 Route Creation in TORA. (Numbers in braces are (reference level, height))

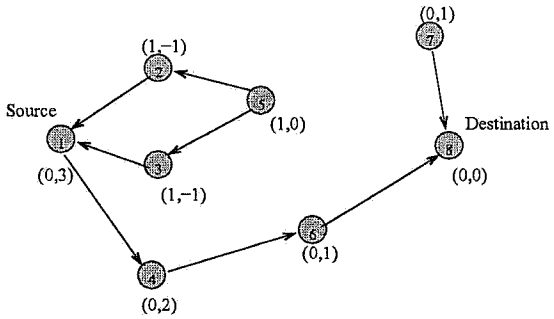


Figure 3.8 Re-establishing route on failure of link 5-7. The new reference level is node 5.

3.2.4 Associativity Based Routing (ABR)

Associativity Based Routing (ABR) [35, 36] is a totally different routing approach for ad-hoc networks. ABR defines a new metric for routing known as the degree of association stability. It is free from loops, deadlocks and packet duplications. In ABR, a route is selected based on the degree of association stability of mobile nodes. Each node periodically generates a beacon to signify its existence. When a neighbor node receives a beacon, it updates its associativity table. For every beacon received, a node increments its associativity tick with respect to the node from which it received the beacon. Association stability means connection stability of one node with respect to another node over time and space. A high value of the associativity tick with respect to a node indicates low node mobility, while a low value may indicate high node mobility. Associativity ticks are reset when the neighbors of a node or the node itself move out of proximity. The fundamental objective of ARB is to find longer-lived routes. The three phases of ARB are: (i) Route discovery, (ii) Route re-construction (RRC) and (iii) Route deletion.

The route discovery phase is a broadcast query and await-reply (BQ-REPLY) cycle. The source node broadcasts a BQ message in search of nodes that have a route to the destination. A node does not forward a BQ request more than once. On receiving a BQ message, an intermediate node appends its address and its associativity ticks with its neighbors to the query packet. The next succeeding node erases its upstream node neighbors associativity tick entries and retains only the entries concerned with itself and its upstream node. Each packet arriving at the destination will contain the associativity ticks of the nodes along the route from source to the destination. The destination can now select the best route by examining the associativity ticks along each of the paths. If multiple paths have the same overall degree of association stability, the route with the minimum number of hops is selected. Once a path has been chosen, the destination sends a REPLY packet back to the source along this path. The nodes on the path that the REPLY packet follows mark their routes as valid. All other routes remain inactive, thus avoiding the chance of duplicate packets arriving at the destination.

The RRC phase consists of partial route discovery, invalid route erasure, valid route updates and new route discovery depending on which node(s) along the route move. Source node movement results in a new BQ-REPLY process because the routing protocol is source-initiated. A route notification (RN) message is issued to erase the route entries associated with downstream nodes. When the destination moves, the destination's immediate upstream node erases its route. A localized query process LQ[H], where H refers to the hop count from the upstream node to the destination, is initiated to determine if the node is still reachable. If the destination receives the LQ packet, it selects the best partial route and sends a REPLY packet; otherwise, the initiating node times out and backtracks to the next upstream node. An RN message is sent to the next upstream node to erase the valid route and inform this node that it should invoke the LQ[H] process. If this process results in backtracking more than halfway to the source, the LQ process is discontinued and the source initiates a new BQ process.

When a discovered route is no longer needed, the source node initiates a route delete (RD) broadcast. All nodes along the route delete the route entry from their routing table. The RD message is propagated by a full broadcast, as opposed to a directed broadcast, because the source node may not be aware of any route node changes that occurred during RRCs.

3.2.5 Signal Stability Routing (SSR)

The Signal Stability Based Adaptive Routing protocol (SSR) [30] is an on-demand routing protocol that selects routes based on the signal strength between nodes and a node's location stability. This route selection criteria has the effect of choosing routes that have "stronger" connectivity. SSR comprises of two cooperative protocols: the Dynamic Routing Protocol (DRP) and the Static Routing Protocol (SRP).

The DRP maintains the Signal Stability Table (SST) and Routing Table (RT). The SST stores the signal strength of neighboring nodes obtained by periodic beacons from the link layer of each neighboring node. The signal strength is either recorded as a strong or weak channel. All transmissions are received by DRP and processed. After updating the appropriate table entries, the DRP passes the packet to the SRP.

The SRP passes the packet up the stack if the node is the intended receiver. If not, it looks up the destination in the RT table and forwards the packet. If there is no entry for the destination in the RT, it initiates a route-search process to find a route. Route-request packets are forwarded to the next hop only if they are received over strong channels and have not been previously processed (to avoid looping). The destination chooses the first arriving route-search packet to send back, as it is highly likely that the packet arrived over the shortest and/or least congested path. The DRP reverses the selected route and sends a route-reply message back to the initiator of route-request. The DRP of the nodes along the path update their RTs accordingly.

Route-search packets arriving at the destination have necessarily arrived on the path of strongest signal stability because the packets arriving over a weak channel are dropped at intermediate nodes. If the source times out before receiving a reply, it changes the PREF (preference) field in the header to indicate that weak channels are acceptable, since these may be the only links over which the packet can be propagated.

When a link failure is detected within the network, the intermediate nodes send an error message to the source indicating which channel has failed. The source then sends an erase message to notify all nodes of the broken link and initiates a new route-search process to find a new path to the destination.

3.3 Hybrid Routing Protocol

Hybrid protocols employ both reactive and proactive schemes to route packets within an ad-hoc network. Such protocols are designed to have the advantages of both the reactive and proactive routing techniques.

3.3.1 Zone Routing Protocol (ZRP)

The Zone Routing Protocol (ZRP) [31, 37] is a hybrid of reactive and proactive routing protocols. It divides the network into several routing zones, where a zone is a local region defined by a single parameter called *zone radius* measured in hops. ZRP uses two different

routing protocols, for routing within a zone and routing between zones. Nodes proactively maintain routing information for nodes within their zone and reactively discover routes for nodes outside their zones. The two routing mechanisms are referred to as Intrazone Routing Protocol (IARP) and Interzone Routing Protocol (IERP) respectively.

IARP is the proactive component of ZRP. It operates inside the routing zone and learns the routes to all the nodes within the zone. The particular protocol to be used is not specified and can be any of the proactive routing protocol. A change in topology propagates information only to nodes within affected zones.

IERP is the reactive component of ZRP and is used for finding routes between different zones. When a node needs to send packets, it first checks to see if the destination is in the same zone. If so, the path to the destination is known (through IARP), and is used to deliver the packet. If the destination is not within the the source's routing zone, the source sends a route query to all the peripheral nodes. *Peripheral Nodes* are the nodes whose minimum distance from the source node is the zone radius. Peripheral nodes in turn forwards the request if the destination node is not within their routing zone. This procedure is repeated until the route request reaches the zone containing the destination node. Either the destination node or some other node within the zone of destination node replies to the query.

The route discovery in ZRP can be made much more efficient in terms of resource usage, though at the expense of longer latency. Instead of querying simultaneously all the peripheral nodes, these nodes can be queried either sequentially, one-by-one or in groups. Thus there is a tradeoff between the cost and latency of the ZRP routing protocol.

CHAPTER 4

The Destination Sequenced Distance Vector (DSDV) Protocol

4.1 Introduction

The Destination-Sequenced Distance Vector (DSDV) [22] is an adaptation of the conventional distance-vector routing protocol to the ad-hoc networks. It is based on the classical Distributed Bellman-Ford (DBF) algorithm [13]. DSDV enables dynamic, self-starting, multi-hop routing between participating mobile nodes wishing to establish and maintain an ad-hoc network. It provides quick route convergence in an ad-hoc network with dynamically changing topology. DSDV maintains routes to every node in the network, thus offering a route immediately whenever it is needed. This avoids any latency associated with the route discovery procedure. DSDV guarantees loop free routes using a novel scheme of destination provided sequence numbering of routing update information.

Overall the DSDV protocol has the following properties:

- Loop-free at all instants
- Dynamic, multi-hop, self-starting
- Low memory requirements
- Quick convergence via triggered updates
- Immediate route availability for all destinations

- Fast processing time
- Minimal route trashing
- Reasonable network load.

This chapter provides detailed specifications of the DSDV protocol along with its functionalities.

4.2 Overview of DSDV Protocol

DSDV is a proactive ad-hoc routing protocol which is based on the idea of distance-vector routing. It offers freedom from routing loops using a scheme of numbering route table entries by sequence numbers provided by the destination.

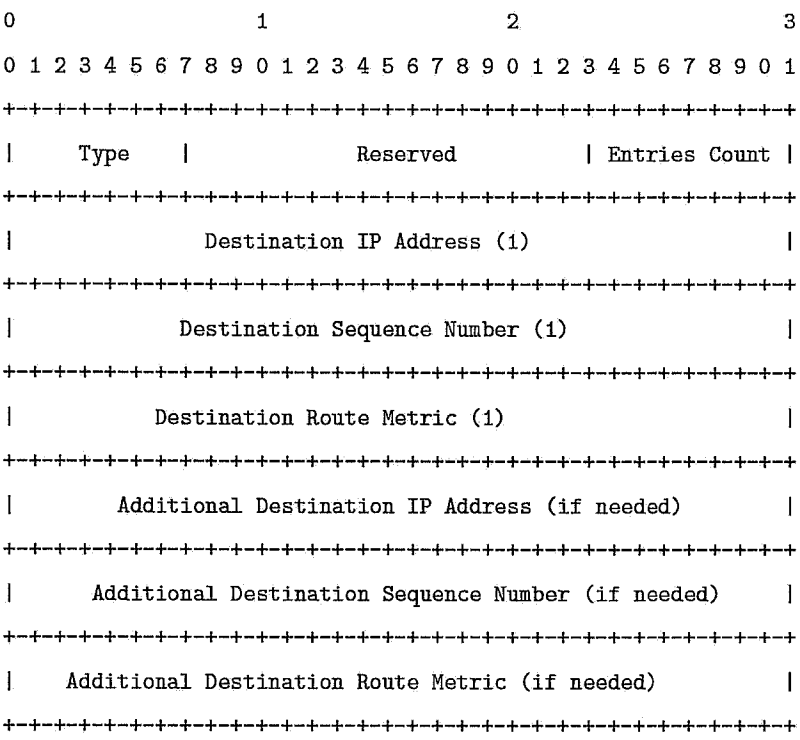
Each mobile node in the network maintains a routing table in which all of the possible destinations within the network and the number of hops to each destination are recorded. Each routing table entry is marked with a sequence number that is originally generated by the destination node. The sequence numbers enable mobile nodes to distinguish stale routes from the new ones, thereby avoiding the formation of routing loops. To maintain the consistency of the routing tables, each node periodically transmits routing updates. Routing updates are also transmitted immediately on receipt of significant new routing information. These are called periodic and triggered routing updates, respectively. A *ROUTE-UPDATE* message is used to send out route advertisements.

DSDV employs two types of route update packets to reduce the amount of information sent out in these update messages. The first known as "full dump", carries all the available routing information. The second type, known as "incremental dump", carries only the routing information which has changed since the last full dump. Full dump packets are transmitted infrequently during periods of occasional movements. DSDV also keeps data about the length of time between the arrival of the first and the arrival of the best route, for each particular destination. This is used to delay advertising routes which are about to change soon. DSDV

expires route table entries if routing updates are not received within the *ROUTE_TIMEOUT* period.

4.3 Message Formats

4.3.1 Route Update (ROUTE-UPDATE) Message Format



The format of Route Update message is illustrated above. It contains the following fields:

```
Type              1
Reserve           Sent as 0; ignored on reception.
Entries Count     The number of destination entries included in the
                  update message; MUST be at least 1.
Destination IP Address
                  The IP address of the destination being advertised
                  by the node.
Destination Sequence Number
```

```
                The sequence number in the route table for the
                destination listed in the previous Destination IP
                Address field.
Destination Route Metric
                The hop count in the route table for the destination
                listed in the Destination IP Address field.
```

The Route Update message is used to send out periodic and triggered routing updates by every node in the ad-hoc network running DSDV. When a node boots up, the Route Update message will have only a single entry corresponding to the local node.

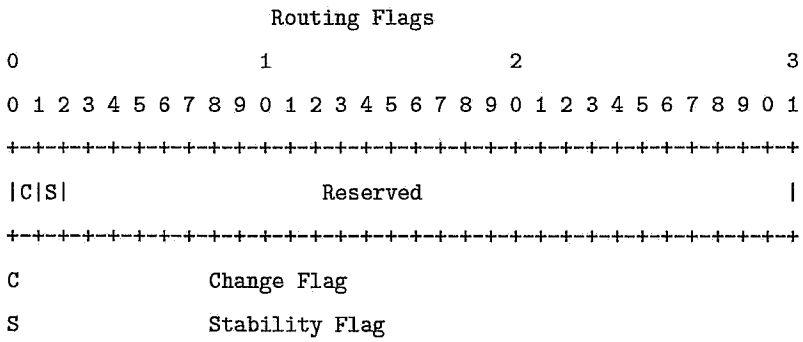
4.4 DSDV Operation

4.4.1 Route Table Entry

DSDV keeps routing information in a route table. The route table contains an entry for every reachable destination in the network. Every route table entry contains following fields:

- Destination IP Address
- Destination Sequence Number
- Next Hop (next node on the path to the destination)
- Route Metric (number of hops needed to reach destination)
- Lifetime (expiration or deletion time of this route)
- Routing Flags

When a node comes up, it initializes its route table with an entry for itself. The destination IP address and next hop are set to the node's local IP address. The sequence number is set to the locally maintained value of destination sequence number. The route metric (hop count) is set to 0. The Lifetime is initialized to current time plus *ROUTE_TIMEOUT* value. The structure of routing flags field is as shown below:



Flag C is used when a node wants to make a decision about doing a full dump. Flag S indicates the stability of a route. If set to 0, the route is not considered very stable and the broadcast of any changes to the route is delayed to allow it to settle down.

4.4.2 Maintaining Sequence Numbers

The DSDV protocol uses sequence numbers to guarantee loop free routes. Every destination is responsible for maintaining a sequence number for itself, called a "destination sequence number." A destination increments its sequence number every time it sends out a periodic routing update message. The sequence numbers generated by the destination nodes are defined to be even numbers. So, when doing periodic broadcast, a local node increments its sequence number by 2. This is to distinguish this sequence number from the sequence number which might be generated for this node by some other mobile node in the network. A node does not increment its sequence number while sending out triggered update messages.

Every route table entry at every node MUST include the latest information available about the sequence number for the destination IP address stored in that entry. A node updates the sequence numbers stored in its routing table entry whenever it receives new routing update messages. The received sequence number for a destination is compared with the sequence number value stored for that destination in the node's route table. The received information supersedes already stored information if it has a higher sequence number or a lower route metric in case the two sequence numbers are same.

The only scenario in which a node may generate a new sequence number for a destination besides itself is when it detects a broken link with one of its previous neighbors. In such a circumstance, the node increments the sequence number for the broken node by 1. This makes the generated sequence number an odd value. This is done so that any new information generated by the destination node will have the next higher even value and will then supersede this information.

In effect, a node may change the sequence number in the routing table entry of a destination node only if:

- It is itself the destination node and is about to send a periodic route update message.
- It receives a periodic update message with the new information about the sequence number for the destination node.
- The route towards the destination node breaks or expires.

4.4.3 Periodic Route Advertisement

The DSDV protocol requires each mobile node to periodically advertise the content of its route table to each of its current neighbors. The advertisement must be made often enough to ensure that all the mobile nodes have a consistent view of the network topology almost always. This is even more important for networks which have a very high rate of topology changes. A route update packet sent out by a node indicates which mobile hosts are accessible from that node, the number of hops necessary to reach those mobile hosts, and the sequence numbers of those hosts as known by the node.

Periodic route advertisement is done through the *ROUTE-UPDATE* message. A node generates route update messages using either full dump or incremental dump (see Section 4.4.7). A node scans through its route table to generate a *ROUTE-UPDATE* message. This message is sent out on the IP limited broadcast address "255.255.255.255", so that all the one-hop neighbors receive this message. When a node reboots, its route table does not contain information about

any other node in the network. In such a scenario, the first periodic update message sent out by the node will only contain a single entry corresponding to itself.

The node also sets a timer for *PERIODIC_UPDATE_INTERVAL* time. The next periodic update will be sent when this timer expires.

4.4.4 Triggered Route Advertisement

Besides advertising route tables through periodic update messages, a node in DSDV also sends out a route update message immediately after its own route table changes significantly. This is called a *triggered route advertisement.* These updates are used to propagate the route information as quickly as possible when there is any topology change within the network. A *ROUTE-UPDATE* message is used to broadcast triggered route advertisement.

Like periodic route advertisements, the route update message generated by triggered advertisement can use either a full dump packet or an incremental dump packet (Section 4.4.7). A node might delay the advertisement of some of these triggered update messages in order to reduce the number of rebroadcasts of possible route entries that normally arrive with the same sequence number (Section 4.4.8).

DSDV uses the following rules to decide which route table changes should be considered significant enough to trigger an update message. A triggered update is sent out if any of the following listed conditions is true:

- If the node receives an update message containing an entry for a destination with a higher sequence number than that stored in the nodes route table for that destination and
  - The received entry has route metric set to *INFINITY*,
    Or
  - The route metric of the received entry is different from the value stored in the route table (also see Section 4.4.8),
    Or

- If the two route metrics are the same but the node from which the entry has been received is different from the next hop value stored in the route table

- If the sequence number contained in the received entry is smaller than that stored in the route table and the route metric of the received entry is set to *INFINITY*.

- If the received and stored sequence numbers are the same, but the received entry has a smaller route metric.

- If the node receives an entry which does not exist in the route table.

- If an existing route breaks or expires.

A change in the sequence number only is not considered to be a significant change, so no update message is sent out for such a change. Sequence number changes are broadcasted when the next periodic advertisement is done.

4.4.5 Processing Route Update Messages

A node does not distinguish between a periodic route update message and a triggered route update message while processing. Upon receiving the route update message, a node scans the list of entries contained in it. For every non-local destination entry in this list, it looks for a matching destination entry in its route table. If no such entry exists, it creates a new entry in its route table corresponding to this destination based on the information received.

If the node already has an entry for the received destination in its route table, it is modified only if:

- The received entry has a higher sequence number than the one stored in the route table

- The sequence numbers are the same but the received entry has a smaller route metric.

Whenever a new entry is created or an existing entry is updated, the next hop in the route entry is assigned to be the node from which the update packet was received. The sequence

number and the route metric of the route table entry are set to the their corresponding received values. The lifetime of the entry is set to the current time plus *ROUTE_TIMEOUT* value.

A node uses the rules mentioned in Section 4.4.4 to make a decision about sending out a triggered route update message.

4.4.6 Handling Broken Links

Mobile hosts cause broken links as they move from place to place. There are two ways a broken link may be detected in DSDV:

1. A broken link may be detected by a layer 2 protocol. For example in IEEE 802.11, the absence of a link layer ACK or a failure to get a CTS after sending an RTS, even after the maximum number of retransmission attempts, indicates a broken link to the neighboring node.

2. If no broadcast is received by the *ROUTE_TIMEOUT* time from a former neighboring node, then the link to that node is considered as broken.

In the second case, the lifetime field of the route table entry is used to detect the broken links. Whenever a route entry expires, the route to that node is considered to be broken.

When a node detects a broken link to a neighboring node, it increments the node's sequence number in its route table entry by 1. It also sets the route metric for that node to *INFINITY*. The Lifetime value for the node is set to current time plus *DELETE_PERIOD*. Route table entries for other destination nodes using this neighboring node as next hop are also updated to have an incremented sequence number, *INFINITY* route metric, and current time plus *DELETE_PERIOD* as their lifetime.

The node then sends out a triggered route update message containing updated route table information as mentioned in Section 4.4.4 .

4.4.7 Full Dump and Incremental Dump

To reduce the amount of routing overhead generated by route update messages, DSDV uses two types of packets to send out these updates: *full dump* and *incremental dump*.

A *full dump* route update packet contains an entry for every route table entry and can require multiple network protocol data units (NPDUs). The NPDU is the maximum size packet which can be transmitted on a communication channel without fragmentation. During periods of occasional movement full dump packets should be transmitted infrequently. Smaller *incremental dump* packets are used to broadcast only those route table entries which have changed since the last full dump. Incremental dump packets are expected to fit into a standard size NPDU, thereby decreasing the amount of traffic generated. In general a node uses incremental dump to sent out route updates.

A node does a full dump only in one of the following scenarios:

- If the node is sending out its first route update message.
- If the number of route table entries which have changed since the last full dump is greater than or equal to *FULL_DUMP_FACTOR* times the total number of entries in the route table.
- If more than the *MAX_FULL_DUMP_TIMEOUT* period has passed since the last full dump.

The Routing Flags field in the route table entry is used to keep track of entries which have changed since the last full dump. The first bit of the routing flags field is designated as "Change Flag Bit." When doing full dump, this flag bit is set/reset to 0 for all the route table entries. Whenever an entry is updated in the route table, the change flag bit for that entry is set to 1.

4.4.8 Damping Fluctuations

DSDV employs a mechanism to damp fluctuations in the route tables. In an environment where many independent nodes transmit routing information asynchronously, fluctuations could

develop. For example a node could receive two routes to the same destination with the same sequence number, however, the one with the worse metric always arrives first. This would result in the continuous outbursts of route update messages and fluctuations of the route tables.

DSDV uses the idea of *settling time* to damp fluctuations of route tables. The *settling time* for a destination is defined as the time duration until the route to that destination becomes stable. The idea is to predict a settling time value for each destination node and use that to delay advertising changes to the route table entry of that destination node. This requires each mobile node to keep a history of the weighted average time that routes to a particular destination fluctuate until the route with the best metric is received.

DSDV stores settling time data in a Settling Time Table with the following fields, keyed by the first field:

- Destination IP Address
- Last Settling Time
- Average Settling Time
- Last Receive Time

The Last Receive Time is the last timestamp when the route table entry for the Destination IP Address was changed. This is needed to calculate the settling time for a destination node. The Last Receive Time for the destination is set to the current system time under the following circumstances:

- When an entry with a higher sequence number is received.
- When an entry with the same sequence number but smaller route metric is received.
- When the node receives an entry for a destination which is not in its route table.

In the last case, the node creates a new entry in its settling time table for that destination, initializing both the Last Settling Time and Average Settling Time to 0.

The settling time is computed by maintaining a running, weighted average over the most recent updates of the routes, for each destination. Whenever a node receives an entry for a destination containing a smaller metric with the same destination sequence number as the one stored in the the route table entry for that destination, it computes the Last Settling Time and Average Settling Time values as follows:

```
If(Average Settling Time == 0 && Last Settling Time == 0)
  Last Settling Time = (Current Time - Last Recv Time)
else
  if(Average Settling Time == 0)
    Average Settling Time = Last Settling Time
    Last Settling Time = (Current Time - Last Recv Time)
  else
    Average Settling Time = (Average Settling Time + Last Settling Time)/2
    Last Settling Time = (Current Time - Last Recv Time)
  end
end
```

Whenever a node receives an entry for a destination with a higher sequence number than the one stored in the route table entry, it delays broadcasting the received information for that destination if the Last Settling Time for that destination is non-zero. This is called *delayed triggered broadcast.* The following formula is used to compute the value of *DELAYED_UPDATE_INTERVAL*, which is the time interval for which the route update for the intended destination must be delayed:

```
if(Last Settling Time == 0)
  broadcast immediately
else
  if( Average Settling Time == 0)
    DELAYED_UPDATE_INTERVAL = Last Settling Time
  else
    DELAYED_UPDATE_INTERVAL = (STL_AVG_FACTOR * Last Settling Time)+ \
      ((1 -STL_AVG_FACTOR)*Average Settling Time)
  end
```

```
end
```

In reality the broadcast of route update message is delayed by *STL_TIME_FACTOR* times the *DELAYED_UPDATE_INTERVAL* to accommodate delays in the network.

