On-Demand Routing in Multi-hop Wireless Mobile Ad Hoc Networks

David A. Maltz
May 2001
CMU-CS-01-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:

David B. Johnson, Chair
M. Satyanarayanan
Hui Zhang
Martha Steenstrup, Stow Research L.L.C.

This research was sponsored in part by the National Science Foundation (NSF) under CAREER Award NCR-9502725, by Caterpillar Corporation, by the Air Force Materiel Command (AFMC) under DARPA contract number F19628-96-C-0061, by an Intel Graduate Fellowship, and by an IBM Cooperative Fellowship.

On-Demand Routing in Multi-hop Wireless Mobile Ad Hoc Networks

by
David A. Maltz

Submitted to the Department of Computer Science
on May 2001, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

An ad hoc network is a collection of wireless mobile nodes dynamically forming a temporary network without the use of any preexisting network infrastructure or centralized administration. Routing protocols used in ad hoc networks must automatically adjust to environments that can vary between the extremes of high mobility with low bandwidth, and low mobility with high bandwidth. This thesis argues that such protocols must operate in an *on-demand* fashion and that they must carefully limit the number of nodes required to react to a given topology change in the network. I have embodied these two principles in a routing protocol called *Dynamic Source Routing* (DSR). As a result of its unique design, the protocol adapts quickly to routing changes when node movement is frequent, yet requires little or no overhead during periods in which nodes move less frequently. By presenting a detailed analysis of DSR's behavior in a variety of situations, this thesis generalizes the lessons learned from DSR so that they can be applied to the many other new routing protocols that have adopted the basic DSR framework. The thesis proves the practicality of the DSR protocol through performance results collected from a full-scale 8 node testbed, and it demonstrates several methodologies for experimenting with protocols and applications in an ad hoc network environment, including the emulation of ad hoc networks.

Thesis Supervisor: David B. Johnson
Title: Associate Professor, Computer Science Department

Thesis Committee Member: M. Satyanarayanan
Title: Carnegie Group Professor, Computer Science Department

Thesis Committee Member: Martha Steenstrup
Title: Consultant, Stow Research L.L.C.

Thesis Committee Member: Hui Zhang
Title: Associate Professor, Computer Science Department

Acknowledgments

The preparation of this thesis and my preparation as a researcher have benefited from the guidance of many people, but none more than Dave Johnson, my advisor. From framing a research question to carrying out an experiment, from kernel hacking to paper writing, and from the finer points of grammar to the finer points of teaching classes, Dave has spent uncountable hours helping me develop as a scientist, and I am truly grateful for his efforts.

I am also grateful for the advice and direction my thesis committee has provided to me during my graduate studies. Satya and Hui helped me focus my thesis work and directed me towards areas of fruitful research. Martha provided extensive comments on the thesis itself and used her tremendous knowledge of the field to help me place the work in its proper context.

My thesis has achieved the breadth that it has, stretching from protocol design through simulation, emulation, and implementation, because I was able to draw on the talents of the entire Monarch Project in building out the systems whose evaluations are reported in this thesis. The protocol comparisons reported in Chapter 5 draw heavily on the work of Josh Broch, Jorjeta Jetcheva, and Yih-Chun Hu. My personal contributions include the choice of protocols to compare and the metrics and analysis used to compare them, the implementations of DSR and IMEP, and hacking on large parts of the other protocols and the simulator. The testbed reported in Chapter 7 is the work of an even larger team, including Josh, Jorjeta, Yih-Chun, Qifa Ke, Ben Bennington, Ratish Punnoose, Pavel Nikitin, Dan Stancil, Satish Shetty, Michael Lohmiller, Sam Weiler, and Jon Schlegel. For the testbed, I participated in a number of roles: I assisted in the implementation of the DSR protocol; I oversaw the testing and evaluation of the protocol (including the design and analysis of the experiments run and the development of the traffic generators and emulation system); and I directed the integration of DSR with Mobile IPv4.

I am indebted to Bob Baron and Jan Harkes for providing and maintaining the trace modulation and Coda code used in Chapter 6 of this thesis. I must also thank Brian Noble for long discussions of emulation philosophy and for providing a model of what a good systems graduate student should be.

On a personal level, I must sincerely thank several people for enabling me to complete my thesis. First and foremost is Josh Broch, who for the past 5 years has been my principal colleague and collaborator. He has helped me maintain my sanity and perspective. He has kept me company in lab to all hours of the night, rewritten my code, traveled around the world with me, and defeated my attempts to make serious use of the cvs log files. Josh is the author of large portions of the extensions to the *ns-2* simulator described in Chapter 3, including the IEEE 802.11 module and much of the glue code that holds the simulator together. Josh also wrote most of the DSR implementation used in the testbed described in Chapter 7.

I am also deeply indebted to Chris Cole. As my supervisor at FreeSpace Communications, Chris arranged for me to have time off work to write my thesis at a critical juncture in its preparation. Without his providing the opportunity, the carrot, and the stick in just the right proportion, I probably would not have finished.

Finally, I must thank my friends and family. This work