In addition, DSDV defines a *MAX_STL_TIME* for every destination, which is the time duration a route has to remain stable, before it is counted as truly stable. The second bit of the Routing Flags in the route table entry is used to indicate the stability of that route. When a route is first created, it is marked as stable (set "Stable Flag bit" to 1). At the time when settling time is being calculated, if the Last Settling Time is less than the *MAX_STL_TIME*, then the Stable Flag bit in the route table entry for that destination is set to 0. If the Last Settling Time is greater than the *MAX_STL_TIME*, then this value is not used in the computation of Average Settling Time and the Last Settling Time is set to *MAX_STL_TIME*. In this case the Stable Flag bit is reset to 1. Whenever an entry with a higher sequence number is received for this destination, its Stable Flag bit is set to 1 and the broadcast is delayed only if the old value of Stable Flag bit was set to 0.

The *MAX_STL_TIME* can be calculated adaptively for each destination based on the history of previous settling time data for that destination, instead of using a statically configured value for all the destinations.

To implement damping fluctuations, a node maintains two route tables. The first one called *Forward Route Table* is used to forward all the data traffic received at the node. The second route table called *Broadcast Route Table* is used to build and send out route update messages. All the received routing information is applied to both the tables except when a delayed broadcast is issued. In such a situation, routing updates are applied only to the Forward Route Table. The two tables are synchronized when the delayed broadcast is actually sent out.

4.4.9 Expiring and Deleting Routes

Every route table entry in DSDV has a lifetime field associated with it. A node sets the lifetime value of a route table entry to *ROUTE_TIMEOUT* when that entry is created or

updated. If a node keeps receiving periodic updates for the destination entries in its route table, the lifetime values of these entries are overwritten before they expire. However, if such periodic updates are not received (due to broken links), then the route table entries expire. In such a situation a node proceeds as stated in Section 4.4.6. A node resets the lifetime value for its own entry to current time plus *ROUTE_TIMEOUT* periodically. A periodic timer is used to detect expired route table entries. The lifetime of the local node's route table entry is reset every time this timer expires.

An expired route table entry is not deleted from the route table for *DELETE_PERIOD* amount of time. This is done to ensure that the broken link information is sent out enough number of times so that every mobile node in the network gets to know about it.

4.4.10 Actions After Reboot

A node participating in the ad-hoc network must take certain actions after reboot as it might lose all sequence number records for all the destinations, including its own sequence number. However, there may be neighboring nodes that are using this node as an active next hop. This can potentially create routing loops. To prevent this possibility, each node on reboot waits for *DELETE_PERIOD* amount of time. During this time, the node does not transmit any periodic or triggered update messages. If the node receives route update messages from other nodes, it creates route table entries using the received information. The node also updates its own sequence number whenever it receives an update message containing an entry for its own IP address. Thus, by the time a node comes out of its reboot waiting phase and becomes an active router again, none of the nodes in the network will have an unexpired entry for this node. If the final value of the local sequence number learnt during the reboot waiting phase is an even number, the node uses that to send out its first periodic update message. Otherwise, if the sequence number value is odd, the local sequence number is set to the next higher even number and that is used to send out the update messages.

4.4.11 DSDV Timers

DSDV requires four different timers for its correct operation. These are *REBOOT_TIMER*, *PERIODIC_UPDATE_TIMER*, *DELAYED_UPDATE_TIMER*, and *BROKEN_LINK_TIMER*. The first timer is set immediately after the node reboots (see Section 4.4.10). The *PERIODIC_UPDATE_TIMER* is used to send out route update messages periodically. In case a node wants to schedule a delayed triggered advertisement, it uses *DELAYED_UPDATE_TIMER*. The *BROKEN_LINK_TIMER* is a periodic timer which is used to detect broken links and to delete expired entries from the route table and settling time table.

4.4.12 Operation at Layer 2

DSDV routing protocol can also be implemented at layer 2 [13]. When DSDV is operated at layer 2, Media Access Control (MAC) addresses [12] are stored in the route tables. Using MAC addresses for routing tables does not introduce a new requirement. The difficulty is that the Layer 3 network protocols provide communication based on the network addresses, and a way must be provided to resolve these Layer 3 addresses into MAC addresses. Otherwise, a multiplicity of different address resolution mechanisms would be needed and a corresponding loss of bandwidth in the wireless medium would be observed whenever the resolution mechanisms are utilized.

The solution is to include Layer 3 (network layer) protocol information along with the Layer 2 information while operating DSDV at Layer 2. Each destination host would advertise which Layer 3 protocol(s) it supports, and each mobile host advertising reachability to that destination would include along, with the advertisement, the information about the Layer 3 protocols supported at that destination. This information would only have to be transmitted when it changes, which occurs rarely. Changes would be transmitted as part of each incremental dump. Since each mobile host could support several Layer 3 protocols, this list would have to be variable in length.

4.4.13 Extending Base Station Coverage

Mobile computers will frequently be used in conjunction with base stations, which allow them to exchange data with other computers connected to the wired network. By participating in the DSDV protocol, base stations can extend their coverage beyond the range imposed by their radio transmitters. When a base station participates in DSDV, it is shown as a default route in the update messages transmitted by the mobile stations. In this way, mobile stations within range of a base station can cooperate to effectively extend the range of the base station to serve other stations outside the range of the base station, as long as those other mobile stations are close to some other mobile station that is within range.

4.5 Configuration Parameters

This section provides default values for some important parameters associated with the DSDV protocol operation.

Parameter Name	Value
PERIODIC_UPDATE_TIMER_INTERVAL	2000 ms
BROKEN_LINK_TIMER_INTERVAL	1000 ms
ROUTE_TIMEOUT	5*PERIODIC_UPDATE_TIMER_INTERVAL
DELETE_PERIOD	5*PERIODIC_UPDATE_TIMER_INTERVAL
MAX_FULL_DUMP_TIMEOUT	5*PERIODIC_UPDATE_TIMER_INTERVAL
FULL_DUMP_FACTOR	0.6
STL_AVG_FACTOR	0.7
STL_TIME_FACTOR 2	
MAX_STL_TIME	PERIODIC_UPDATE_INTERVAL/4

A particular mobile node may wish to change some of these parameters, in particular *PERIODIC_UPDATE_TIMER_INTERVAL*, *FULL_DUMP_FACTOR* and *STL_TIME_FACTOR*. Choice of these parameters may effect the performance of the DSDV routing protocol.

4.6 Proof of Loop-free Property

In this section we show that destination sequence numbering guarantees freedom from loops.

Let us consider a collection of N mobile nodes in steady-state i.e. route tables of all nodes have already converged to the actual shortest paths. At this instant the next node indicators to each destination induce a tree rooted at that destination. Thus, routing tables of all nodes in the network can be collectively visualized as forming N trees, one rooted at each destination. Let us focus on a specific destination d for the purpose of this discussion. p_i^d is defined as the next-hop for destination d at node i. s_i^d is defined as the sequence number value for node d stored at node i.

Potentially a loop may form each time node i changes its next-hop. This can happen in two cases. First, when node i detects that the link to its next-hop is broken, the node sets the p_i^d to *nil*. Clearly, this action cannot form a loop involving node i. The second scenario occurs when node i receives, from one of its neighbors k, a route to d and

- the new route contains a higher sequence number, i.e., $s_k^d > s_i^d$,

  Or

- the sequence number s_k^d is same as s_i^d, but the received route has a shorter route metric.

In the first case, by choosing k as its next hop, node i cannot form a loop. This can be deduced as follows. A node i propagates the sequence number s_i^d to its neighbor only after receiving it from its current next-hop. Therefore, at all times the sequence number value stored at the next-hop is always greater or equal to the value stored at i. Starting from node i, if we follow the chain of next-hop pointers, the sequence number values stored at visited nodes would form a nondecreasing sequence. Now suppose node i forms a loop by choosing k as its next-hop. This would imply node i lies both before and after k in the chain. Since it lies after k, we must have $s_k^d \leq s_i^d$. But this contradicts our initial assumption that $s_k^d > s_i^d$. Hence, loop-formation cannot occur if nodes use newer sequence numbers to pick routes.

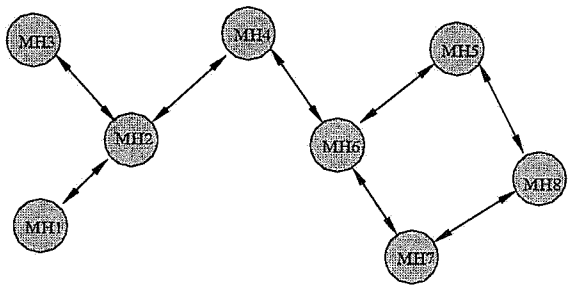


Figure 4.1 An example of Ad-hoc Network

The loop-free property holds in the second scenario due to the theorem proved in [38], which states that in presence of static or decreasing link weights then distance-vector algorithm always maintain loop-free paths.

4.7 Example of DSDV in Operation

Consider node MH_4 in the ad-hoc network shown in Figure 4.1 . Table 4.1 shows a possible structure of the forwarding table which is maintained at MH_4. The address of each Mobile Host is represented as MH_i and sequence numbers are denoted as $SNNN\_MH_i$, where $SNNN$ is a sequence number value. Also, suppose that there are entries for all other Mobile Hosts, with sequence numbers $SNNN\_MH_i$ at node MH_4. The install time field helps determine when to delete stale routes. The install time field stores the absolute time when the route table entry was created first. This field is updated every time the route table entry changes. The install field corresponds to the *Lifetime* field mentioned in Section 4.4.1

From Table 4.1, we can see that all of the nodes become available to MH_4 at about the same time because, for most of them, their install time is about the same. We can also surmise that none of the links were broken, because all of the sequence number fields have even digits in their units place. Table 4.2 shows the structure of advertised route update message of node MH_4.

Now suppose that MH_1 moves into the general vicinity of MH_5 and MH_7, and away from others (especially MH_2). The new internal forwarding table at MH_4 might then appear as

Table 4.1 Structure of the MH_4 forwarding table

Destination	Next Hop	Metric	Sequence Number	Install
MH_1	MH_2	2	S406_MH_1	T001_MH_4
MH_2	MH_2	1	S128_MH_2	T001_MH_4
MH_3	MH_2	2	S564_MH_3	T001_MH_4
MH_4	MH_4	0	S710_MH_4	T001_MH_4
MH_5	MH_6	2	S392_MH_5	T002_MH_4
MH_6	MH_6	1	S076_MH_6	T001_MH_4
MH_7	MH_6	2	S128_MH_7	T002_MH_2
MH_8	MH_6	3	S050_MH_8	T002_MH_2

Table 4.2 Advertised Route Table by MH_4

Destination	Metric	Sequence Number
MH_1	2	S406_MH_1
MH_2	1	S128_MH_2
MH_3	2	S564_MH_3
MH_4	0	S710_MH_4
MH_5	2	S392_MH_5
MH_6	1	S076_MH_6
MH_7	2	S128_MH_7
MH_8	3	S050_MH_8

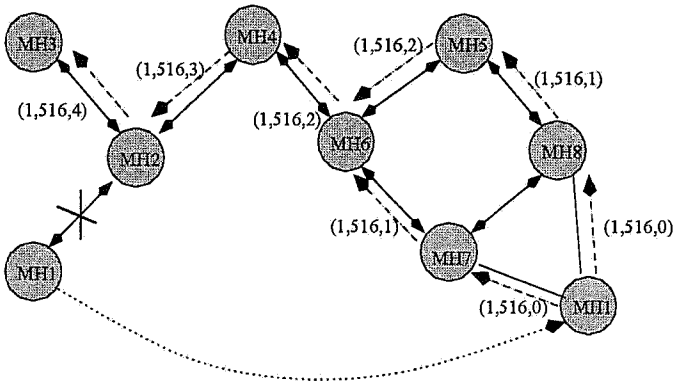
Table 4.3 MH_4 forwarding table (updated)

Destination	Next Hop	Metric	Sequence Number	Install
MH_1	MH_6	3	S516_MH_1	T810_MH_4
MH_2	MH_2	1	S238_MH_2	T001_MH_4
MH_3	MH_2	2	S674_MH_3	T001_MH_4
MH_4	MH_4	0	S820_MH_4	T001_MH_4
MH_5	MH_6	2	S502_MH_5	T002_MH_4
MH_6	MH_6	1	S186_MH_6	T001_MH_4
MH_7	MH_6	2	S238_MH_7	T002_MH_4
MH_8	MH_6	3	S160_MH_8	T002_MH_4

shown in Table 4.3. Only the entry for MH_1 shows a new metric, but in the intervening time, many new sequence number entries have been received. The first entry thus must be advertised in subsequent incremental routing information until the next full update occurs.

When MH_1 moved into the vicinity of MH_5 and MH_6, it triggered an immediate incremental routing information update which was then broadcast to MH_6. MH_6, having determined that significant new routing information had been received, also triggered an immediate update which carried along the new routing information for MH_1. MH_4, upon receiving this information, would then broadcast it at every periodic update interval until the next full dump. Figure 4.1 illustrates the logical propagation of a route entry from node MH_1 to other nodes in the network. MH_1 attaches a new sequence number to the route advertisement. Only the relevant information for node MH_1 has been shown in Figure 4.1. The actual route update messages broadcasted by each node would contain its entire route table, not just the information for MH_1.

The incremental advertised routing update at MH_4 would have the form as shown in Table 4.4. The information for MH_4 comes first since it is doing the advertisement. The information for MH_1 comes next, as MH_1 is the only one that has any significant route changes affecting it.



(x,y,z) represents (destination address, sequence number, route metric (hop count))

Figure 4.2 Mobility in an Ad-hoc Network

Table 4.4 MH_4 advertised table (updated)

Destination	Metric	Sequence Number
MH_4	0	S820_MH_4
MH_1	3	S516_MH_1
MH_2	1	S238_MH_2
Mh_3	2	S674_MH_3
MH_5	2	S502_MH_5
MH_6	1	S186_MH_6
MH_7	2	S238_MH_7
MH_8	3	S160_MH_8

CHAPTER 5

Implementation of DSDV

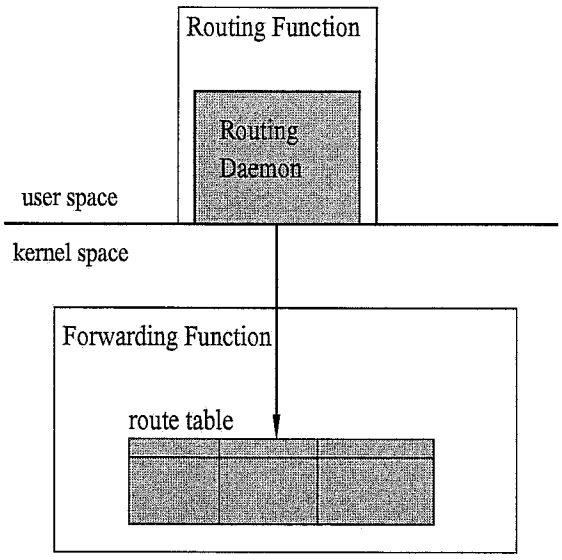
This chapter describes the details of our implementation of the DSDV routing protocol on the Linux operating system. It starts by giving an overview of routing architecture for modern Unix-like operating systems.

5.1 The OS Routing Architecture

Every operating system provides two modes of operations: a user space, where application programs written by the users can be executed, and a kernel space, where only operations unique to the operating system kernel can be executed. User space programs make use of "system calls" to avail these services which can run only in the kernel space.

In most Unix-like operating systems, the routing functionality is divided into two parts [12]. The first part, called *forwarding function*, resides inside the operating system kernel and deals with the task of directing packets to different outgoing network interfaces based on a table-driven process. The in-kernel forwarding function maintains a kernel routing table (Figure 5.1b) and consults this table before sending each packet to the corresponding "next-hop" neighbor through the corresponding outgoing network "interface."

The second part called *routing function* resides in the user space and is responsible for populating the kernel routing table. This functionality is usually implemented by a user-space routing daemon program (Figure 5.1a). The routing daemon engages in information exchange



(a) Routing Architecture

destination	next-hop	interface
10.0.0.1	138.126.136.1	eth0

(b) A typical kernel route table structure

Figure 5.1 Routing Architecture of Unix-like Operating Systems

with its peers at other nodes and employs some sort of routing protocol to compute the proper routes.

The separation of these two functions allows an efficient packet flow inside the kernel without incurring any context switch overhead, and yet provides the flexibility for adding or changing the routing protocol. Placing the *routing function* outside the kernel has the advantage of relieving the kernel of CPU or memory intensive tasks such as complex route computation or long-duration route discovery.

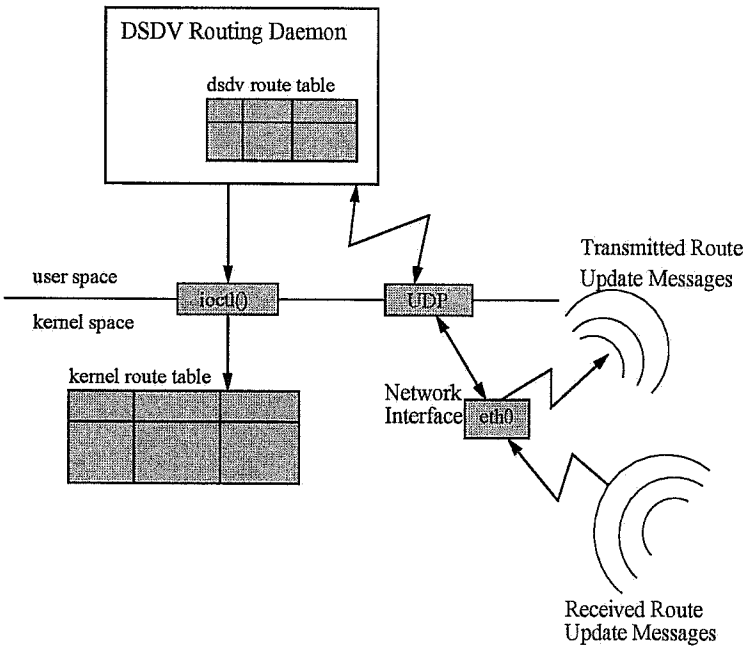


Figure 5.2 User-Space DSDV Routing Daemon

5.2 Overview of the Implementation

The DSDV routing protocol is implemented as a user-space routing daemon program. Our implementation was done on the Linux 2.4.17 kernel, but since the implementation requires no changes inside the kernel, it would work as well on any other Linux kernel version. The entire implementation has been done in C++.

Figure 5.2 illustrates the design of the DSDV daemon and its interactions with the kernel. The user-space DSDV routing daemon is a sequential program and creates no parallel threads or processes. The DSDV daemon maintains a local copy of the route table called *dsdv route table*. This route table contains learnt routes to all the destinations in the ad-hoc network. The routing daemon modifies this table whenever it receives new route update information on the wireless interface. The DSDV daemon transmits its own routing information periodically or in the event of significant changes to the *dsdv route table*. The dsdv route table is used to modify the route table inside the kernel through *ioctl()* system calls. *ioctl()* is a system call

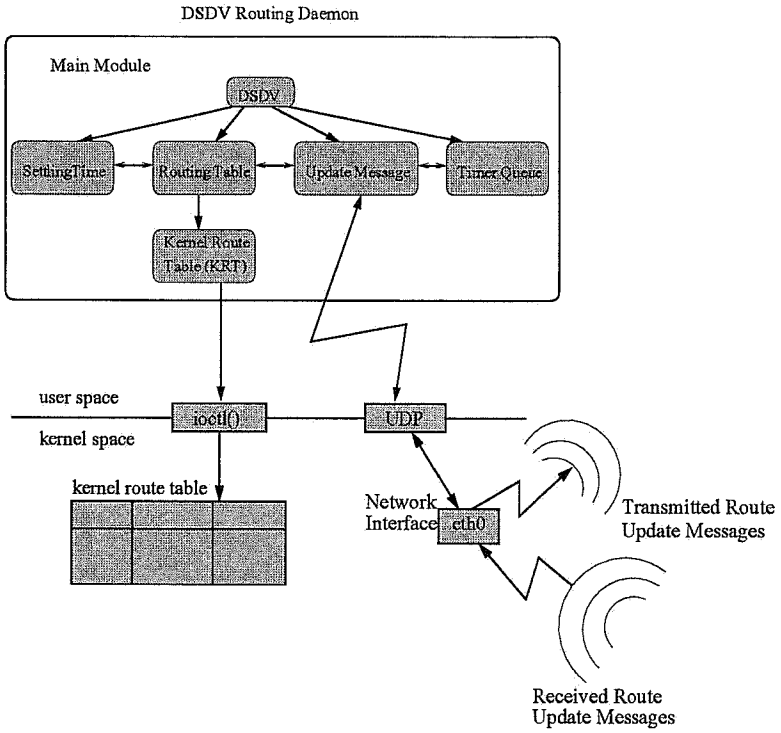


Figure 5.3 Modular Design of DSDV Routing Daemon

which allows manipulating the kernel route table from the user space. Whenever there are any significant changes in the dsdv route table, the corresponding entries in the kernel route tables are also modified by the DSDV routing daemon.

5.3 Implementation Details

The DSDV protocol implementation is divided into a number of modules. Figure 5.3 depicts a detailed modular design of DSDV routing daemon and how various modules interact with the kernel. This section provides an in-depth view of the implementation of DSDV routing protocol. It starts by describing the various data structures used. Different modules are explained in the subsequent section..

5.3.1 The Main Data Structures

Following are the data structures maintained by DSDV. The list shows only the private members of each class.

5.3.1.1 Routing Table Entry

The *rtable_entry* class defines the structure of a route table entry.

```
class rtable_entry {
    u_int32_t  dest_ip;       /* ip address of the destination node */
    u_int32_t  next_hop;      /* ip address of the next hop towards dest_ip */
    u_int32_t  dest_seq_num;  /* destination sequence number for dest_ip */
    u_int32_t  rt_metric;     /* number of hops to the destination */
    u_int64_t  lifetime;      /* lifetime of this route entry */
    u_int32_t  routing_flags; /* routing flags for the entry */
};
```

5.3.1.2 Routing Table

The DSDV Route table is defined by the class *routingTable*. It contains a *map* of *rtable_entry* to store routes for the destination nodes. This *map* is keyed by the IP address of destination node.

```
class routingTable{
    map<u_int32_t,rtable_entry>rTableMap;
};
```

5.3.1.3 Broadcast Entry

The *broadcast_entry* class defines the structure of a single destination entry being transmitted within the route update message.

```
class broadcast_entry{
    u_int32_t dest_ip;
    u_int32_t dest_seq_num;
    u_int32_t rt_metric;
};
```

5.3.1.4 Update Message

The *updateMessage* class defines the structure of the route update message advertised by a node in DSDV. It contains a list of *broadcast_entry*, one for each destination ip address being advertised.

```
class updateMessage{
    u_int8_t        type;              /* message type */
    u_int8_t        reserved1;
    u_int8_t        reserved2;
    u_int8_t        entries_cnt;       /* number of destination entries being transmitted */
    list<broadcast_entry>    brentry_list;
};
```

5.3.1.5 Settling Time Entry

The *settlingTimeEntry* class defines the structure of an entry to store the settling time data for a destination node.

```
class settlingTimeEntry{
    u_int32_t   dest_ip;          /* ip address of destination */
    u_int32_t   last_stl_time;    /* last computed settling time */
    u_int32_t   avg_stl_time;     /* average settling time */
    u_int64_t   last_recv_time;   /* timestamp when route table entry for
                                   * dest_ip was last updated/created */
};
```

5.3.1.6 Settling Time Table

The *settlingTimeTable* class is used by DSDV to store the settling time data for different destination nodes. It contains a *map* of *settlingTimeEntry* and is keyed by the IP address of the destination node.

```
class settlingTimeTable{
    map<u_int32_t,settlingTimeEntry>stlTimeMap;
};
```

5.3.1.7 Timer Entry

The *timer* class gives the structure of each timer entry maintained by DSDV.

```
class timer{
    u_int64_t       timeout;  /* timeout for the timer */
    timer_hfunc_t   handler;  /* handler function */
    void            *data;    /* data to be passed to handler function */
};
```

5.3.1.8 Timer Queue

The class *timerQueue* gives the structure of the timer queue maintained by the DSDV protocol. It contains a list of *timer* class entries.

```
class timerQueue{
    list<timer>     timerQ;
};
```

5.3.2 Modules

The DSDV implementation consists of seven major modules. Each of these modules are explained below.

5.3.2.1 The Main Module

The Main module ties together all the other modules of the user space routing daemon. It declares and initializes global data structures using command line arguments. It creates an instance of *dsdv* class (Section 5.3.2.2) and calls its *dsdv_daemon()* function, which is the main starting point of dsdv routing daemon.

5.3.2.2 The Dsdv Module

This module comprises the main thread of control inside the DSDV routing daemon. It contains an object of class *dsdvSocket* as its member function. This class provides functions

to initialize *dsdv socket* which is used to send and receive route control informations. It also provides interface functions to read and write data from/to dsdv socket.

The main control starts at *dsdv_daemon()* function. Member functions of *dsdvSocket* module are used to initialize dsdv socket. This module also adds the local node's route entry into dsdv and kernel route tables. It then sets up a reboot timer and a periodic broken link timer and enters into the main daemon loop. When the reboot timer expires, the node broadcasts its first route update message and starts a periodic update timer. The periodic broken link timer monitors the link status of neighboring nodes periodically to detect broken links.

When a packet arrives at dsdv socket, the node uses the first 8 bits in the packet to determine the type of the packet. Only packets of type *DSDV_PERIODIC_UPDATE* are processed by the routing daemon. If dsdv daemon receives a packet of desired type, it creates an *updateMessage* object from it and calls the *applyUpdates()* function of that class to incorporate received routing information into local route tables. A triggered route update message is sent out if needed.

Whenever a node needs to transmit a route update message, it calls the *getChangedEntriesCnt()* function of *routeTable* module to decide if the route update should use a full dump packet. It then calls *generateUpdateMessage()* function of the same module to generate an *updateMessage* object. The content of this object is copied into a send buffer and the packet is broadcasted on the dsdv socket.

5.3.2.3 The UpdateMessage Module

As explained above (Section 5.3.1.4), this module defines the structure of the route update message. It provides two main functionalities. First, it offers member functions to create an *updateMessage* object from the incoming data buffer and vice versa. Second, the *applyUpdates()* function of this module is responsible for extracting the useful informations from received route updates and applying it to the local route tables. The logic of the *applyUpdates()* function is as follows.

The *applyUpdates()* function scans through the received list of destination entries, and for every received entry it looks for a matching destination entry in the *dsdv route table*. If an entry

does not already exist, it adds a new entry for the received destination into dsdv as well as the kernel route tables. The later is achieved using the interface functions of *kernel route table* module. On the other hand, if an entry does exist in dsdv route table, it compares the value of the received sequence number with the value of stored sequence number for that destination. If the received value is higher than the stored one, it updates the route table entry with the new received information about next-hop, sequence number and route metric. The lifetime of the entry is also updated to current time plus *ROUTE_TIMEOUT*. The kernel route for this destination is deleted (if it exists) and a new route is inserted in the kernel route table. A delayed triggered route advertisement is scheduled if the received route metric is higher than the stored one. Otherwise an immediate triggered broadcast is done. If the values of both the sequence numbers are the same, then the existing route table entry is updated only if it has a higher route metric than the received entry. If the node gets an entry with a smaller sequence number that has an *INFINITY* route metric, it issues an immediate triggered route update to send out its latest information.

5.3.2.4 The RoutingTable Module

This module is responsible for maintaining the *dsdv route table*. It defines three main interface functions. The *getChangesEntriesCnt()* function scans the route table entries looking for the *Change Flag Bit*. It returns the number of route table entries which have changed since the last full dump.

The *generateUpdateMessage()* function creates the *updateMessage* object to be transmitted during route advertisement. It first inserts the local node's entry into the list. Then it scans through the route entries in dsdv route table. If the full dump flag is set, it adds all the route table entries to the list in *updateMessage*. Otherwise, for every route table entry, it checks the *Change Flag Bit* and adds only those entries into the list for which this flag is set.

The *refreshEntries()* function is called every time a periodic broken link timer expires. This function scans through the list of route table entries looking for expired entries (*lifetime* < current time). If such an entry is found and has an *INFINITY* route metric, then this entry is

deleted from the route table. If the found entry has a route metric not equal to *INFINITY*, the active route to this destination node is assumed to be broken. The kernel route table entry for this destination is deleted and the route table entry is modified to set an *INFINITY* route metric and incremented (+1) sequence number. If the expired route corresponds to a neighboring node (route metric = 1), then the route table entries for all other destination nodes using this node as the next-hop are also expired. These entries are also assigned an *INFINITY* route metric and incremented destination sequence number. The kernel routes for these new destinations are also deleted.

5.3.2.5 The Kernel Route Table (KRT) Module

This module provides functions to add and delete routes from the kernel route table. It uses the *ioctl()* system call to modify kernel routes. It takes an object of *rtable_entry* class as an input, generates a *struct rtentry* object out of that, which is passed to the *ioctl()* calls.

5.3.2.6 The SettlingTimeTable Module

This module is used to implement the damping fluctuations feature of the DSDV routing protocol. It provides member function *setAvgStlTime()* to compute the values of *last_stl_time* and *avg_stl_Time* as described in Section 4.4.8 of Chapter 4. It also provides an interface to set the *last_recv_time* value for a destination node.

This module collaborates with the *updateMessage* module to accurately maintain data in the settling time table. The value of *last_recv_time* is set inside *updateMessage* module whenever the route table entry for a destination node is created/updated. The *last_stl_time* and *avg_stl_time* values are also computed inside *updateMessage* module when an entry with the same sequence number but smaller route metric is received for a destination.

5.3.2.7 The TimerQueue Module

The DSDV routing daemon uses the *timerQueue* module to maintain different timers. The protocol maintains four different kinds of timer as stated in Section 4.4.11 of Chapter 4.

This module provides a universal handler function for all the timers, called *scheduleTimer()*. This function is called when any of the timers expires and it in turn calls the handler functions of all the expired timers. Last, it sets the system timer to the next unexpired timer value.

The *set_timer()* function is the generic function which is used to add new timer entries to the timer queue. It also resets the system timer (if needed) to the smallest unexpired timer value.

5.3.3 Configuration Parameters

The following values of DSDV configuration parameters have been in the current implementation:

```
#define PERIODIC_UPDATE_INTERVAL        2000
#define ROUTE_TIMEOUT                   5*PERIODIC_UPDATE_INTERVAL
#define DELETE_PERIOD                   5*PERIODIC_UPDATE_INTERVAL
#define BROKEN_LINK_TIMER_INTERVAL      1000
#define MAX_FULL_DUMP_TIMEOUT           5*PERIODIC_UPDATE_INTERVAL
#define FULL_DUMP_FACTOR                0.6
#define STL_AVG_FACTOR                  0.7
#define STL_TIME_FACTOR                 2
```

CHAPTER 6

Adaptive DSDV: Design and Implementation

6.1 Motivation

The DSDV [22] routing protocol has a number of configuration parameters. These include the periodic update interval and the full dump interval.

Choice of some or all of these parameters affects the performance of the DSDV protocol. It is very difficult to obtain optimal values for different DSDV parameters. Simulation results with different sets of parameters might help to find out optimal values for these parameters. But, it is not possible to exactly duplicate real life scenarios in the simulation world, and so one can not set optimal parameter values to run DSDV protocol on a real test bed is still a big challenge. Also, to a very great extent the choice of these parameters depends on the network size and the rate of mobility within the network. Thus, even if optimal values are chosen for these parameters based on some simulation data, these values would no longer be optimal when the mobility pattern of the network changes or mobile nodes get added/deleted to/from the ad-hoc network. What is needed is a mechanism to enable the network to automatically figure out the values of these parameters based on the dynamics of the network currently and in recent the past. This would obviate the need of hard coding the values of these parameters. Also, the use of a dynamic algorithm to choose these parameter values would imply that optimal parameter values are chosen for all kinds of networks irrespective of their sizes and mobility patterns.

The Adaptive DSDV (A-DSDV) protocol that we have developed addresses these issues and provides a mechanism to adaptively and dynamically select the values of important DSDV parameters.

6.2 Design of Adaptive DSDV

The Adaptive DSDV (A-DSDV) protocol is an extension of the DSDV routing protocol. It addresses the issue of how to select optimal values of DSDV parameters irrespective of the size of the network and the mobility patterns within the network. The goal of the protocol is to provide a dynamic mechanism, to determine the values of main DSDV parameters, which is adaptive to the changes in the network conditions. A-DSDV has been designed with the vision of enabling a fully automatic version of the DSDV protocol without the need to set any hard coded parameters.

The protocol provides adaptive mechanisms for the following DSDV parameters:

- Periodic Update Interval
- Full Dump Interval

6.2.1 The Periodic Update Interval

The periodic update interval is the most important parameter of the DSDV routing protocol. The value of this parameter affects the aggregate routing overhead of the protocol. Setting this parameter to a very low value results in too much routing overhead, while setting it to a very high value results in slow convergence of the DSDV routing protocol.

The idea behind computing the periodic update interval adaptively as the network dynamics change can be explained as follows. If a network is fairly stable in terms of node mobility, then the periodic update interval should be large to avoid unnecessary rebroadcast of the same routing information. On the other hand, if the rate of change of network topology is high for the network, then periodic update interval should be small enough to ensure that all the mobile nodes in the network get updated routing informations as soon as possible.

To compute the periodic update intervals adaptively, a mobile node follows these steps:

- It keeps a history of the last k periodic update intervals. It also keeps track of the number of route table changes in each of those intervals. Let u_i denotes the *ith* periodic update interval and let n_i denotes the number of route table changes during that update interval.
- The new periodic update interval is computed using the following formula.

$$u = \frac{1}{k}\sum_{i=1}^{k}\frac{u_i}{\alpha n_i} \tag{6.1}$$

  where α represents the scaling factor for route table changes and it is set to 2.
- In case there are no routing updates in the network, the periodic update interval converges to *MAX_PERIODIC_UPDATE_INTERVAL*.

The equation for computing the adaptive periodic update interval has been designed so that update periods with very high number of route table changes tend to decrease the resulting value of the new computed interval. The new periodic update interval is inversely proportional to the number of route table updates in all the past k update periods. So, the higher the number of routing changes, the smaller will be the value of next update interval. Thus the algorithm fully adapts to the changes in the network topology, as desired.

6.2.2 The Full Dump Interval

The next important DSDV parameter addressed in the design of adaptive DSDV is the full dump interval. A full dump consists of advertising all available routing informations. The full dump interval is the time duration between two such successive route advertisements. If the full dump route updates are sent very frequently, it results in a high routing overhead. On the other hand, if the full dump interval is too large, then the size of incremental dumps will eventually become large and it will consume a considerable amount of network bandwidth. This increases the routing overhead, and thus is not desirable. Also, any newly added nodes in the ad-hoc network might experience long delays before they receive all the routing informations. This increases the overall convergence time of the routing protocol.

In the original DSDV, the full dump interval is not defined in the units of time. DSDV does a full dump in the following two conditions:

- Whenever the number of modified routing table entries becomes more than the *FULL_DUMP_FACTOR* times the total number of entries in the route table,
  Or
- The time duration since last full dump becomes more than the *MAX_FULL_DUMP_INTERVAL.*

The A-DSDV protocol optimizes the first of these two conditions. The second condition is retained as is.

6.2.2.1 The $\sqrt{(2n)}$ Law for the Full Dump Interval

We provide a new "$\sqrt{(2n)}$ Law" for the length of the full dump interval, and show that it is optimal.

The idea is to minimize the average cost of all dumps. Figure 6.1 illustrates the average cost in each successive periodic update interval just after a full dump has been done. At interval 0, a full dump route advertisement was done. Let us also assume that the average cost of an incremental dump increases linearly with time. After k incremental dumps, a full dump is performed again, which advertises all the available route table entries (n). The average cost (say C_k) of all routing updates after the last full dump is given by:

$$C_k = \frac{1+2+3+-----+k+n}{k+1} \tag{6.2}$$

$$= \frac{k}{2} + \frac{n}{k+1} \tag{6.3}$$

$k = \sqrt{(2n)} - 1$ minimizes the above cost function.

Thus we obtain the following theorem:

Theorem: The $\sqrt{(2n)}$ law for the full dump interval

The average cost of route table updates is optimal when a full dump is done after every $\sqrt{(2n)} - 1$ *incremental dumps.*

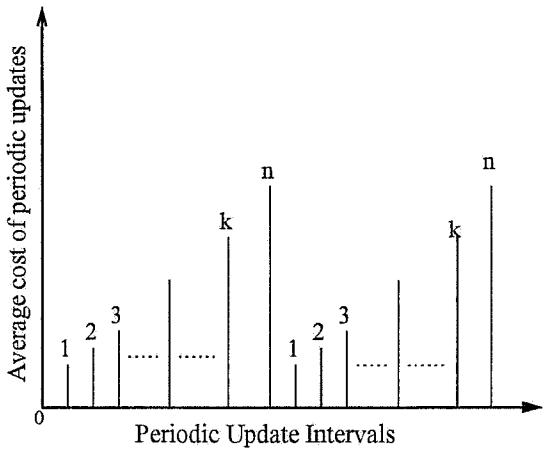


Figure 6.1 Average Cost of Full Dump

This theorem has been employed in designing the following rule, which states when a node should attempt a full dump of routing informations.

- Once the number of modified routing table entries reaches $\sqrt{(2n)} - 1$ the node should do a full dump.

Implementing this requires Adaptive DSDV to keep a running average of the value of n.

6.3 Overview of the Implementation

The A-DSDV has been implemented on top of the previously described implementation of DSDV. It modifies DSDV to use adaptive periodic update intervals, and $\sqrt{(2n)}$ Law for advertising the full dump of routing information. Figure 6.2 illustrates the design of Adaptive DSDV protocol highlighting the additional logical modules it adds to the DSDV routing protocol.

A-DSDV stores the *MAX_UPDATE_HISTORY_CNT* number of past periodic update intervals along with the number of routing changes in each one of them. It uses this to compute the values of update intervals dynamically. The protocol uses the newly computed value as the value of DSDV parameter *PERIODIC_UPDATE_INTERVAL*. Before doing a broadcast of

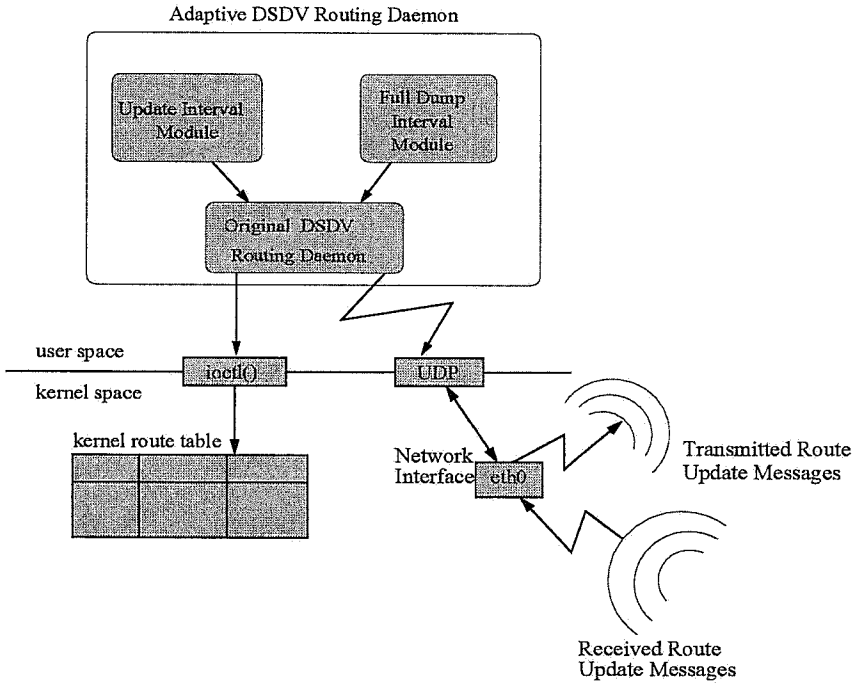


Figure 6.2 Adaptive DSDV Routing Daemon

routing informations, the $\sqrt{(2n)}$ Law is applied to make a decision about doing a full dump of available route information.

6.4 Implementation Details

6.4.1 The Main Data Structures

A-DSDV employs the following additional data structures to implement its functionalities.

```
class updateInterval{
   u_int32_t interval;   /* duration of the update interval */
   u_int32_t rUpdateCnt; /* number of routing updates in during this interval */
};
   deque<updateInterval>   qUpdateInterval; /* queue to store the history of update intervals */
   u_int32_t               avg_N; /* running average of the number of route table entries */
```

The values of the past k periodic update intervals are stored in a queue of update intervals called *qUpdateInterval*. Each entry is an object of type *updateInterval* and also stores the

number of route table updates during that interval. Variable *avg_N* is used to keep a running average of the number of entries in the routing table. The protocol initializes this to 0.

6.4.2 The Modules of A-DSDV

The Adaptive DSDV functionality is split into two logical modules: *Update Interval* module and *Full Dump Interval* module.

6.4.2.1 The Update Interval Module

This module is responsible for computing the adaptive periodic update intervals. It maintains a global variable to record the time of the last periodic route update. When a periodic update message is transmitted, this variable is used to record the value of the current periodic update interval. A global variable called *num_routing_updates* is incremented every time a route table entry is modified. This variable is stored in *qUpdateInterval*, along with the value of current periodic update interval. The *num_routing_updates* is reset to 0 whenever a periodic update is sent out. The module uses equation(6.1) to compute the value of new periodic update interval and uses this to set the next periodic update timer.

DSDV is modified to define the value of *PERIODIC_UPDATE_INTERVAL* as the output of this module. All other DSDV parameters defined in terms of *PERIODIC_UPDATE_INTERVAL* are also modified to use the adaptively computed value of the update interval.

6.4.2.2 The Full Dump Interval Module

This modules is called just before transmitting a periodic route update packet. It uses the $\sqrt{(2n)}$ Law to determine if a full dump should be done. Let us assume that t_{curr} denotes the current system time. The module employs the following formula to compute the average value of n.

$$avg\_N = \{\beta * agv\_N\} + \{\{1 - \beta\} * t_{curr},\} \tag{6.4}$$

where β denotes the fraction used for computing the running average for the value of n, and is called *RUNNING_AVG_FRACTION.*

Before transmitting a route update message, the full dump module is invoked. It scans the route table to count the number of modified route table entries since the last full dump. The first bit of the routing flags in the route table entry is used for this purpose. If the number of modified route table entries are more than $\sqrt{(2n)}$, then the module returns true and DSDV protocol broadcasts a full dump of available routing information. If the module returns false, only modified route table entries are transmitted.

6.4.3 A-DSDV Parameters

Though A-DSDV is designed to make DSDV free of hard coded parameters, it introduces some new parameters. Different choices for these parameters however do not drastically change the performance of A-DSDV protocol and it is considerably easier to come up with default values for these parameters.

```
MAX_PERIODIC_UPDATE_INTERVAL        10000 ms
RUNNING_AVG_FRACTION                0.4
MAX_UPDATE_INTERVAL_HISTORY_CNT     10  /* number of past periodic
```

CHAPTER 7

System Services for Ad-hoc On-demand Routing

7.1 Introduction

Most of the MANET routing protocol studies are simulation based [39, 40, 41], with very few real implementations. One of the goals of this thesis is to provide implementations of some of the MANET routing protocols and conduct studies on real test beds. Validating MANET algorithms in real systems is necessary for their proliferation in the real world. However, some of the MANET routing protocols require sophisticated system-level programming which makes the task of implementing these routing protocols fairly difficult.

Most of the proactive ad-hoc routing protocols fit very nicely into the Routing Architecture described in Section 5.1 of Chapter 5. These protocols can be easily implemented as user-space daemons without requiring any special support from the operating system or the kernel. This is however not true in general for reactive routing protocols. Such protocols often employ new routing models or have special requirements that are not directly supported by current operating systems. This makes it a tough job to carry out the implementation of these MANET routing protocols.

The core reason for having such difficulties in implementing reactive ad-hoc routing protocols is the lack of system support and programming abstractions in general purpose operating systems (such as Unix/Linux). Without proper systems support and convenient programming abstractions, implementors are forced to do low-level system programming, and often end up

making unplanned changes to the system internals in order to gain the additional functionality required for reactive ad-hoc routing. Not only is this a non-trivial task, but in practice it can also lead to unstable systems, incompatible changes (caused by different implementations), and undeployable solutions. For example, if an ad-hoc network implementation requires a special version of the OS kernel, fewer users will be willing to install it.

This chapter addresses some of these issues and develops system support and programming abstractions needed to facilitate reactive MANET routing protocol implementations and deployment. This work has been done jointly with two other authors [42]. The solution provides a set of system services which meet the requirements of most of the reactive ad-hoc routing protocols. The new programming abstractions also allow easy programming of reactive ad-hoc routing protocols without the need for doing low-level systems programming.

7.2 Challenges in Reactive Ad-hoc Routing

Reactive on-demand ad-hoc routing protocols do not fit into the traditional software architecture for performing routing on Unix-like operating systems (Section 5.1). This is due to the fact that they are based on a different routing model than the proactive routing protocols. Such protocols discover routes as needed instead of maintaining a route table containing routes to every other destination in the network.

Normally, each packet going through the kernel forwarding function will be matched against the kernel route table. If no entry matches its destination, the kernel will drop the packet immediately. However, this is exactly what should not be done in order to implement on-demand ad-hoc routing protocols, (e.g., DSR [29] and AODV [27]). In on-demand ad-hoc routing, not all routes will exist *apriori*; they must be "discovered" when needed. In such cases, the correct behavior should be: To notify the ad-hoc routing daemon of a route request, and to withhold the packet until the discovery finishes, and route table has been updated.

These on-demand protocols typically maintain a cache of recently used routes in user space to optimize the route discovery overhead. Each entry in this user space route cache has an associated timer, which needs to be reset when that route is used. The entry is deleted (both

from the user space route cache and the kernel routing table) when the timer for that entry expires. The trouble is that the user space routing daemon has no way of finding out when a route is used in the kernel because no record is available of the time when a route was last used.

To summarize, we identify the following system capabilities required for on-demand mobile ad-hoc routing:

1. To identify the need for a route request.
2. To notify ad-hoc routing daemon of a route request.
3. To queue outstanding packets waiting for route requests.
4. To re-inject them after route discovery, and
5. To refresh timers in the user space route cache when routes are used in the kernel.

Unfortunately, none of the current general purpose operating systems (including Linux) meet these requirements. The next section provides a general solution for the issues raised here. It proposes this solution to be included in all future OS internals so that reactive ad-hoc routing can be easily supported.

7.3 A General Solution for On-demand Routing Protocol

A general solution is developed to support on-demand routing in general purpose operating systems. The purpose is to suggest modifications to these operating systems so that ad-hoc routing can be easily supported in the future. Enhancements to the current packet-forwarding function are proposed with the following mechanisms.

First, an additional flag should be added to each kernel route entry to denote whether it is an *on-demand* entry. An on-demand route entry is said to be *deferred* if it has empty next-hop or interface fields, meaning that the route is yet to be discovered. Instead of getting dropped as in normal packet forwarding, packets matching a deferred route will be processed by a special module, which implements the desired behavior as described in Section 7.2. It is not necessary

to include every possible on-demand destination in the route table. Flagging a subnet-based route or the default route as on-demand can serve the same purpose.

Second, a new on-demand routing module (ODRM) should be added to complement the kernel packet-forwarding function and implement the desired on-demand routing functionalities. When it receives a packet for a deferred route, the module first notifies the user-space ad-hoc routing daemon of a *route request* for the packet's destination. Then, it stores the packet in a temporary buffer and waits for the ad-hoc routing daemon to return with the route discovery status. Once this process finishes and the corresponding kernel route table entry is populated, the stored packets are removed from the temporary buffer and re-injected into packet forwarding path.

Next, a timestamp field needs to be added to each route entry to record the last time this entry was used in packet forwarding. This timestamp can be used to retire a stale route that has not been used for a long time.

Finally, we should provide a programming abstraction (API) so that these new mechanisms can be conveniently used in an ad-hoc routing daemon program. The API should contain the following functions:

- ```
  int route_add(addr_t dest,
                addr_t next_hop, char *dev);
  int route_del(addr_t dest);
  ```

  Basic routines to add or delete an on-demand entry from kernel route table. To add a deferred route entry, specify `next_hop` to be 0. (Here, `addr_t` is a generic type for the network address, such as `unsigned long` for IPv4 address.)

- ```
  int open_route_request();
  int read_route_request(int fd,          struct route_info *r_info);
  ```

  ODRM notifies the ad-hoc routing daemon about the route requests in the form of an asynchronous stream. The first function returns a file descriptor for this stream and the second function fills in information about the route requests in the second argument, `struct route_info*` which is defined as follows :

```
struct route_info {
        addr_t dest;
        addr_t src;
        u_int8_t protocol;
    };
```

This information is provided to enable the routing daemon to implement protocol semantics correctly. For example a different action may be warranted if the packet was generated locally rather than being forwarded for some other host. This can be deduced from the `src` field of `struct route_info`. Similarly, some protocols may need to know if the packet for which a route request was received was a TCP packet or a UDP packet.

The file descriptor semantics allow the ad-hoc routing daemon to use either an event-driven or a polling strategy. The second function blocks until the next route request becomes available.

- `int route_discovery_done(addr_t dest, int result);`

  This informs the ODRM that a route discovery for the given destination has completed and the kernel route table populated. The result field indicates whether the route discovery was successful or not.

- `int query_route_idle_time(addr_t dest, int valid_flag);`

  Given a destination, this returns the idle time recorded in the kernel route table for this entry (elapsed time since the last use of the route). The `valid_flag` is used to differentiate between the return values.

valid_flag	Interpretation
1	Idle time since a packet was last forwarded to dest
0	Idle time since a packet was last received from the dest

- `int close_route_request(int fd);`

  This function is called by the routing daemon when it no more desires to receive any more route requests. This enables the ODRM to free the memory used up by packets already queued up and close the fd.

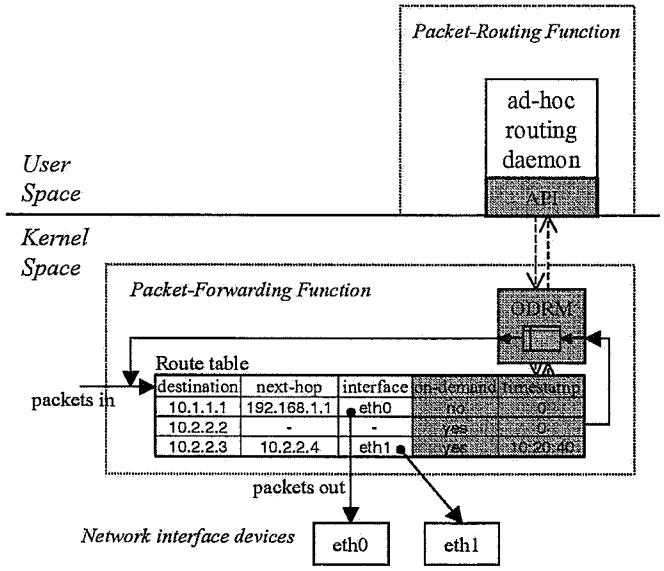


Figure 7.1 The general solution

- `int set_route_auto_timeout(addr_t dest,`

  `int sec);`

  This function lets the ODRM delete a route automatically if not used for the given time.

Figure 7.1 illustrates the architecture and components for this general solution. The shaded parts implement the proposed general solution..

Implementing this architecture in a Unix-like modern operating system usually requires moderate changes to the system internals. For example, the kernel route table data structure can be expanded with a new column to hold the last-use timestamp. The extra "on-demand" bit can be accommodated in the existing `rt_flags` field. The ODRM module can use a simple hashed table to store and process deferred-routing packets. The route-request stream can be implemented using a socket or a special file (such as in the `/proc` file system). Finally, the API can be implemented as a user-space programming library.

It is debatable whether the ODRM functionality should be implemented outside the kernel and whether it is better to queue all deferred-routing packets in user-space. The advantage of doing so in user space is that it reduces the kernel complexity and memory usage. If the routing

protocol requires prolonged route discovery procedure, it will be more efficient to buffer packets outside the kernel. The disadvantage is the need to copy every deferred-routing packet from kernel to user-space and to re-inject it back to the kernel when the routes are ready, but it can be argued that such overhead is insignificant compared with the time and overhead incurred in an average route discovery. A user-space ODRM function does impose two new requirements: The ability for the kernel to pass an arbitrary network packet including headers to user-space (i.e., not generated by or destined to this host), and the ability for the user-space module to re-inject the same packet into the kernel intact. The former may require system-dependent mechanisms such as a special device driver, but the latter can use the raw socket interface that is common to most Unix-like operating systems.

7.4 Implementation of ODRM in Linux

A long-term objective is to implement this solution in common operating systems and make it the standard in future versions. However, the immediate goal is to make it available to current Linux 2.4 users because getting changes accepted into a standard release is a time consuming process. Although it is not too difficult to modify Linux 2.4 kernel code to implement this, it would be better if the services described in Section 7.3 could be provided without making such changes. This strategy certainly has practical value as fewer users are willing to modify their operating systems. To do this efficiently needs a careful design, which is described in this section.

Unlike many other Unix-like operating systems, Linux provides several flexible mechanisms for extending the kernel functionalities. These include loadable modules, where a new kernel function can be inserted into a running kernel without recompiling or rebooting, and a packet filtering and mangling facility called Netfilter [43]. In particular, Netfilter provides a set of hooks in the kernel networking stack where kernel modules can register callback functions, and can change/mangle each packet traversing through the corresponding hooks. The implementation of system services uses only these two Linux mechanisms and it does not make any changes to the standard kernel code.

7.4.1 Design and Mechanisms

In the current Linux implementation, the ODRM functionality is implemented in user-space to reduce the kernel complexity and memory requirements. There are two possible ways to implement this functionality in user space. It can be implemented as a shared library which exposes an API. Any routing daemon wishing to use this functionality will link this library with its code. Another approach is to implement the ODRM functionality as a separate daemon program (Ad-hoc Support Daemon) which communicates with the routing daemon using some inter-process communication mechanism like sockets. Both approaches have their pros and cons. The library approach is definitely more efficient because it does not have the overhead of inter-process communication. However, any bug in the library is likely to crash the routing daemon also. The library approach gives a more natural picture of the ODRM functionality as system services, i.e, the API is available as a direct function call once the appropriate header files are included.

This functionality has been implemented as a library called the Ad-hoc Support Library (ASL) or libASL. ASL implements the API described in Section 7.3. The rest of this section explains the implementation of the different issues described in Section 7.2 and 7.3.

To solve the problem of identifying the need for a route request, we need to filter all packets for which there exists no route. Without modifying the routing table structure, there is no simple way to do that in the kernel. This has been accomplished with an unused local tunnel device called Universal TUN/TAP (`tun`) as the "interface" device for these destinations. To catch packets for all such destinations, a default route can be used which can be setup like this:

```
ifconfig tun 127.0.0.2 netmask 255.255.255.255 \
        broadcast 127.0.0.2 up
route add default dev tun
```

TUN/TAP is a virtual tunnel device that makes available all received packets to a user-space program through the `/dev/net/tun` device. In current implementation, this device is opened by a call to `open_route_request()`; hence it receive all packets that the kernel writes to `tun`,

i.e., all packets for which there is no route. This also solves the problem of passing and storing packets in user-space.

Whenever a new packet is read from `/dev/net/tun`, the ad-hoc routing daemon which has opened a route request gets notified on that fd. It can read the details of the route request through `read_route_request()` and can then initiate route discovery for the requested destinations. These packets are temporarily queued in a hashed table keyed by the destination IP address. This functionality is implemented in the Ad-hoc Support Library. Since the buffer is in user-space, a large buffer is available to queue packets. This means that packets would not be lost even if the route discovery delays are large.

The next issue is to re-inject packets back into the IP stack after a successful route discovery. The mechanism used is a raw socket[1]. A packet sent through a raw socket will be inserted as is (bypassing any IP and header processing) to the kernel output chain just before the packet-forwarding function. Here, a raw socket is used to send the queued packets out. These packets are appropriately routed in kernel using the newly discovered routes.

Now let us look at the last problem mentioned in Section 7.2, to refresh entries in the user space route cache when a route is used. Since we are not making changes to the kernel route table, the only way is to maintain a separate timestamp table for each entry in the route table. A simple kernel module called `route_check` is designed to maintain this table and register it at Netfilter's `POST_ROUTING` hook (after route table lookup and before entering the physical network interface). This means that every outgoing packet will pass through this module. It simply peeks at the packet header and updates the corresponding timestamp value. This timestamp information is made available to user-space programs using an entry in the `/proc` file system. The `query_route_idle_time()` function exposed by the ASL API reads this file (`/proc/asl/route_check`) to determine the idle time for a destination. The ad-hoc routing daemon can check the freshness of a route by reading this file, and delete the stale routes from the kernel route table accordingly.

[1]Raw sockets are normally used to handle packets that the kernel does not support explicitly. The `ping` program, for example, uses raw sockets to generate ICMP packets.

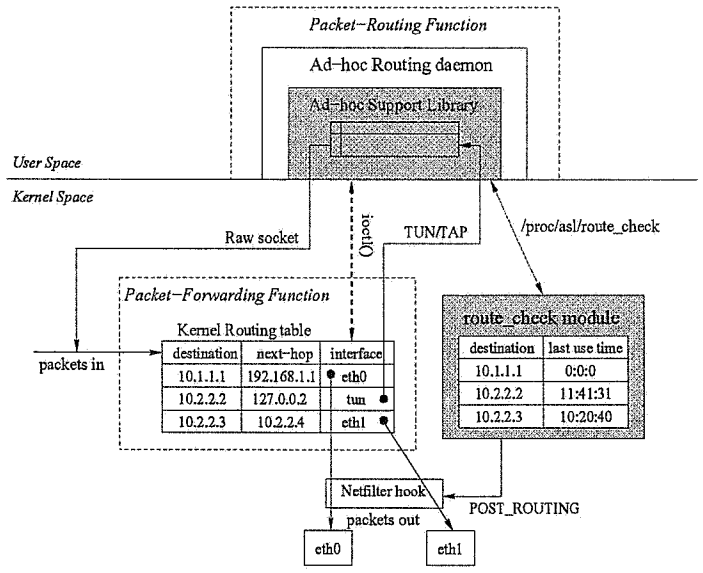


Figure 7.2 Linux implementation structure

7.4.2 Implementation Details

Figure 7.2 illustrates the structure of our implementation. The two main components are the user space library ASL and the kernel module `route_check`. The library implements the API described in Section 7.3. We now briefly describe how we implement these function in our library.

`route_add()` and `route_del()` functions add or delete routes to the kernel using the `ioctl()` interface. When the user indicates that the route be a deferred route by specifying an empty next-hop, the device for the route is automatically made to be `tun`. `open_route_request()` initializes the tun device, the raw socket, the data structures to queue the deferred packets and also inserts the `route_check module` in the kernel. The data structure to store the packets is a hash table of queues, keyed by the destination IP address. This function returns the descriptor of the tun device which can be monitored using a polling or event driven strategy. `read_route_request()` blocks reading from this tun device. When a packet is received on tun, this function stores the packet and delivers information about the packet in the form of `struct route_info`. The return value of the function indicates if a route discovery is already

in progress for this destination. Based on this the routing daemon initiates route discovery and calls `route_discovery_done()` upon completion of this process. If route discovery was successful then this function retrieves the packets for that destination from the storage and sends them out on the raw socket. If route could not be found then the packets are thrown away and the memory used for them is freed. `query_route_idle_time()` reads the `last_use_time` for that destination from `/proc/asl/route_check` and returns the idle time. This needs to be called whenever the routing daemon has to make a decision to expire routes from its user space route cache. `close_route_request()` simply shuts down all the sockets, frees all the memory for storing the packets and removes the `route_check module` from the kernel.

The following is a pseudocode of a typical routing daemon which uses this library:

```
aslfd = open_route_request()
route_add(default,0) /* add deferred route */
loop  /* this could be select or poll */
  wait for input from {aslfd or other fd's}
  if input from aslfd
    dest = read_route_request()
    if(route request is new)
      do route discovery for dest
      if successful
        add route for dest to kernel
        route_discovery_done(success)
      else
        route_discovery_done(failure)
      end
    else
      continue
    end
  end
  if input from other fd's
    process according to protocol semantics
    /*call before expiring routes*/
    query_route_idle_time()
```

```
  end
end
close_route_request()
```

7.5 ASL System Requirements

The Ad hoc Support Library (ASL) uses several features exclusive to the Linux 2.4 kernel. The following are required for ASL to work:

- Kernel version greater than 2.4.3
- Loadable module support in the kernel
- TUN/TAP support in the kernel.
- Netfilter software and support in the kernel.

We have used ASL to implement the AODV routing protocol. Details are provided in Chapter 9.

CHAPTER 8

Ad-hoc On-demand Distance Vector Routing

The Ad-hoc On-demand Distance Vector (AODV) [27, 44] routing protocol is intended for use by mobile nodes in an ad-hoc network. It not only offers quick adaptation to dynamic link conditions, but also incurs low processing and memory overhead. It uses destination sequence numbers to ensure freedom from routing loops at all times (even in the face of anomalous delivery of routing control messages), avoiding problems (such as "counting to infinity" [12]) associated with classical distance vector protocols.

8.1 Properties of AODV

The AODV routing algorithm is an on-demand routing algorithm, meaning that it builds routes between nodes only as needed by source nodes and maintains them only as long as necessary. AODV is loop free at all times, even while repairing broken links. This loop freedom is accomplished through the use of sequence numbers. The concept of sequence numbers in AODV is same as the sequence numbering used in the DSDV routing protocol. The only difference is that the AODV does not differentiate between even and odd sequence numbers, while DSDV does. Every node maintains its own monotonically increasing sequence numbers, which it increases every time it sends out a new route request. This sequence number ensures that the most recent route is selected whenever the route discovery procedure is executed. AODV utilizes the process of route discovery and route maintenance to discover and manage routes in a mobile ad-hoc network.

AODV currently require symmetric links between neighboring nodes. It utilizes an enhanced version of the traditional route table to store and maintain routes to the destination nodes. It also provides quick deletion of invalid routes through the use of a special route error message. AODV responds to topological changes that affect active routes in a quick and timely manner. It builds routes from routing control messages with only a small amount of overhead, and no additional network overhead. Finally, AODV does not place any additional overhead on data packets because it does not utilize source routing. The AODV protocol supports both unicast and multicast communication abilities. Here we focus on the unicast features of AODV routing protocol.

8.2 Protocol Overview

AODV is a pure on-demand ad-hoc routing protocol. It uses the route request/route reply discovery cycle to discover routes to new destination nodes. Route Requests (RREQs), Route Replies (RREPs), and Route Errors (RERRs) are the three main message types used by AODV. When a route to a destination is needed, node broadcasts a RREQ message. A node in the network having a "fresh enough" route to that destination replies back with a RREP message. The RERR message is used to notify other nodes in the event of a link break.

8.3 Route Table Management

AODV stores information about all known routes to the destination nodes in a route table. Each route table entry contains the following fields:

- Destination IP Address
- Destination Sequence Number
- Interface
- Hop Count (number of hops needed to reach destination)

- Last Hop Count
- Next Hop
- List of Precursors
- Lifetime (expiration or deletion time of the route)
- Routing Flags.

The Last Hop Count field is used to prevent uncontrolled dissemination of RREQ messages. The Precursor list for a destination is the list of nodes which use the local node as next-hop towards that destination. This information is used to disseminate RERR messages to affected nodes in case of a link breakage.

8.4 Route Establishment

As long as the end points of a communication connection have valid routes to each other, AODV does not play any role. Whenever a new route is needed, AODVs route selection mechanisms come into play. AODV uses RREQ and RREP messages to discover new routes.

8.4.1 Route Discovery

When a source node desires to send a packet to some destination node and does not already have a valid route to that destination, it initiates a *route discovery* process. To begin such a process, the source node creates a RREQ packet. In addition to the source node's IP address, its current sequence number, hop count, and a broadcast ID, the RREQ also contains the latest sequence number for the destination node which the source node is aware of. The source node increments its own sequence number each time it sends out a route request. The IP address of the source node and broadcast ID uniquely identifies a RREQ packet. The Source node also sets the hop count value to zero. After creating the RREQ packet, the source node broadcasts the packet to its neighboring nodes. A timer is then set to wait for the RREP packet.

A node receiving a RREQ packet first checks to determine if it has received a RREQ with the same source IP address and broadcast ID in the recent past. If yes, it discards the received RREQ. On the other hand if it is a new RREQ, the node proceeds as explained next. The node creates a *reverse route* entry to the source IP address in its route table if it does not already have one. If an entry for the source node already exists, then it is modified in two cases. First, if the received RREQ contains a higher sequence number for the source node than the one stored in the route table entry for the source node. Second, if the received sequence number is same as the stored sequence number for the source node, and the hop count specified by RREQ plus one is smaller than the existing hop count value in the route table entry for the source node. The sequence number and hop count information for *reverse route* entry is copied from the received RREQ packet. This route entry is assigned an updated *lifetime* value and will be deleted if not used within the specified lifetime. A node discards all the RREQ messages received over a unidirectional link without any processing.

The node receiving a RREQ generates a route reply (RREP) if it is either the destination node or if it has an active route to the destination node with the corresponding sequence number greater than or equal to the destination sequence number contained in the RREQ packet. When either of these conditions is satisfied, the node does not rebroadcast the RREQ. Otherwise, the node updates and broadcasts the RREQ packet to its neighboring nodes. The RREQ update involves incrementing the hop count field in the RREQ packet to account for the new hop through this node.

8.4.2 Expanding Ring Search

Each time a node initiates route discovery for some new destination, it must broadcast a RREQ across the entire network. For a small network, the impact of this flooding is minimal. However,for a large network the impact may become increasingly detrimental. To prevent network-wide dissemination of RREQs, the source node should use an expanding ring search technique, which allows a search of increasingly larger areas of the network, if a route to the destination is not found. In an expanding ring search, the source node sets the Time to Live

(TTL) value of RREQ packet to an initial value of *TTL_START*. If an expired route to the destination exist, then the initial TTL value is set to Last Hop Count plus *TTL_INCREMENT*. RREP timer is set based on the current value of TTL being used. If no route reply is received and the timer expires, the TTL value is incremented by *TTL_INCREMENT* in the subsequent attempts until it reaches *TTL_THRESHOLD*. After this the RREQ is broadcasted to the entire network. The number of times RREQ can be broadcasted across the entire network is limited by the *RREQ_RETRIES* parameter.

8.4.3 Forward Path Setup

Eventually, the RREQ will reach a node which is the destination node itself or an intermediate node which has a "fresh enough" route to the destination. If an intermediate node has a route entry for the desired destination, it determines whether the route is current by comparing the destination sequence number in its own route entry to the destination sequence number in the RREQ. If the RREQ sequence number for the destination is greater than that recorded by the intermediate node, the intermediate node must not use its recorded route to respond to the RREQ. Instead, it just rebroadcasts the RREQ. The intermediate node can reply only if it has an active route to the destination node with the corresponding sequence number greater than or equal to the destination sequence number contained in the RREQ packet.

If the node does have an active route to the destination, it creates a RREP packet. The destination IP address and the source IP address are copied to RREP packet from the received RREQ packet. The processing of RREQ packet is slightly different, depending on whether the node is itself the requested destination, or it is an intermediate node with an admissible route to the destination. The two cases are described below. Once the RREP is created, it is unicasted to the next-hop on the *reverse route* towards the source of the RREQ packet.

If the node generating RREP is the destination itself, it updates its own sequence number to the maximum of its current sequence number and the destination sequence number contained in the RREQ. The resulting sequence number is placed in the RREP packet. The node also

puts a zero hop count value in the RREP packet. The lifetime of the supplied route is set to the local node's own lifetime value.

If an intermediate node generates a RREP message, it copies the last known value of the sequence number for the destination into the RREP packet. The hop count value from the route table entry is also placed into the generated RREP. The node also updates the precursor lists for the source and destination IP addresses. The next hop in the source route entry is added to the precursor list of the destination route entry. Also, the next hop towards the destination node is added to the precursor list of the source node's route table entry. The intermediate node might also generate another RREP packet called Gratuitous RREP.

By the time a RREQ packet arrives at a node that can supply a route to the destination, a reverse path has been established to the source of the RREQ. As the RREP travels back to the source, each node along the path sets up a forward pointer to the node from which the RREP came, updates its timeout information for route entries to the source and destination, and records the latest destination sequence number for the requested destination. As the RREP is forwarded back towards the node which originated the RREQ message, the hop count field is incremented by one at each hop. Thus, when the RREP reaches the source, the hop count represents the distance, in hops, of the destination from the source node. The source node then uses this route to send out the queued data packets.

8.4.3.1 Gratuitous RREP

After a node receives a RREQ and responds with a RREP, it discards the RREQ. If intermediate nodes reply to every transmission of a given RREQ, the destination does not receive any copies of it. In this situation it does not learn of a route to the source node. This could cause the destination to initiate a network-wide route discovery (for example, if the source is attempting to establish a TCP session). If the destination node needs to learn the route to the source node, the source node sets a "Gratuitous RREP" flag in the RREQ packet. If, in response to a RREQ packet with the "Gratuitous RREP" flag set, an intermediate node returns

a RREP, it also unicasts a gratuitous RREP to the destination node to supply it with a route for the source node.

8.5 Route Maintenance

Once a route has been discovered for a given source-destination pair, it is maintained as long as needed by the source node. Movement of nodes within the ad-hoc network affects only the routes containing those nodes. Such a path is called an active path. Movement not along an active path does not trigger any protocol action. If the source node moves during an active session, it can reinitiate route discovery to establish a new route to the destination. When either the destination or some intermediate node moves, there could be two different actions taken, depending upon the features supported by the protocol.

The node upstream of the link break may choose to repair the link locally. This is called *local repair*. An upstream node decides to do this only if the destination was no further than *MAX_REPAIR_TLL* hops away. To repair the link break, the node increments the sequence number for the destination and then broadcasts a RREQ for that destination. If the node receives a RREP before the RREP timer expires, it adds the new route to its route table. Also, if the hop count for the new route is more than the old route, the node receiving the RREP sends out a special route error (RERR) message to the precursor nodes of this route. This message is tagged with a special flag and contains the new information about the destination. An intermediate node forwards this RERR message without deleting the route to the destination. When it reaches the source node, the node might decide to reinitiate a route discovery. On the other hand, if the node does not receive a RREP at the end of discovery period, it proceeds as follows.

If the upstream node does not decide to perform a local repair or if it does not get back a RREP for the local repair attempt, it node sends out a RERR message to the affected source node. It lists each of the destinations that are now unreachable because of the loss of the link. The upstream node then transmits the RERR (either through unicast or broadcast) to the nodes in the precursor list (if any) of the destination node. When the neighbors receive the

RERR, they mark their route to the destination as invalid by setting the hop count for the destination as i*INFINITY* and in turn propagate the RERR to their precursor nodes. When a source node receives the RERR it can reinitiate a route discovery if the route is still needed.

8.6 Local Connectivity Management

A node keeps track of its connectivity with its neighboring nodes. This information is obtained from the broadcasts sent by the neighbors. Whenever a node receives a broadcast from a neighbor, it updates its local connectivity information by creating/updating the route table entry for that node. Nodes in AODV also use a special kind of RREP message called *Hello Message* to supply connectivity information. The hello message contains the identity of the originating node and the lifetime value for the route to that node. The purpose of hello messages in AODV is similar to what standard hello packets are used for, viz. to provide connectivity information. If a node has not broadcasted any control packets within last *hello interval* time, it broadcasts to its neighbors a *hello message*, containing its own IP address and current sequence number. This message is prevented from being re-broadcasted outside the neighborhood of the node because it contains a TTL value of 1. The failure to receive any transmission from a previous neighbor in the (*allowed_hello_loss* * *hello interval*) time duration is an indication that the connectivity with the neighbor is lost. This is considered as a broken link and the node initiates processing as mentioned in Section 8.5. Broken links can also be detected using link layer feedbacks. For example, in IEEE 802.11, absence of a link layer ACK or failure to get CTS after sending RTS, even after the maximum number of retransmission attempts, indicates loss of the link to this active next hop.

8.7 Actions After Reboot

After reboot, a node would have lost its own sequence number as well as the sequence numbers for other destinations in the network. Since, the neighboring nodes may be using this node as an active next hop, this might create routing loops. To prevent this possibility, each

node on reboot waits for *delete_period* time, during which it does not send out any RREPs. If the node receives a RREQ, RREP or RERR control packet, it creates route entries as appropriate given the sequence number information in the control packet. If the node receives a data packet for some other destination, it broadcasts a RERR and resets the waiting reboot timer to expire after current time plus *delete_period*. Thus, by the time a node comes out of the waiting phase and becomes an active router again, none of its neighbors will be using it as an active next hop any more. A node's local sequence number gets updated if it receives a RREQ for itself from some other node, as the RREQ always carries the maximum destination sequence number seen en route.

8.8 Multiple Interfaces

AODV is designed to operate on nodes having multiple interfaces. It is designed to operate smoothly over wired, as well as wireless, networks. To make this possible, the interface over which packets arrive must be known to AODV whenever a packet is received. This includes the reception of RREQ, RREP and RERR messages. In AODV, whenever a packet is received, the interface on which that packet was received is recorded in the route table entry for that neighbor, along with all the other appropriate information. Similarly, whenever a route to a new destination is learnt, the interface through which the destination can be reached is also recorded into the destination's route table entry. When multiple interfaces are available, a node transmitting a RREQ message broadcasts that message on all the interfaces that have been configured for operation in the ad-hoc network.

8.9 Subnet Routing

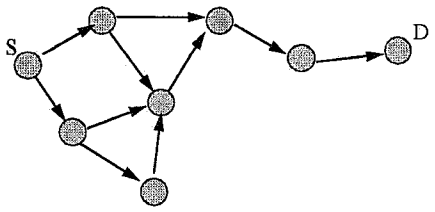
In case of subnet routing, a route to any one of the nodes in the collection is nearly as good as a route to any other node. Thus, route information to all of the nodes can be summarized by a single route table entry, and route aggregation is possible.

To work with AODV, routes to the subnet have to be assigned a destination sequence number. For a subnet, all that is needed is that one of the nodes on the subnet takes responsibility for creating and managing subnet sequence numbers. If there is a router on the subnet already, that node is the logical choice, if not, some other node has to be assigned this function as well as the function of forwarding traffic for other nodes on the subnet. The node managing the sequence number is called the subnet leader, and it must be considered the default router for all subnet nodes.

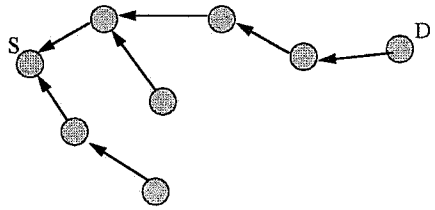
Nodes on the subnet forward RREQ to the subnet leader. The subnet leader creates a reverse route to its subnet nodes in the same way as to any other node in the network. RREP messages through any node on the subnet must be sent back to the source though the subnet leader.

8.10 Security Considerations

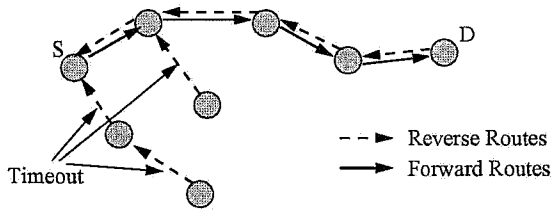
Currently, AODV does not specify any special security measures. Routing protocols, however, are prime targets for impersonation attacks. These attacks include transmitting RREPs with false routing information resulting in malicious denial of service to the destination and/or malicious inspection and consumption of traffic intended for delivery to the destination. Additionally, wireless transmission is inherently insecure. Packets are received by anyone within the transmission range, and if they are not encrypted, they can also be read by anyone. To protect against these attacks, authentication techniques involving generation of unforgeable and cryptographically strong message digests or digital signatures can be used. It is expected that, in environments where security is an issue, IPSec authentication headers will be employed along with the necessary key management to distribute keys to the members of the ad-hoc networks using AODV.



(a) Propogation of Route Request



(b) Reverse Path Formation

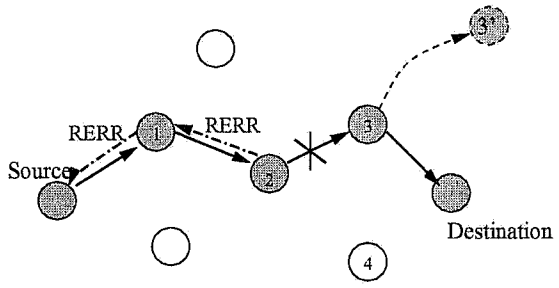


(c) Forward Path Formation

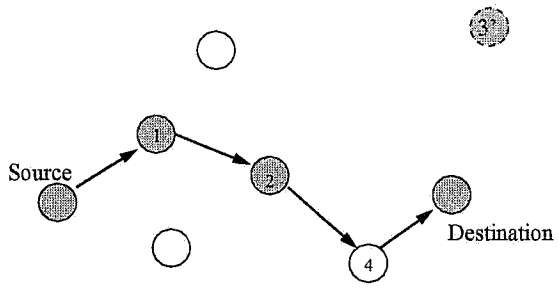
Figure 8.1 Route Discovery in AODV

8.11 Example of AODV in Action

Figure 8.1 illustrates the route discover procedure in AODV. Source node S requiring route to destination node D broadcasts a RREQ message. The RREQ message is forwarded by the intermediate nodes (Figure 8.1a). Each node receiving the RREQ message creates a *reverse route* to the source as shown in Figure 8.1b. Destination node D identifies that the RREQ message is intended for it. It generates a RREP message with the latest known route information and unicasts it along the reverse route towards source S. Every node forwarding RREP creates a *forward route* to the destination node D. This is depicted in Figure 8.1c.



(a) Broken Link detection



(b) Newly Discovered Route

Figure 8.2 Route Maintenance in AODV

The route maintenance procedure of AODV is illustrated in Figure 8.2. The original path from the source to the destination is through nodes 1, 2 and 3. Suppose node 3 then moves to location 3', causing a break in the connectivity with node 2. Node 2 notices this break and sends a RERR to node 1. Node 1 marks this route as invalid and then forwards the RERR to the source. On receiving the RERR, the source node determines that it still needs the route, and so it reinitiates a new route discovery. Figure 8.2b shows the new found route though node 4.

CHAPTER 9

AODV Implementation

We begin by describing some prior implementations of the AODV routing protocol [45, 46, 47, 48]. The AODV-UCSB [45] is the implementation of AODV developed at the University of California at Santa Barbara. It achieves the on-demand functionality by copying every packet from the kernel space to the user space. By matching these packets against the entries in the user-space route cache, AODV-UCSB identifies the packets for which there exists no route and initiates route discovery. There are two obvious drawbacks of this approach: every packet has to cross the address space twice, inducing much overhead, and for every packet the routing is done twice as well, once in the user-space and once again in the kernel. The implementation of AODV developed at Uppasala University (AODV-UU) [46] also suffers from the similar drawbacks. Kernel-AODV [47] implementation from NIST has been implemented as a single kernel module. This approach is not very desirable, as complex protocol processing can slow the kernel, hog the memory, and crash the whole system if there is a serious bug in the routing protocol.

The AODV routing daemon (AODV-UIUC) implemented by us does not suffer from any of these drawbacks. AODV-UIUC is much simpler, cleaner, and more efficient than any of the existing implementations of AODV routing protocol.

This chapter describes the implementation details of the AODV-UIUC routing protocol. The protocol has been implemented as a user-space routing daemon on the Linux operating

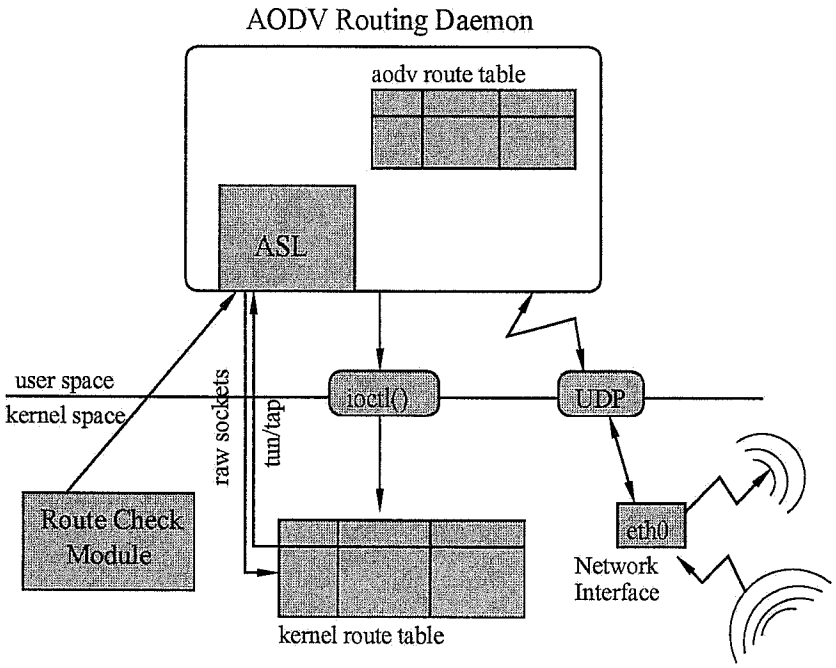


Figure 9.1 AODV Routing Daemon

system. The current implementation version supports all the features of AODV draft 10 [44]. The implementation is done in C++ and on Linux kernel 2.4.17.

9.1 Overview of the Implementation

Figure 9.1 gives the design of the user space AODV implementation. The AODV routing daemon maintains a copy of the route table in the user space called *aodv route table*, which it uses to modify the main kernel route table.

The routing daemon utilizes the services offered by the Ad-hoc Support Library (ASL) described in Chapter 7. The routing daemon starts by opening a route request with the ASL. This action returns the *tun* file descriptor, used by ASL to read packets from the tun device, to the routing daemon. The *tun* device is configured by ASL so that a data packet with no valid kernel route for it is sent on this tun device. Thus whenever an unrouted packet is received at tun device, the routing daemon gets to know about this. It calls the *read_route_request()*

function of ASL to decide if a route discovery needs to be initiated for the destination node in the received packet. ASL maintains a queue of all the packets received at tun device and for which the route discovery procedure is in progress. If the queue already contains a packet destined to that particular destination node, then a new route discovery is not needed. On the other hand, if there is no pending packet in the ASL queue, the routing daemon is instructed by ASL to start a new route discovery.

The AODV routing daemon broadcasts a RREQ packet and sets up a timer to wait for the RREP. It uses the *expanding ring search* technique to prevent uncontrolled dissemination of RREQ messages. If the timer times out, the routing daemon rebroadcasts the RREQ message with an increased TTL value. The rebroadcast of RREQ can be done up to *RREQ_RETRIES* number of times. If a RREP is received within allocated number of RREQ retries attempt, the routing daemon inserts a new route for the destination in aodv route table as well as kernel route table. It then returns a "route found" status to the ASL. ASL then reinjects all the packets, queued for that particular destination, into the kernel using raw sockets. On the other hand, if a RREP is not received within the timeout limit, the routing daemon returns a "no route" status to ASL. ASL then discards all the packets queued for that destination.

The AODV Routing daemon implements *hello messages* to keep track of neighborhood connectivity. Whenever the daemon receives the first *hello message* from one of its neighbors, it starts a broken link timer for that neighbor with the timer interval as *allowed_hello_loss* * *hello_interval*. This timer is reset every time a new aodv control packet is received from that neighbor. If this timer expires, then the link to the neighbor is assumed to be broken. The IP address of this neighbor as well as any other destination node using this neighboring node as the next hop, is then put into a list of locally repairable destination nodes. If a data packet is received for any of these destination nodes in near future, a local repair attempt is made to discover another route. If the local repair attempt fails at a node, it generates a RERR message and sends it to all the nodes in the precursor list of the destination node.

9.2 Features Supported

The current version of the AODV routing daemon is fully compliant with the Version 10 of AODV draft [44]. Following are the features supported in the current implementation of AODV:

- Route Request/Route Reply discovery
- Gratuitous Route Reply
- Route Reply Acknowledgement
- Route Error Messages
- Hello Messages
- Local Repair
- Unidirectional Links
- Actions after reboot.

9.3 Implementation Details

This section provides more details regarding the implementation of AODV routing daemon. The implementation is divided into a number of modules. Figure 9.2 illustrates all the major AODV modules and their interaction with the kernel. The current implementation uses the system services support provided by the Ad-hoc Support Library (ASL) to implement reactive ad-hoc routing functions. The AODV daemon has been implemented as a sequential program and it does not create any new threads or processes.

9.3.1 Ad-hoc Support Library API

Our AODV implementation utilizes the Ad-hoc Support Library API functions to capture un-routable data packets, queue these packets and reinject them back into the kernel once the

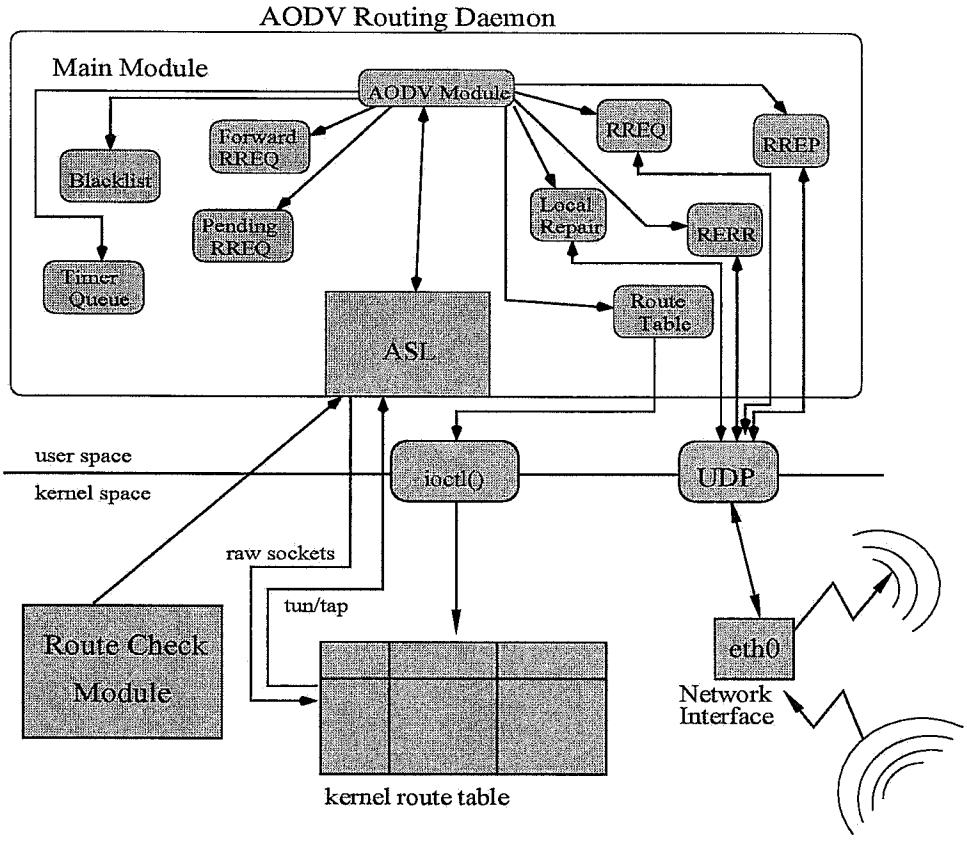


Figure 9.2 Modular Design of AODV Routing Daemon

desired route has been discovered. The following are the ASL API functions used by the AODV routing daemon.

- `int open_route_request();`

  Opens and returns *tun* file descriptor to the routing daemon.

- `int read_route_request(int fd, struct route_info *r_info);`

  Provides relevant information to the routing daemon in `struct route_info*` to start the route discovery.

- `int route_discovery_done(addr_t dest, int result);`

  Returns the result of route discovery (success or failure) to the ASL.

- `int query_route_idle_time(addr_t dest, int valid_flag );`

  Returns the idle time for the given destination.

- `int close_route_request(int fd);`

  Used to shut down all communication with ASL.

More details about the functionalities offered by these API functions are described in Chapter 7.

9.3.2 Main Data Structures

Our AODV implementation maintains the following data structures. Only private members are shown for every class.

9.3.2.1 Routing Table Entry

The *rtable_entry* class defines the structure of a route table entry. The *last_hop_count* field is needed to prevent uncontrolled dissemination of RREQ messages.

```
class rtable_entry{
    u_int32_t          dest_ip;        /* ip address of the destination node */
```

```
    u_int32_t         dest_seq_num;  /* destination sequence number for dest_ip */
    unsigned int      interface;     /* network interface for this route */
    u_int8_t          hop_cnt;       /* number of hops to the destination */
    u_int8_t          last_hop_cnt;
    u_int32_t         next_hop;      /* ip address of the next hop towards dest_ip */
    list<u_int32_t>   precursors;    /* list of precursor node ip addresses */
    u_int64_t         lifetime;      /* lifetime of this route entry */
    u_int32_t         routing_flags; /* routing flags for the entry */
};
```

9.3.2.2 Routing Table

The AODV Route table is defined by class *routingTable*. It contains a *map* of *rtable_entry* to store routes for the destination nodes. This *map* is keyed by the IP address of destination node.

```
class routingTable{
    map<u_int32_t,rtable_entry>rTableMap;
};
```

9.3.2.3 Route Request

The structure of route request (RREQ) message is defined by the *RREQ* class.

```
class RREQ{
    u_char type:8;          /* message type, set to AODV_RREQ */
    u_char J:1,             /* reserved for multicast */
    R:1,                    /* reserved for multicast */
    G:1,                    /* gratuitous RREP flag */
    reserved1:5;
    u_char reserved2:8;
    u_int8_t hop_cnt;       /* number of hops from source to the current node processing RREQ */
    u_int32_t rreq_id;      /* id of this RREQ */
    u_int32_t dest_ip;      /* IP address of destination for which route is needed*/
    u_int32_t dest_seq_num; /* sequence number of destination node */
    u_int32_t src_ip;       /* IP address of the node generating RREQ */
```

```
    u_int32_t src_seq_num;  /* sequence number of the source node */
};
```

G is the gratuitous RREP flag and, if set, indicates that a gratuitous RREP should be unicast to the destination IP address. Reserved fields are currently ignored.

9.3.2.4 Route Reply

The *RREP* class defines the structure of a route reply (RREP) message.

```
class RREP{
    u_char type:8;          /* message type, set to AODV_RREP */
    u_char R:1,             /* reserved for multicast */
    A:1,                    /* RREP acknowledgement flag */
    reserved1:6;
    u_char reserved2:3,
    prefix_size:5;
    u_int8_t hop_cnt;       /* number of hops from source to the destination */
    u_int32_t dest_ip;      /* IP address of destination for which route is supplied */
    u_int32_t dest_seq_num; /* sequence number of destination node */
    u_int32_t src_ip;       /* IP address of the node originating RREQ */
    u_int32_t lifetime;     /* lifetime of the supplied route */
};
```

The *A* field is set if the link over which the RREP is being forwarded is unreliable or unidirectional. A node receiving RREP with this flag set, sends an acknowledgement back to the node from which the RREP was received.

9.3.2.5 Unreachable Destination Entry

Class *unrchDest* defines the structure of a single entry in the route error message.

```
class unrchDest {
   u_int32_t    dest_ip;
   u_int32_t    dest_seq_num;
};
```

9.3.2.6 Route Error

The structure of a route error message is defined by *RERR* class. It contains a list of *unrchDest* objects. No delete flag is set when a node has performed local repair of a link, and upstream nodes should not delete the route in this case.

```
class RERR{
    u_int8_t          type;              /* message type, set to AODV_RERR */
    u_int8_t          N:1,               /* No delete flag */
    reserved1:7;
    u_int8_t          reserved2;
    u_int8_t          dest_cnt;          /* number of unreachable destinations */
    list<unrchDest>   unreachable_dest;  /* list of unreachable destinations */
};
```

9.3.2.7 Route Reply Acknowledgement

This class defines the structure of Route Reply Acknowledgement (RREP ACK) message. It is sent by a node when *A* flag is set in RREP.

```
class RREP_ACK{
    u_char    type:8;       /* message type, set to AODV_RREP_ACK */
    u_char    reserved:8;
};
```

9.3.2.8 Pending Route Request List Entry

This class defines the structure of a single RREQ entry waiting for the RREP. *retries* and *ttl* fields are required for implementing expanding ring search technique.

```
class rreq_list_entry{
    RREQ       rreqOb;   /* copy of the RREQ object broadcasted */
    u_int32_t  retries;  /* number of times RREQ is broadcasted with maximum TTL value */
    int        ttl;      /* current value of TTL used */
};
```

9.3.2.9 Pending Route Request List

An instance of this class is used to store all pending RREQs at the source node for which a RREP is anticipated. This is needed so that RREQs can be transmitted again in case RREPs were not received on time. It consists of a list of *rreq_list_entry* objects each one storing information about a single pending RREQ.

```
class rreqPendingList{
    list<rreq_list_entry>    rreq_list;
};
```

9.3.2.10 Forward Route Request List Entry

To avoid reprocessing RREQs, a node stores information about recently processed RREQs. This class defines the structure of information stored for each received RREQ,

```
class fw_rreq_entry {
    u_int32_t    src_ip;   /* IP address of node originating RREQ */
    u_int32_t    rreq_id;  /* unique ID of the RREQ */
    u_int64_t    lifetime; /* lifetime associated with this entry */
};
```

9.3.2.11 Forward Route Request List

This list is used to store the recently processed RREQ messages to avoid reprocessing them again.

```
class fwRreqList{
    list<fw_rreq_entry>    fw_rreq_list;
};
```

9.3.2.12 Local Repair Entry

This class stores the information about a single destination node for which local repair can be done. Storing IP address of the destination node is enough to perform a local repair for that node.

```
class local_repair_entry{
    u_int32_t    dest_ip;   /* IP address of the locally repairable node */
};
```

9.3.2.13 Local Repair List

This stores the list of all the destination nodes for which a local repair can be done.

```
class localRepair{
    list<local_repair_entry>    local_repair_list;
};
```

9.3.2.14 Black List Entry

If a node detects a unidirectional link to a neighboring node, it adds that node to a list called "blacklist". Any RREQ received from a node in the blacklist set is dropped. This entry defines the structure of all the information stored for a single node in the blacklist.

```
class blacklist_entry {
    u_int32_t    dest_ip;   /* IP address on the blacklist node */
    u_int64_t    lifetime;  /* timeout for this entry */
};
```

9.3.2.15 Black List

This is the list where data about all the blacklist nodes are stored.

```
class blacklist{
    list<blacklist_entry>    bList;
};
```

9.3.2.16 Timer Entry

AODV uses same timer handling mechanism as used in DSDV. This class defines the structure of a single timer entry.

```
class timer{
    u_int64_t     timeout; /* timeout of the timer entry */
    timer_hfunc_t handler; /* handler function for the timer */
    void          *data;   /* data passed to the handler function */
};
```

9.3.2.17 Timer Queue

This is the queue of the timer maintained by AODV routing daemon.

```
class timerQueue{
    list<timer>    timerQ;
};
```

9.3.3 Modules

The AODV routing daemon consists of eleven main modules. These modules cooperate to implement different AODV features. Following is an in-depth description of each of these modules.

9.3.3.1 Main Module

The main module ties together all the other modules of the AODV routing daemon. It parses command line arguments and uses these to initializes global flags. It also declares other global data structures like the routing table, pending route request list, forward route request list, local repair list, black list and timer queue. It creates an instance of *aodv* class and calls its *aodv_daemon()* function, which is the main starting point of the AODV routing daemon.

9.3.3.2 Aodv Module

This is the module which defines the main flow of control inside AODV daemon. The processing starts at function *aodv_daemon()*. It contains an array of file descriptors and their associated handler functions. The array of descriptors includes an aodv socket, which is used to transmit and receive all AODV control messages, and the tun descriptor returned by ASL.

open_route_request() function of ASL is called to obtain tun file descriptor. These descriptors are obtained/created inside *aodv_init()* function, which is called from *aodv_daemon()*.

After initialization, the module registers to listen for various system signals. It then starts a reboot timer and a periodic refresh timer. The first is to ensure that the routing daemon waits for *DELETE_PERIOD* time just after reboot, before sending out any RREP. This is needed to avoid possibility of any routing loops. The second timer is used to periodically check for stale entries in different tables and lists maintained by the AODV routing daemon. This includes the route table, forward route request list and the list containing blacklist nodes. The module then enters into main routing daemon loop, waiting on aodv socket and tun file descriptor, returned by ASL.

If a data packet for some destination is received at tun descriptor, *aodv* module calls *read_route_request()* function of ASL to decide if a route discovery needs to be initiated. If ASL has not already queued a packet for this destination, it signals *aodv* module to start a route discovery. ASL returns source IP, destination IP and protocol field of the data packet to *aodv* module in *struct route_info*. If the packet was generated locally, *aodv* module sends out a RREQ packet. It creates a RREQ packet utilizing interface functions of *RREQ* class. This RREQ packet is added to the pending list of RREQs. The *aodv* module starts a timer to wait for the RREP to come. The TTL of the outgoing RREQ is set using the expanding ring search technique. This RREQ packet is then transmitted to the local broadcast address. When a RREP is not received as expected, *aodv* module retransmits the RREQ up to a maximum of *RREQ_RETRIES* times. If a RREP is received within this limit, it calls *route_discovery_done()* API function of ASL with *ASL_ROUTE_FOUND* result to signal that a route has been successfully discovered. However, if no RREP is received, it returns a status of *ASL_NO_ROUTE* to the ASL.

On the other hand, if the source of the data packet was some other node in the network, the *aodv* module checks to see if the destination is in the local repair list. If yes, then the link to the destination is recently broken. In such a scenario, the module tries to repair the link locally. It generates a new RREQ message for the destination and broadcasts it to its neighboring node.

It also starts a timer to wait for the RREP for this local repair attempt. If the timer expires, the node generates a RERR message containing an entry for the destination and sends it to the nodes in the precursor list of that destination. However, if a RREP is received but the newer hop count to the destination is larger than the old value, the module sends out a RERR to the precursors of the destination node with the *N* flag set. If the node is not in the local repair list, a *ASL_NO_ROUTE* result is sent to ASL.

If the module receives data on *aodv* socket, it first examines the source address of the received packet. All the packets from the local node are discarded. If the received data is not generated locally, the module looks at the first 8 bits of the received packet to determine the type of the message. The message type should be one of the*AODV_RREQ, AODV_RREP, AODV_RERR* and *AODV_RREP_ACK*. Further processing of these messages are handled by the corresponding modules.

Any time a new route is added to the aodv route table in the form of a reverse route, a forward route or on the receipt of a hello message, the aodv daemon always returns a *ASL_ROUTE_FOUND* to the ASL, so that any pending packets for the destination would be routed as soon as possible.

9.3.3.3 RREQ Module

A received RREQ message is handed over to this module for further processing. This module first checks to see if the RREQ was received over a unidirectional link. If yes, the packet is discarded. It calls the *neighborUpdate()* function of *routeTable* class to create/update an entry for the node from which the RREQ is received. It looks for the RREQ in the forward route request list. If the RREQ is already there, the current RREQ is ignored. If the RREQ is not already processed, a new entry for the RREQ is created in the forward RREQ list. The module also creates a reverse route to the source of the RREQ. If the RREQ was destined for the local node, it calls the interface function of *RREP* class to generate a RREP. Otherwise, a RREP is generated if the node has a valid route to the destination. The generated RREP is sent to the next-hop node towards the source of RREQ. The module also generates a gratuitous RREP

message if the incoming RREQ had its *G* flag set. In all other cases, the module rebroadcasts the received RREQ after incrementing the hop count.

9.3.3.4 RREP Module

All the received RREP messages are passed on to this module. This module is responsible for further processing of these RREP messages. The module sends out a *RREP-ACK* message if the received RREP has its *A* flag set. It creates/updates the entry for the neighboring node from which the RREP was received. It also creates a forward route for the destination node in the received RREP message. If the local node is not the intended receives of the RREP, it unicasts the RREP to the next-hop along the path to the source node after updating the precursor lists of source and destination route table entries. If the RREP is meant for the node itself, the node does not do any more processing.

9.3.3.5 RERR Module

This module is responsible for handling RERR messages received by the AODV routing daemon. This module first creates a *RERR* object from the received packet buffer. It also creates/updates the neighboring node entry. If the received RERR message has the *N* flag set, then the module transmits the RERR message to the list of precursor nodes in the route table entry for the destination IP in the RERR message. The route for the destination node is not invalidated. On the other hand, if the *N* flag is not set in the RERR message, the module creates another RERR message. This new message consists of those destinations in the received RERR for which there exists a corresponding entry in the aodv routing table and has the transmitter of the received RERR as the next hop. The newly created RERR message is transmitted to the list of precursor nodes of the destinations in the new RERR message.

9.3.3.6 Routing Table Module

This module is responsible for updates to the aodv route table as well as to the kernel route table. It provides *neighborUpdate()* function to create or modify the route table entry to the

neighboring node every time an AODV control message is received. It also implements functions to create or update reverse and forward routes. The *refreshEntries()* member function of this module is called whenever the periodic refresh timer expires. For any route table entry which has expired, the module calls *query_route_idle_time()* API function of ASL to determine the idle time for this route entry. If the idle time is equal to or more than the*ACTIVE_ROUTE_TIMEOUT* value, then this route entry is made invalid by modifying the hop count and the sequence number values. However, if the idle time is less than the *ACTIVE_ROUTE_TIMEOUT*, then the lifetime of the route entry is updated.

9.3.3.7 Pending Route Request Module

Whenever a node broadcasts a RREQ message, it stores that message in a list of RREQs called *rreqPendingList*. The Pending Route Request module is responsible for maintaining this list. Every entry of the list stores the number of times that RREQ has been broadcasted (retries value), the current value of TTL and the actual RREQ packet. An entry from this list is removed when the RREP for the destination is received or when maximum number of RREQ retries have been attempted.

9.3.3.8 Forward Route Request Module

To avoid processing a RREQ multiple times, whenever a node receives a new RREQ packet, it stores the source IP address and the route request ID in a list. This list is called forward route request list (*fwRreqList*). Each entry in this list is assigned a lifetime equal to PATH_TRAVERSAL_TIME. When a RREQ is received, a node first checks if it has a similar RREQ entry in the forward route request list. If yes, the received RREQ is discarded. The periodic refresh timer monitors this list for stale entries and removes them..

9.3.3.9 Local Repair Module

Nodes in an ad-hoc network monitor their neighbors for continued connectivity information. The current implementation of AODV uses hello messages to supply connectivity information.

When a node first gets a hello packet from one of its neighbor, it initiates a broken link timer for that neighboring node. This timer is reset whenever any packet is received from that neighbor. If this timer expires, the link to the neighbor is assumed to be broken. The neighboring node is then put into the list of locally repairable nodes. All the other destinations nodes with their next hop as the neighboring node are also put into the list of locally repairable nodes. This list is called the *localRepair* list, and is managed by the Local Repair module. If a data packet is received for any of the locally repairable nodes within the near future, the node transmits a RREQ to repair the broken link.

9.3.3.10 Blacklist Module

An ad-hoc network may contain unidirectional links. It is possible that the transmission of a RREP fails over a unidirectional link. Thus the source node will not receive the RREP. The same scenario might happen with every RREQ retransmission attempt also. This is possible even if a bidirectional route exists between the source and the destination, because in AODV any node acts only on the first RREQ with the same RREQ ID and source IP address.

To prevent this problem, the Blacklist module maintains a list (called *blacklist*) of neighboring nodes to which unidirectional links exist. Whenever transmission of a RREP message fails, the next hop of the failed RREP message is added to the "blacklist". Such a failure is detected by the absence of a route reply acknowledgement (RREP-ACK) message. A node ignores all RREQs received from any node in its blacklist set. The periodic refresh timer examines entries in this list and removes then after *BLACK_LIST* timeout.

9.3.3.11 Timer Queue Module

The AODV routing daemon uses the *timerQueue* module to maintain different timers. This module provides a universal handler function for all the timers, called *scheduleTimer()*. This function is called when any of the timers expires and it in turn calls the handler functions of all the expired timers. Last, it sets the system timer to the next unexpired timer value. The *set_timer()* function is the generic function which is used to add new timer entries to the timer

queue. It also resets the system timer (if needed) to the smallest unexpired timer value. The Timer Queue module also provides functions to remove RREP and RREP_ACK timer from the queue of the timers.

CHAPTER 10

Interoperability Support Services for Routing Protocols in Hybrid Ad-hoc Networks

10.1 Motivation

Currently a number of routing protocols exist for mobile ad-hoc networks. Some of these protocols are proactive in nature, maintaining routing information for all the other nodes in the network, while some are on-demand based, discovering routes only when needed. There is a third category of protocols called *hybrid* protocols. These protocols employ both reactive and proactive schemes to come up with better optimized routing protocols for MANET.

No single category of MANET routing protocols qualify for the one-size-fits-all proverb. While proactive routing protocols guarantee immediate route availability, they do not scale well. Also broadcast of periodic route update messages consume a considerable amount of network bandwidth. On the other hand, reactive routing protocols have a significant amount of initial delay involved for route determination. Also most of these reactive protocols employ global flooding of route request to discover a route. This could require significant control traffic and hence could consume a considerable amount of scarce wireless bandwidth. However, these protocols scale well as the size of the routing control messages is not dependent on the size of the network.

We believe that the two main categories of ad-hoc routing protocols, proactive and reactive, are going to co-exist in the forth coming future. Thus, some kind of mechanism is needed to

enable them to inter-operate with each other. We could imagine an ad-hoc network where some of the nodes are running proactive routing protocols and others are running reactive routing protocols. We call this a *hybrid ad-hoc network* and the nodes in such a network are classified as *reactive nodes* and *proactive nodes* based on the type of routing protocols being run at these nodes. In a *hybrid* ad-hoc network, it is required that the two categories of nodes be able to talk to each other for the correct and efficient functioning of the network. For example, a *hybrid* network could be partitioned if *reactive nodes* are not able to communicate with *proactive nodes*. Thus, it is desirable to develop schemes for enabling inter-operability between the reactive and proactive ad-hoc routing protocols. Currently, no such scheme exists in the MANET community.

We have conducted research to develop mechanisms for supporting inter-operability between proactive and reactive ad-hoc routing protocols. This lead to the development of Interoperability Support Services (ISS) Regime for hybrid ad-hoc networks. ISS can be utilized to enhance existing reactive and proactive ad-hoc routing protocols to enable them to talk to each other with very little modifications. The idea is to provide a set of API functions which can be used by the existing proactive and reactive ad-hoc routing protocols to achieve inter-operability with each other. The ISS approach to provide inter-operability is both elegant and clean as it requires minimal modifications to the ad-hoc routing daemon's code and does not require any modification to the existing kernel code. This chapter explains the main design principle behind Interoperability Support Services (ISS).

10.2 Communication in a Hybrid Ad-hoc Network

An hybrid ad-hoc network consists of nodes, some of which might be running a proactive routing protocol (*proactive nodes*) while others might be running a reactive routing protocol (*reactive nodes*). Nodes in a *hybrid network* are divided into groups of proactive nodes. Each such group is called a *proactive island*. A single *proactive island* comprises of all those proactive nodes which are capable of communicating with each other using routes consisting of only proactive nodes. A *proactive island* might just consist of a single proactive node. These *proactive*

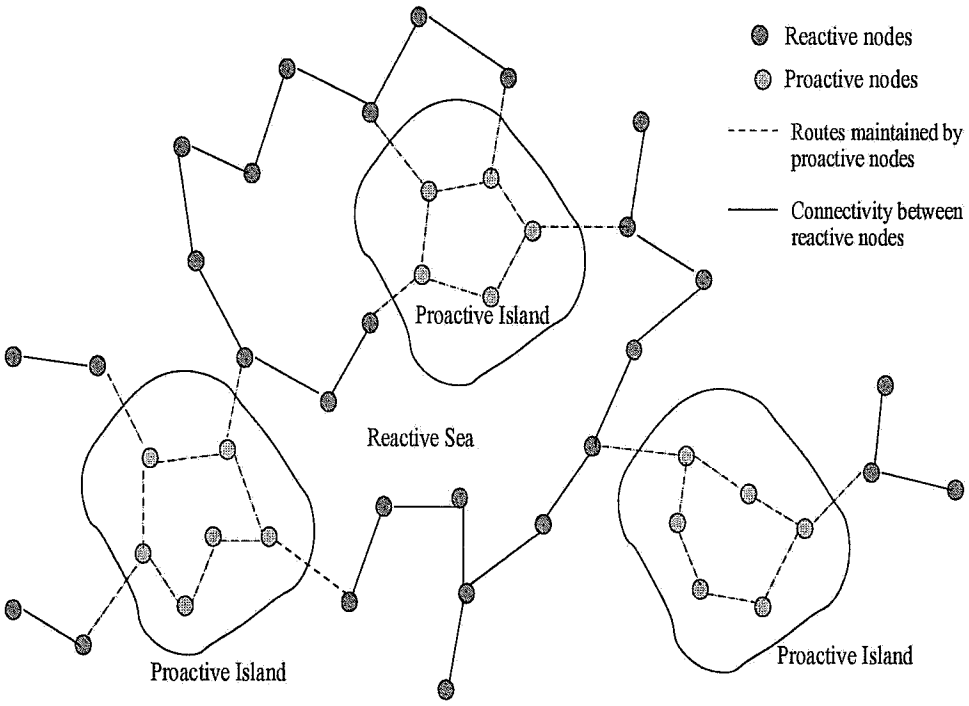


Figure 10.1 An example of hybrid ad-hoc network

islands are scattered in a pool of reactive nodes which we call a *reactive sea.* Connectivity between two *proactive islands* is achieved by intermediate reactive nodes in the *reactive sea.* Figure 10.1 illustrates an example of a *hybrid ad-hoc network* depicting *proactive islands* and *reactive sea.*

For communication to be possible in such a hybrid network, a routing path might need to pass though the nodes in the *reactive sea* and in the *proactive islands.* Special mechanisms are needed to discover and maintain such routing paths as nodes on such a routing path do not understand each other's routing protocol. Also, when a link breaks within the network, the broken link notification might need to be sent to the nodes in multiple *proactive islands* through the intermediate nodes in the *reactive sea.* This is the problem of route maintenance. Thus, following are the two main issues which need to be addressed to enable communication within a hybrid ad-hoc network:

- How to discover a route passing through *proactive islands* and *reactive sea.*
- How to maintain these routes efficiently.

10.3 Design and Mechanism

Proactive nodes in an island maintain route tables entries for all the other proactive nodes in the same island. Every proactive node also maintains route table entries for all the reactive nodes which are one-hop away from any of the nodes in the proactive island. These entries are created, when proactive nodes receive hello messages (see Chapter 2) from the neighboring reactive nodes, and are marked with *REACTIVE* flag to differentiate them from route table entries for the proactive nodes. The design of Interoperability Support Services (ISS) adds three new message types. These are *route discovery notification* (RD_NOTIFY), *route found notification* (RF_NOTIFY) and *route lost notification*(RL_NOTIFY). The two main data structures maintained by ISS are *route reply table* and *notification table.* ISS also maintains a queue in the user-space to store data packets for which routes need to be discovered.

This section presents design ideas for the route discovery and route maintenance procedures employed by ISS to discover and maintain routes in an hybrid ad-hoc network.

10.3.1 Route Discovery in a Hybrid Network

If a proactive node needs to send a data packet to another proactive node within its own island or to a reactive node in the one-hop neighborhood of its proactive island, it already has a route to the destination node. It uses this route to send out the data packet. On the other hand, if the proactive node needs to communicate with a node for which it does not have a route, there should be some kind of route discovery procedure which is initiated. To start this, the proactive node uses a new kind of message called *route discovery notification* (RD_NOTIFY) message. The node sends out this message to all the reactive nodes in the one-hop neighborhood of the proactive island. A *route discovery notification* message contains the address of the destination node for which the route needs to be discovered. A reactive node, on receiving such a notification message starts a route discovery procedure for the destination IP address specified in the *RD_NOTIFY* message, if it does not already have a route to the destination. It broadcasts a route request (RREQ) message. This message might need to be passed through both proactive and reactive nodes, to reach the destination or an intermediate node with a route for the destination.

Every proactive node maintains a *route reply table*. Every entry in the route reply table contains the IP address of the destination node, list of IP addresses of the source nodes which require a route to that destination, and the list of IP addresses of the next-hops to these source nodes. When a proactive node receives a route request message from one of the reactive nodes (this is determined by the *REACTIVE* flag in the route table entry), it creates an entry in its *route reply table*. The source and destination IP addresses are obtained from the route request packet. The next-hop field is set to the IP address of the node from which the RREQ was received. If the node does not have a route for the destination node, it modifies the received route request, to make the source address as its own IP address, and sends out this route request to all the reactive nodes in the one-hop neighborhood of its proactive island. This can

be done as every node in the proactive island maintains route table entries for all the one-hop reactive neighbors. This action is taken by a proactive node every time a route request crosses the boundary of *reactive sea* and the *proactive island*. When a proactive node receives a route request from another proactive node, it just forwards it to the next hop along the targeted reactive node. When the route request passes through the reactive nodes, they create *reverse routes* [44] to the source in the route request packet and broadcast the route request if needed. Thus the reactive nodes bordering the *proactive island* will have reverse routes to the proactive nodes in the island.

The route request finally reaches either the destination node or a node with a "fresh enough" route to the destination. If this node is a proactive node, it looks for an entry in its *route reply table* for this destination node. It then sends out a *route found notification* (*RF_NOTIFY*) message to every node in the next-hop list for this destination node. The route found notification message also contains the IP address of the source node to which a route reply should be sent. A reactive node, upon receiving this notification message, creates a route entry in its route cache for the destination node, setting the next-hop field to the IP address of the node from which the notification message is received. It then creates a route reply for the destination node and sends it to the source node specified in the *RD_NOTIFY* message. On the other hand, if the node having a route to the destination is a reactive node, it creates a route reply message and sends it to the next hop on the path to the source node, specified in the route request message. Note that this source IP address may be different from the IP address of the node which originated the route request, as proactive nodes along the route request path modify the source address in the route request message.

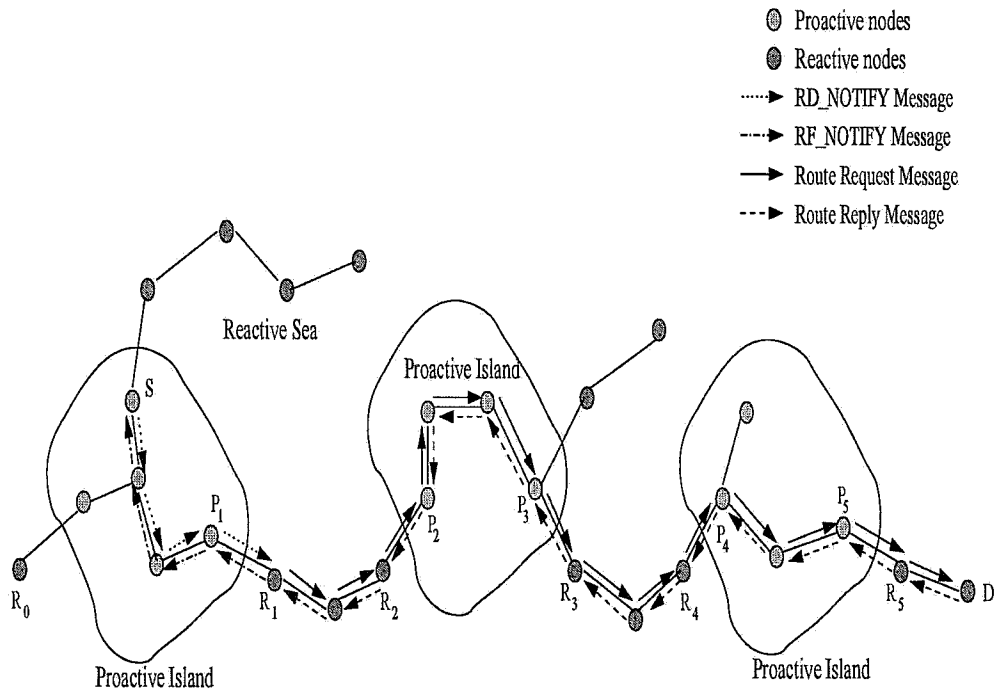
When a proactive node receives a route reply message, it checks its *route reply table* for an entry to the destination node. If such an entry is found, it sends out the route reply to every node in the list of source nodes stored in the entry. Before transmitting the route reply, the source IP address in the route reply is changed to the source IP address stored in the *route reply table* entry. Once the route reply has been forwarded to a source node, the corresponding entry in the *route reply table* is deleted. Eventually, the route reply reaches the reactive node,

which originated the route request. The reactive node then creates a route table entry for the destination node in its route cache.

Every reactive node maintains a *notification table*. Each entry in the table stores the IP address of the destination node, a list of IP addresses of the proactive nodes which have sent route discovery notification (*RD_NOTIFY*) message for this destination node, and the list of IP addresses of the next-hops to these proactive nodes. When a reactive node receives a route reply for the destination node, it generates a *route found notification* (*RF_NOTIFY*) message and transmits it to all the proactive nodes in the notification table entry for this destination node, by forwarding it to the next-hop node along the path to the proactive node. The next-hop proactive node receiving *RF_NOTIFY* message establishes a route to the destination node, setting the reactive node, from which the *RF_NOTIFY* message was received, as the next-hop towards the destination. This route is propagated to other nodes in the proactive island, and eventually the source node seeking a route to the destination gets this route. All the data packets stored for this destination are transmitted on the newly discovered route. If at the reactive node, the notification table does not contain an entry for the destination IP address in the route reply packet, then the route discovery was initiated by locally generated data packets. In such a case the new route is used to transmit all the locally stored data packets. Even if an entry for the destination node exists in the *notification table*, there could be some locally generated data packets for that destination. Thus, upon receiving the route reply, the reactive node always checks if it has any data packets queued locally for the destination IP address and transmits them using the newly discovered route.

10.3.2 Example of Route Discovery

Figure 10.2 depicts an example of the route discovery process in an hybrid ad-hoc network. Source node S generates a data packet for destination node D. It does not have a route to D. It generates a *RD_NOTIFY* message and sends it to all the reactive nodes (including R_0 and R_1) in one-hop neighborhood of this proactive island. This example illustrates the route discovery done by node R_1. On receiving the *RD_NOTIFY* message, node R_1 creates an entry



(a) Route discovery in hybrid ad-hoc network

Destination	List of Sources	List of Next Hops
D	S	P_1

(b) Notification table at R_1

Destination	List of Sources	List of Next Hops
D	R_1	R_2

(c) Route reply table at P_2

Destination	List of Sources	List of Next Hops
D	P_2	R_4

(d) Route reply table at P_4

Figure 10.2 Route discovery process in a hybrid ad-hoc network

in its *notification table* for destination node D, with the source IP address set to the IP address of node S and the next-hop address set to the IP address of node P_1 (Figure 10.2b). R_1 then generates a route request (RREQ) message for destination D and broadcasts it. Figure 10.2a depicts a typical path taken by the route request to travel to the destination D. Node P_2 receives this route request message. It creates an entry in its *route reply table* for destination D. The source IP address is set to the received source IP address in the route request (R_1). The next-hop address is set to R_2 (Figure 10.2c). P_2 also changes the source IP address in the route request to its own IP address. It then transmits the RREQ to all the reactive nodes in the one-hop neighborhood (including R_3). R_3 re-broadcasts the route request. Node P_4 receives this route request from node R_4. It creates an entry for the destination D in its *route reply table* setting the source IP address to the received source IP address in the route request, which is P_2. The next-hop address is set to R_4 (Figure 10.2d). Node P_4 also changes the source address in the route request to its own IP address and forwards it to node R_5, along with other reactive nodes. Eventually, the destination node D receives the route request.

Destination D generates a route reply and sends it to node R_5. The source IP address in the route reply (RREP) is set to P_4 (same as the source IP address in the received route request message). Node R_5 has a reverse route to P_4 and forwards the RREP packet on that. Node P_4 on receiving the RREP, consults its *route reply table* to find an entry for the destination D. An entry is found with source IP as P_2 and next-hop as R_4 (Figure 10.2d). It modifies the RREP to change the source IP to P_2 and sends it to the node R_4. Node R_4 has a reverse route to P_2 and it uses that to forward the route reply packet to P_2. On receiving the route reply packet, node P_2 looks at the entry for destination D in its *route reply table* (Figure 10.2c). Using that it changes the source IP of RREP to R_1 and forwards this RREP to node R_2. Finally node R_1 receives this reply. It consults its *notification table* to find an entry for destination D (Figure 10.2b). It then transmits a *RF_NOTIFY* message to the source node S by forwarding it to the next-hop node P_1. Upon receiving this message, node P_1 creates a route for destination D in its route table with R_1 as next hop. It then circulates this route to all the nodes in its *proactive island.* When the *RF_NOTIFY* message reaches source node S, it knows that a route

to the destination D has been discovered. Node S eventually gets this route, as it is propagated by node P_1 to the entire proactive island. It then transmits all the data packets queued for destination D.

10.3.3 Route Maintenance

If a reactive node detects a link break, it broadcasts a route error (RERR) message. This RERR message is sent to all the predecessor nodes affected by this link breakage. If a proactive node receives a RERR message for some destination node D, it deletes the route table entry for that destination. The new information is sent immediately to all the nodes in the *proactive island* through triggered route update message. The received RERR message is also transmitted to all the reactive nodes in the one-hop neighborhood of the proactive island. These reactive nodes propagate the topology change information contained in the RERR message to the other effected nodes.

If a proactive node (say node k) leaves the *proactive island*, some proactive node in the island detects the link break with this node k. The proactive node then transmits a route lost notification (RL_NOTIFY) message to all the reactive nodes in the one-hop neighborhood of the proactive island. This topology change information will propagate further, if any of these reactive nodes possess an active route to the node k.

CHAPTER 11

Testing, Experimentation and Analysis

The implementations of DSDV, A-DSDV and AODV were tested on a real test bed of laptop computers. Both multihop and co-located experiments were performed to test the implementations. A real test bed performance study was also conducted for all the three routing protocols.

11.1 Test-Bed Setup

The test bed used for experimental studies consisted of 6 Compaq Presario laptops. Each of these laptops were equipped with Cisco Aironet 350 wireless card. The first phase of testing was done in a co-located manner, where all the 6 laptops were lying on a single desk. This was followed by multiple hops experiments, where the 6 laptops were scattered all over the *Coordinated science lab (CSL)* to establish a multihop ad-hoc network.

The two main purposes of testing were to ensure the correctness of the implementations and to conduct a performance analysis of the implemented routing protocols.

11.2 Functionality Testing

The DSDV, A-DSDV, and AODV routing protocol implementations were tested for their correctness. Both co-located and multihop experiments were conducted for all the three implementations to test out various protocol features.

11.2.1 DSDV

The following tests were performed to test out the DSDV implementation.

Periodic Updates: This tests that the periodic routing update messages were generated and processed correctly.

Triggered Updates: This tests that the triggered routing update messages were generated and processed correctly.

Full Dump and Incremental Dump: This tests that the *full dump* routing update messages were generated only under the conditions specified by the DSDV routing protocol. All other routing update messages used *incremental dump* packets.

Damping Fluctuations: This tests that the settling time data is used to delay advertising changes to an unstable route.

Broken Links and Route Expiry: This test was done to ensure that when a node leaves the network, one of its predecessor nodes detects the broken link, forwards the broken link information to other nodes in the network, and the route entry, for the node which has left the network, eventually expires at every node in the network.

Actions After Reboot: This test was done to ensure that any node waits for *DELETE_PERIOD* time after reboot, before sending out its first route update message, to avoid the possibility of formation of any routing loops.

11.2.2 Adaptive DSDV

Adaptive DSDV (A-DSDV) was tested to make sure that the values of periodic update intervals were computed adaptively as the network dynamics changes. A log was created to monitor the changes in the periodic update intervals. Testing also confirmed that in a static network, the periodic update interval converges to the maximum specified value for the update interval. Testing was also performed to ensure that the $\sqrt{(2n)}$ law was followed while carrying out full dump of routing informations.

11.2.3 AODV

The following tests were conducted to test the correctness of AODV implementation:

Route request/Route reply discovery cycle: This tests that a route request (RREQ) is broadcasted by a source node, whenever a route is needed. This RREQ is processed correctly by all the intermediate nodes and eventually a route reply (RREP) is generated by a node possessing an unexpired route to the destination. The generated RREP is correctly processed and forwarded to the source node by the other intermediate nodes. This involves testing both single hop and multiple hops route discovery cycles.

Gratuitous RREP: This tests that an intermediate node replying to a RREQ also generates a gratuitous RREP, if the *G* flag is set in the RREQ packet.

Route Reply Acknowledgement: This tests that a node generates a route reply acknowledgement (RREP-ACK) and transmits it back to the node from which the RREP was received, if the received RREP has its *A* flag set.

Hello Messages: This tests that if hello messages are enabled, a node sends out these messages every *HELLO_INTERVAL* seconds. A node also creates route table entries for neighbors from which hello messages are received. This test also ensures that a node starts a broken link timer for every such neighbor.

Broken Link and Local Repair: This tests that if hello messages are not received from a previous neighbor, then the node assumes that the link is broken. It puts the destination node into the local repair list and attempts a local repair if a data packet is received within the timeout period.

Route Error: This tests that a node generates a route error (RERR) message whenever a local repair attempt fails or whenever it receives a RERR message from one of its neighbors.

Unidirectional Links: This test ensures that a node drops all the RREQ packets from a neighboring node with which it has a unidirectional link.

Actions After Reboot: This test ensures that a node waits for *DELETE_PERIOD* time after reboot, before acting as a full blown router, to avoid the possibility of formation of any routing loops.

All of these tests were performed in both co-located and multihop ad-hoc networks.

11.3 Performance Study

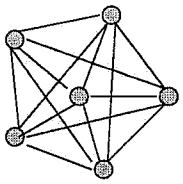
A detailed performance evaluation of DSDV, A-DSDV, and AODV was conducted in co-located as well as multihop ad-hoc networks. Performance analysis was done only for static ad-hoc networks. A traffic generator application (*traffic*) was used to generate traffic at a specified rate. The following parameters of the Cisco Aironet 350 cards were configured while conducting performance studies.

Txpower: This defines the packet transmission power level of the card. It was set to 100mW throughout the entire experimentation.

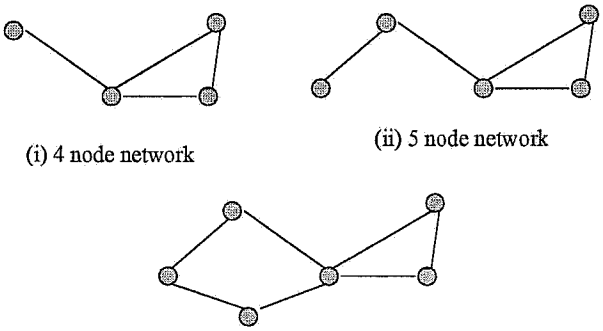
Card Bit Rate: This defines the speed at which the card transmits the bits over the wireless medium. Three different values (1Mbps, 5.5Mbps and 11Mbps) were used for this parameter. The card bit rate was set to the rate at which the traffic was generated by the traffic generator.

RTS Threshold: RTS/CTS adds a handshake before each packet transmission to make sure that the channel is clear. This parameter sets the size of the smallest packet for which a node sends RTS. Two values were used for this parameter. A value of '1' ensures that the RTS/CTS handshake will be performed for all the packets transmitted. A value equal to '2312' switches off the RTS/CTS handshake.

The traffic generator was run on each node in the network for one minute, pumping data at the specified data rate. The number of correctly received packets were recorded at each node in the network. The throughput was computed as the ratio of the sum total of all the



(a) Co-located ad-hoc network

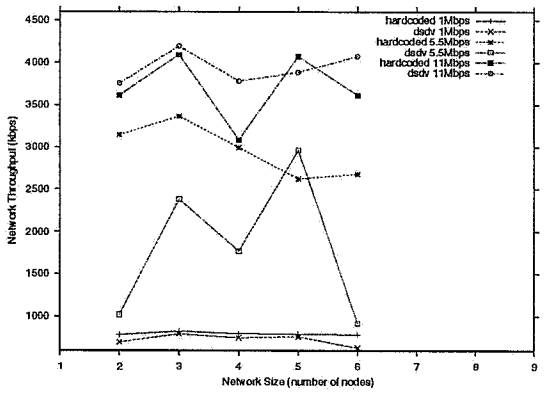


(b) Multihop ad-hoc networks

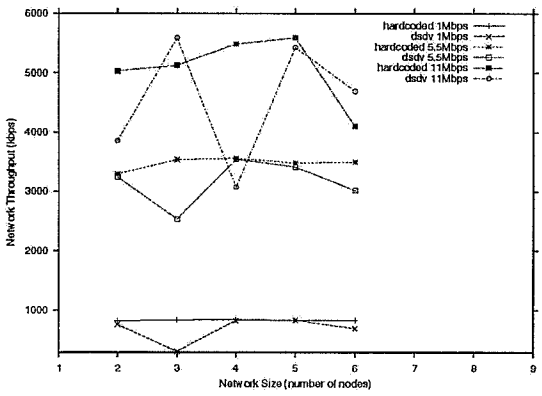
Figure 11.1 Test-bed setup for performance studies

correctly received packets to the total time for which the traffic was generated (one minute in our case). During each simulation interval, every node records the total number of bytes of all the routing control packets transmitted by the node. This quantity was defined as the *local routing overhead* of each node. The routing overhead for the entire network was computed by taking the sum total of the local routing overheads for every node in the network.

The metrics considered while conducting performance analysis were *throughput* and *routing overhead*. Figure 11.1 depicts the ad-hoc network setup used for performance testing. Figure 11.1a shows the co-located ad-hoc network, while the Figure 11.1b shows the multiple hops ad-hoc networks of 4, 5 and 6 nodes used while conducting performance experiments.



(a) Network Throughput with RTS/CTS enabled



(b) Network Throughput with RTS/CTS disabled

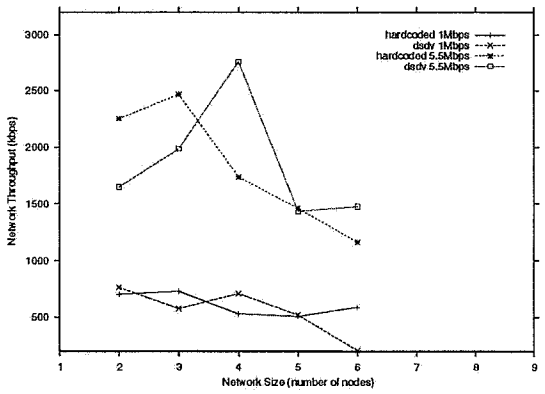
Figure 11.2 DSDV Throughput for co-located ad-hoc networks

11.3.1 Throughput

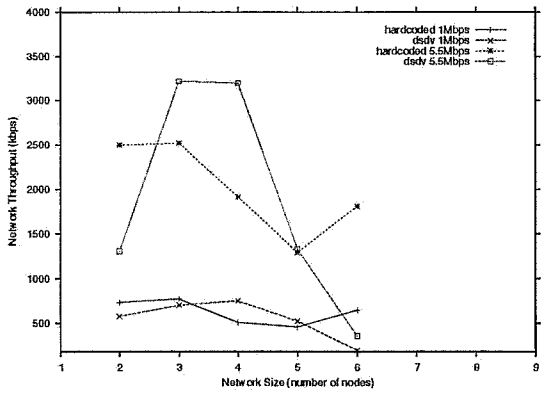
To better study the throughput variations of the ad-hoc routing protocols, we also conducted the throughput experiments with the routes hardcoded into the kernel routing table. This set of experiments were classified as *hardcoded* experimentation. The throughput results of different routing protocols were compared with the data obtained from the hardcoded experimentations.

First, we present throughput data individually for the DSDV, A-DSDV and AODV routing protocols, as compared to the throughput data for the hardcoded experiments. Then we do a comparison of throughput results of DSDV and A-DSDV routing protocols. Finally, the throughput results of all the three routing protocols are compared together.

Figure 11.2 presents the network throughput for the DSDV routing protocol as compared to the network throughput for the hardcoded experiments. The throughput results are presented for ad-hoc networks of different sizes and at different traffic rates. The results shown in Figure 11.2 correspond to co-located ad-hoc networks. Figure 11.2(a) presents the DSDV throughput with RTS/CTS handshake enabled, while the Figure 11.2(b) gives the throughput data without the RTS/CTS handshake. The throughput in general decreases as the number of nodes increases in the network. At lower traffic rates, the throughput performance of DSDV



(a) Network Throughput with RTS/CTS enabled

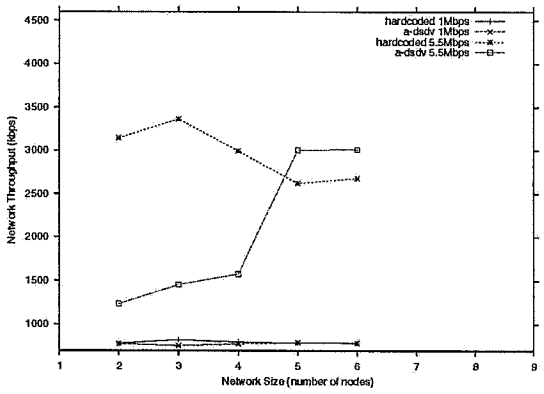


(b) Network Throughput with RTS/CTS disabled

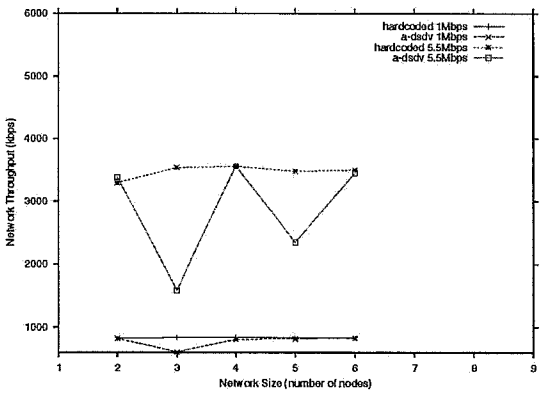
Figure 11.3 DSDV Throughput for multihop ad-hoc networks

is almost the same as for the hardcoded experimentations. At higher traffic rates, the DSDV routing protocol shows lower throughput than the hardcoded experiments. This is evident from 5.5Mbps and 11Mbps plots in Figure 11.2(a) and Figure 11.2(b). DSDV still achieves throughput results very close to the hardcoded throughput data most of the time, sometimes even surpassing the hardcoded value. The major drops in the DSDV throughput at high traffic rates could be because of the fact that some of the routing update packets might be getting delayed or lost due to high data traffic. This might lead to the expiration and deletion of routes and hence a drop in the network throughput. Figure 11.3 shows throughput data for multihop ad-hoc networks running the DSDV routing protocol. The comparison with hardcoded results are made for 1Mbps and 5.5Mbps traffic rates. The DSDV routing protocol shows better throughput performance than the hardcoded experiments most of the time.

Figure 11.4 illustrates the throughput performance of the Adaptive DSDV (A-DSDV) routing protocol for co-located ad-hoc networks, as compared to the throughput results for the hardcoded experiments. The throughput results are shown for 1Mbps and 5.5Mbps traffic rates. Figure 11.4(a) illustrates the throughput data for A-DSDV routing protocol with RTS/CTS enabled, while the throughput data without the RTS/CTS handshake is depicted in Figure 11.4(b). At 1Mbps traffic rate, A-DSDV shows throughput performance results almost same as the hard-

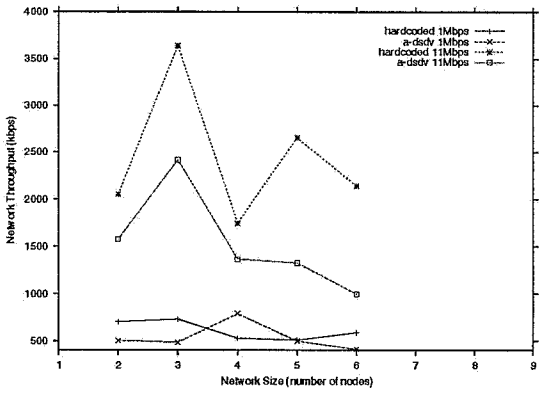


(a) Network Throughput with RTS/CTS enabled

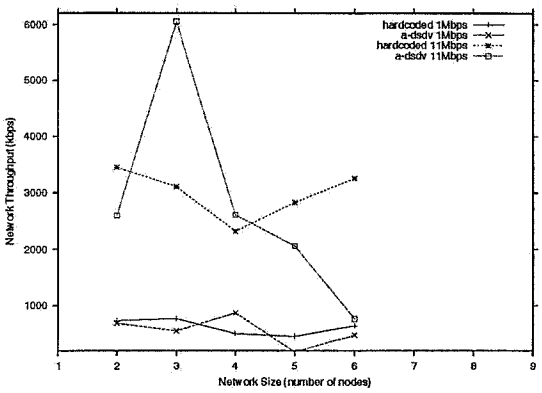


(b) Network Throughput with RTS/CTS disabled

Figure 11.4 A-DSDV Throughput for co-located ad-hoc networks



(a) Network Throughput with RTS/CTS enabled



(b) Network Throughput with RTS/CTS disabled

Figure 11.5 A-DSDV Throughput for multihop ad-hoc networks

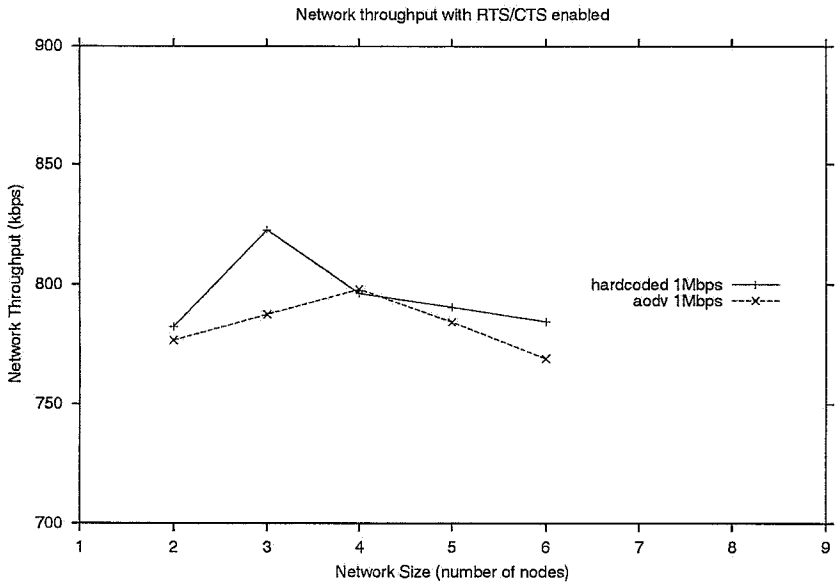
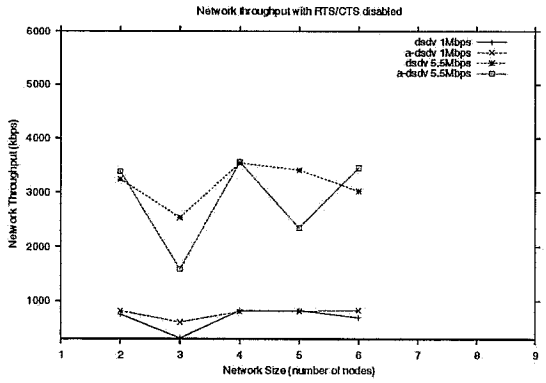


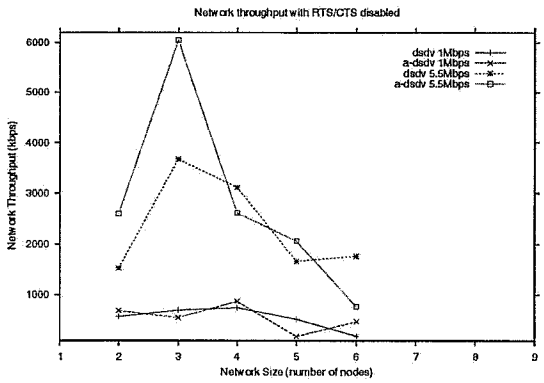
Figure 11.6 Throughput for AODV routing protocol

coded results. At higher traffic rates, A-DSDV shows lower throughput than the throughput results for the hardcoded experiments for some of the cases (2,3 and 4 node networks in Figure 11.4(a)). However, for most of the cases, it achieves throughput results very close to the throughput results for the hardcoded experiments, even exceeding the hardcoded values in certain scenarios (5 and 6 node networks in Figure 11.4(a)). The reason for drops in A-DSDV throughput at higher traffic rates could be the same as the one for DSDV routing protocol, viz. loss of or delay in routing update messages at higher traffic rates, resulting in the deletion of certain routes. Figure 11.5 represents the throughput results of A-DSDV routing protocol for multihop ad-hoc networks. As expected, for lower traffic rates, the throughput results for A-DSDV routing protocol is almost same as that for the hardcoded experiments. At higher traffic rates, A-DSDV exhibits throughput performance very close to the hardcoded results most of the time. For certain cases, at higher traffic rates, A-DSDV shows better throughput data than the hardcoded results (3 and 4 node networks in Figure 11.5(b).

Figure 11.6 shows throughput results of AODV routing protocol as compared to the throughput data for the hardcoded experiments. The results are shown for 1Mbps traffic rate. The



(a) Network Throughput for co-located networks



(b) Network Throughput for multihop networks

Figure 11.7 Throughput Comparison between DSDV and A-DSDV

AODV routing protocol shows throughput results almost similar to the throughput results of the hardcoded experiments.

Figure 11.7 presents the comparison of throughput for the DSDV and A-DSDV routing protocols at different traffic rates. Figure 11.7(a) shows the throughput comparison for co-located ad-hoc networks, while Figure 11.7(b) shows the throughput comparison for multihop ad-hoc networks. For both co-located and multihop ad-hoc networks, at lower traffic rates, the measured throughput for A-DSDV is better than the DSDV for most of the time. At higher traffic rates also, A-DSDV achieves better throughput than DSDV in most of the cases. Even for the cases where the A-DSDV throughput is smaller than the DSDV throughput, the difference is not much. Thus A-DSDV exhibits throughput performance better than or equal to DSDV most of the time.

Figure 11.8 shows the throughput data for DSDV, A-DSDV, AODV and the hardcoded experiments. AODV routing protocol performs very close to the hardcoded throughput results. A-DSDV comes very close to the AODV performance results. Also, on an average, A-DSDV performs better than the DSDV routing protocol. The performance of the DSDV protocol is worst of all the four plots. This could be because of the fact that the frequent periodic route update messages in DSDV consume a considerable amount of network bandwidth. This is not

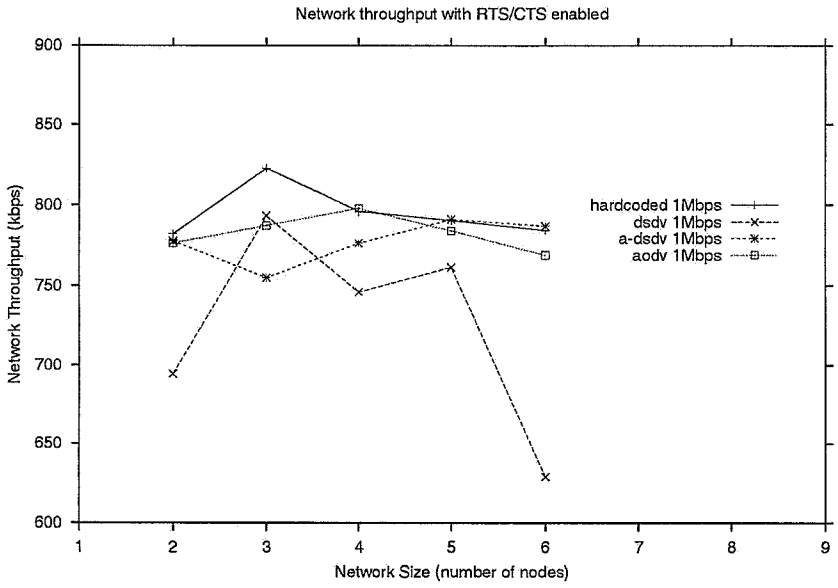
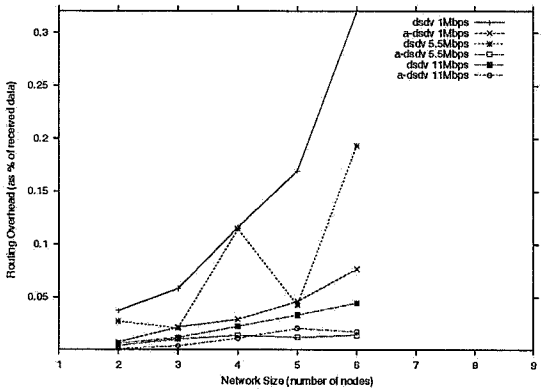


Figure 11.8 Throughput for different routing protocols

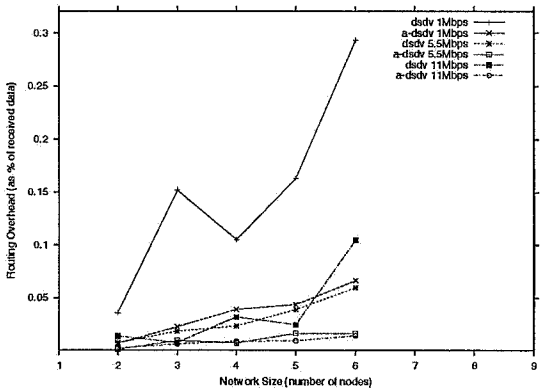
the case with A-DSDV as it reduces the frequency of route update messages for static ad-hoc networks.

11.3.2 Routing Overhead

Figure 11.9 shows the comparison of routing overheads between DSDV and A-DSDV routing protocols for co-located ad-hoc networks, at different traffic rates. Figure 11.9(a) shows the routing overhead data with RTS/CTS enabled, while Figure 11.9(b) illustrates the routing overhead data with RTS/CTS disabled. At all the traffic rates, the routing overhead of A-DSDV is substantially smaller than the routing overhead of DSDV routing protocol. This is because for a static network, the frequency of periodic route update messages is much smaller for A-DSDV routing protocol as compared to DSDV routing protocol. This effectively reduces the routing overhead for the A-DSDV protocol. Also, routing overhead for A-DSDV is somewhat higher with RTS/CTS handshake enabled than without the RTS/CTS handshake. Figure 11.10 illustrates the routing overhead data for DSDV and A-DSDV for multihop networks. As expected, A-DSDV has much smaller routing overhead as compared to the DSDV protocol. Figure 11.11

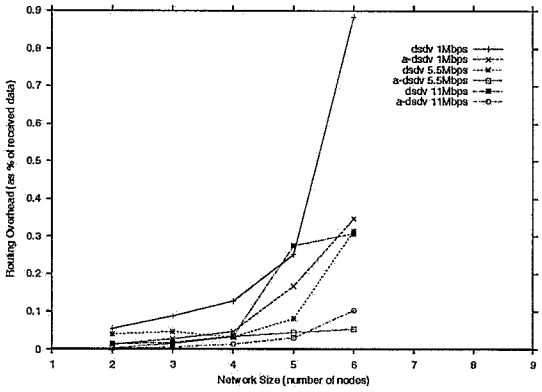


(a) Routing Overhead with RTS/CTS enabled

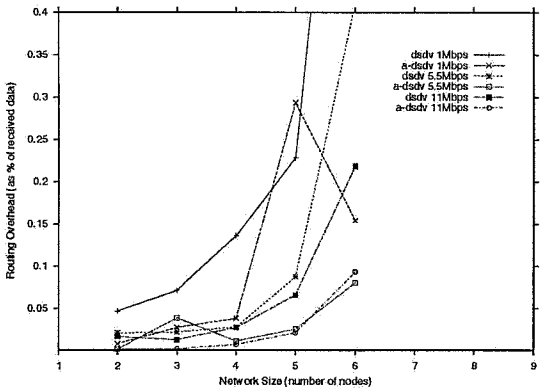


(b) Routing Overhead with RTS/CTS disabled

Figure 11.9 Routing Overhead for co-located ad-hoc networks



(a) Routing Overhead with RTS/CTS enabled



(b) Routing Overhead with RTS/CTS disabled

Figure 11.10 Routing Overhead for multihop ad-hoc networks

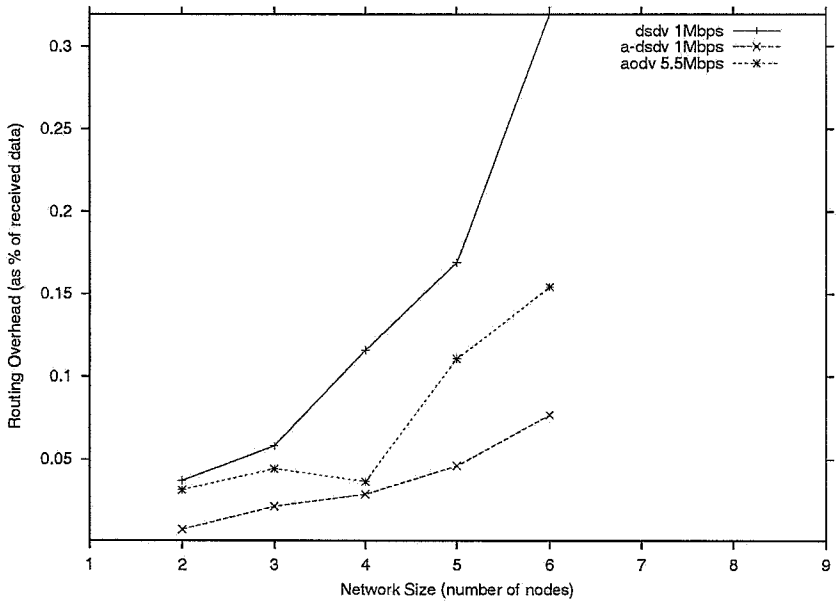
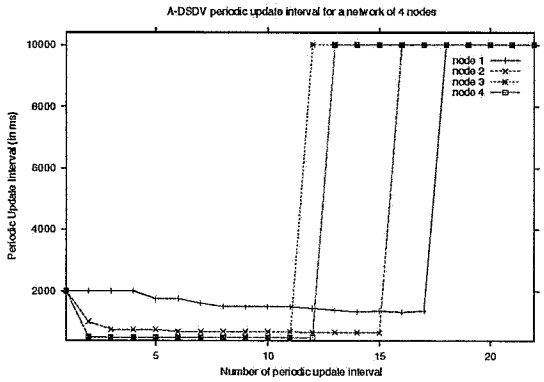


Figure 11.11 Routing overhead for different routing protocols

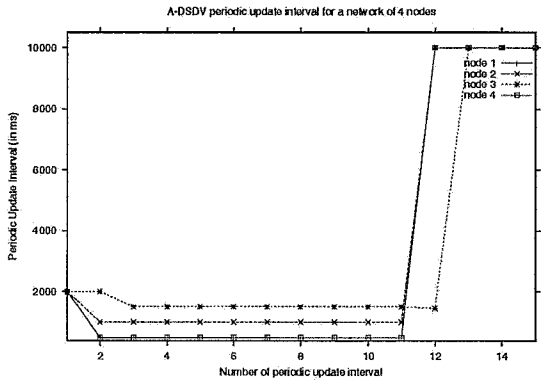
shows routing overhead for AODV, DSDV and A-DSDV for co-located ad-hoc networks with 1Mbps traffic rate. DSDV has the highest routing overhead, followed by AODV. A-DSDV displays the lowest routing overhead characteristics.

11.3.3 Adaptive Periodic Update Interval for A-DSDV

Figure 11.12 shows the variations of periodic update interval at every node for a 4 node ad-hoc network running A-DSDV routing protocol. The periodic update interval data for both co-located and multihop networks are shown. The value of the periodic update interval for all the 4 nodes finally converges to *MAX_PERIODIC_UPDATE_INETERVAL* value specified by A-DSDV protocol (10 seconds in our case). Figure 11.13 shows similar results for a co-located ad-hoc network of 5 nodes. In this case also, the value of the periodic update interval converges to *MAX_PERIODIC_UPDATE_INTERVAL* for all the 5 nodes.



(a) Periodic update interval for a co-located network



(b) Periodic update interval for a multihop network

Figure 11.12 Variations in A-DSDV periodic update interval for 4 node ad-hoc networks

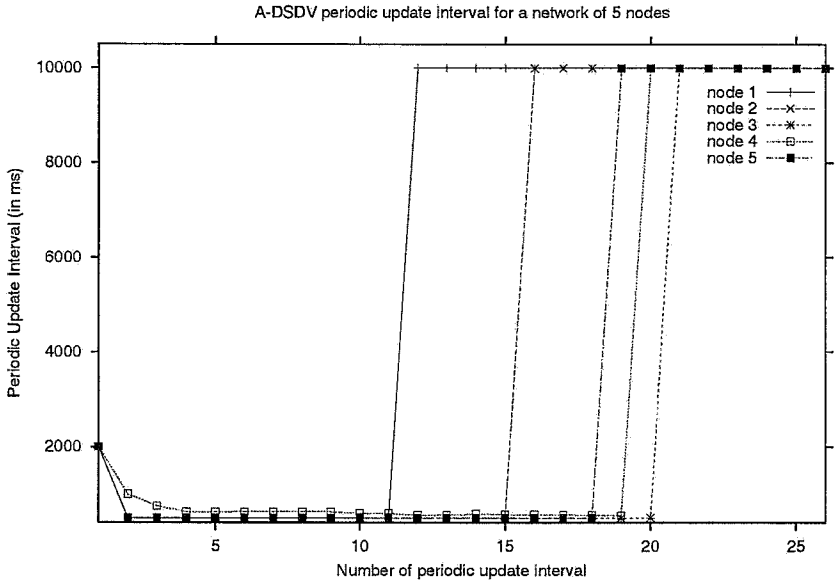


Figure 11.13 A-DSDV Periodic update interval for a 5 node co-located ad-hoc network

CHAPTER 12

Conclusion

We have provided implementations of DSDV ad-hoc routing protocol on the Linux kernel. We also provide a detailed specification of DSDV routing protocol. An improved version of DSDV routing protocol called *Adaptive DSDV* (A-DSDV) has been designed and implemented. A-DSDV provides a fully automatic version of the DSDV routing protocol in terms of configuring various DSDV parameters. This completely obviates the need of hardcoding values for any of the DSDV parameters.

We have done an in-depth exploration of issues involved with on-demand routing in ad-hoc networks. An Ad-hoc Support Library (ASL) has been developed to provide system services for on-demand routing in mobile ad-hoc networks. The AODV routing protocol has been implemented using ASL. The resulting AODV implementation (AODV-UIUC) is much simpler and cleaner than many of the existing AODV implementations. AODV-UIUC is also more efficient than other existing implementations of AODV in terms of per packet processing overhead, and it does not require any modifications to the Linux kernel code.

We have also conducted an in-depth performance study of DSDV, A-DSDV and AODV routing protocols on a real test bed consisting of 6 laptop computers equipped with Cisco Aironet 350 wireless cards. The performance experimentations have been conducted both for co-located and multihop ad-hoc networks. The analysis of experimental results shows that the A-DSDV routing protocol performs better than or equal to the DSDV routing protocol, in terms of throughput, most of the time. The AODV routing protocol exhibits maximum throughput

performance. Also, the throughput results for the A-DSDV routing protocol matches closely with the throughput data for AODV. In terms of routing overhead, A-DSDV shows better performance than the DSDV routing protocol. Also, for co-located ad-hoc networks, the routing overhead incurred by A-DSDV is smaller than both the DSDV and AODV routing protocols.

We have also presented a novel approach to enable inter-operability between reactive and proactive routing protocols in an hybrid ad-hoc network. An Inter-operability Support Services (ISS) module has been designed to provide services to enable inter-operability between proactive and reactive ad-hoc routing protocols. The ISS module can be utilized to enhance the existing ad-hoc routing protocols to support inter-operability with minimal modifications to the routing daemon code.

REFERENCES

[1] Phil Roberts and Basavraj Patil (chairs), "IETF IP Routing for Wireless/Mobile Hosts (Mobile IP) working group," http://www.ietf.org/html.charters/mobileip-charter.html.

[2] "IEEE Workshop on Disaster Recovery Networks," Jun 2002, http://comet.ctr.columbia.edu/diren.

[3] Jochen H. Schiller, *Mobile Communications*, chapter 4, pp. 84–91, Addison-Wesley, Pearson Education Limited, 2000, ISBN: 9814053392.

[4] Charles E. Perkins, Bobby Woolf, and Sherman R. Alpert, *Mobile IP Design Principles and Practices*, Prentice Hall PTR, first edition, January 1998, ISBN: 0201634694.

[5] "IEEE 802.3 CSMA/CD (Ethernet) working group," http://grouper.ieee.org/groups/802/3/.

[6] Chane L. Fullmer and J.J. Garcia-Luna-Aceves, "Solutions to Hidden Terminal Problems in Wireless Networks," in *Proceedings of ACM SIGCOMM*, Cannes, France, 1997.

[7] "IEEE 802.11 Working Group for Wireless LANs," http://grouper.ieee.org/groups/802/11/main.html.

[8] Bob O'Hara and Al Petrick, *IEEE 802.11 Handbook: A Designer's Companion*, Standards Information Networks IEEE Press, 1999, ISBN: 0738118559.

[9] Robert Rozovsky and P. R. Kumar, "SEEDEX: A MAC protocol for ad hoc networks," in *Proceedings of The ACM Symposium on Mobile Ad Hoc Networking and Computing, MOBIHOC*, 2001.

[10] Swetha Narayanaswamy, Vikas Kawadia, R. S. Sreenivas, and P. R. Kumar, "Power control in ad-hoc networks: Theory, architecture, algorithm and implementation of the COMPOW protocol," in *European Wireless Conference*, 2002.

[11] Joseph Macker and Scott Corson (chairs), "Mobile Ad hoc Networks (MANET) Charter," http://www.ietf.org/html.charters/manet-charter.html.

[12] Larry L. Peterson and Bruce S. Davie, *Computer Networks A Systems Approach*, Morgan Kaufmann, San Francisco, CA, 2000, ISBN: 1558605770.

[13] Dimitri Bertsekas and Robert Gallager, *Data Networks*, Prentice Hall, Englewood Cliffs, New Jersey, 1987, ISBN: 0131968254.

[14] Douglas E. Comer, *Internetworking with TCP/IP Principles, Protocols and Architecture*, vol. 1, Prentice Hall of India, New Delhi, 2000, ISBN: 8120310535.

[15] Eugene Blanchard, *Introduction to Networking and Data Communications*, chapter 42, 2001, Online Book.

[16] Christian Huitema, *Routing in the Internet*, vol. 1, Prentice Hall, Englewood Cliffs, New Jersey, 1995, ISBN: 0131321927.

[17] Bassam Halabi, *Internet Routing Architecture*, Cisco Press, New Riders Publishing, Indianapolis, IN, 1997.

[18] "The Internet Engineering Task Force (IETF) homepage," http://www.ietf.org.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw Hill, 2000, ISBN: 0262032937 .

[20] Charles E. Perkins, *Ad Hoc Networking*, Addison-Wesley, Pearson Education Limited, New Jersey, 2001, ISBN: 0201309769.

[21] E. Royer and C-K. Toh, "A review of current routing protocols for ah hoc mobile wireless networks," *IEEE Personal Communication*, vol. 6, no. 2, pp. 46–55, April 1999.

[22] Charles E. Perkins and Pravin R. Bhagwat, "Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers," in *Proceedings of ACM SIGCOMM*, London, U.K., September 1994, pp. 234–244.

[23] C. Chiang, H. Wu, W. Liu, and M. Gerla, "Routing in Clustered Multihop, Mobile Wireless Networks," in *The IEEE Singapore International Conference on Networks (SICON)*, April 1997, pp. 197–211.

[24] Shree Murthy and J. J. Garcia-Luna-Aceves, "An Efficient Routing Protocol for Wireless Networks," *ACM Mobile Networks and Applications Journal, Special issue on Routing in Mobile Communication Networks*, vol. 1, no. 2, pp. 183–197, October 1996.

[25] A. Iwata, C. Chiang, G. Pei, M. Gerla, and T. Chen, "Scalable routing strategies for ad-hoc wireless networks," *IEEE Journal on Selected Areas in Communications*, August 1999.

[26] X. Chen, L. Qi, and D. Sun, "Global and superlinear convergence of the smoothing Newton method and its application to general box constrained variational inequalities," *Mathematics of Computation*, vol. 67, no. 222, 1998.

[27] Charles E. Perkins, Elizabeth M. Royer, and Samir Das, "Ad hoc on demand distance vector routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, LA, February 1999, pp. 90–100.

[28] Vincent D. Park and M. Scott Corson, "A highly adaptive distributed routing algorithm for mobile wireless networks," in *Proceedings of IEEE INFOCOM*, Kobe, Japan, April 1997.

[29] David B. Johnson and David A. Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing*, Tomasz Imielinski and Hank Korth, Eds., vol. 353. Kluwer Academic Publishers, 1996.

[30] R. Dube, C. Rais, K. Wang, and S. Tripathi, "Signal stability based adaptive routing (ssa) for ad hoc mobile networks," in *IEEE Personal Communication*, February 1997.

[31] Zygmunt Haas, "A new routing protocol for the reconfigurable wireless networks," in *Proceedings of the IEEE Int. Conf. on Universal Personal Communications*, October 1997.

[32] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, and A. Qayyum nd L. Viennot, "Optimized link state routing protocol for ad hoc networks," in *IEEE INMIC, Pakistan*, 2001.

[33] M Joa-Ng and I.-T. Lu, "A peer-to-peer zone based two-level link state routing for mobile ad hoc networks," in *IEEE Journal on Selected Areas in Communication, Special issue on Ad-Hoc Networks*, August 1999.

[34] D. Maltz, J. Broch, and D. Johnson, "Experiences designing and building a multi-hop wireless ad hoc network testbed," Technical Report CMU-CS-99-116, School of Computer Science, Carnegie Mellon University (http://www.monarch.cs.cmu.edu/papers.html), March 1999.

[35] Chai-Keong Toh, "A novel distributed routing protocol to support ad hoc mobile computing," in *Proceedings of IEEE 15th Annual International Conference on Computers and Communications*, Phoenix, March 1996, pp. 480–486.

[36] C.-K. Toh, "Long-lived ad-hoc routing based on the concept of associativity," March 1999, Internet Draft of IETF MANET Working Group.

[37] Z. J. Haas and M. R. Pearlman, "The Zone Routing Protocol (ZRP) for ad hoc networks," August 1998, IETF Internet Draft, Mobile Ad hoc Network (MANET) Working Group.

[38] J. M. Jaffe and F. H. Moss, "A responsive distributed algorithm for computer networks," in *IEEE Transcations on Communications*, July 1982.

[39] S. Das, R. Castaneda, and J. Yan, "Simulation based performance evaluation of mobile, ad hoc network routing protocols," *ACM/Baltzer Mobile Networks and Applications (MONET) Journal*, pp. 179–189, July 2000.

[40] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," in *Mobile Computing and Networking*, 1998, pp. 85–97.

[41] Samir R. Das, Charles E. Perkins, and Elizabeth M. Royer, "Performance comparison of two on-demand routing protocols for ad hoc networks," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, March 2000, pp. 3–12.

[42] Vikas Kawadia, Yongguang Zhang, and Binita Gupta, "System services for implementing ad hoc routing protocols," in *International Workshop on Ad Hoc Networking*, 2002.

[43] "Netfilter/Iptables homepage," http://www.netfilter.org.

[44] C. E. Perkins, E. M. Royer, and Samir R. Das, "Ad hoc on-demand distance vector routing," January 2002, IETF Internet Draft, draft-ietf-manet-aodv-10.txt, work in progress.

[45] "AODV Implementation at University of California, Santa Barbara," http://moment.cs.ucsb.edu/AODV/aodv.html.

[46] "AODV Implementation at Uppasala University," http://www.docs.uu.se/ henrikl/aodv.

[47] "Kernel AODV at National Institute of Standards and Technology (NIST)," http://w3.antd.nist.gov/wctg/aodv_kernel.

[48] Elizabeth M. Royer and Charles E. Perkins, "An implemenatation study of the aodv routing protocol," in *Proceedings of the IEEE Wireless Communications and Networking Conference*, Chicago, IL, September 2000.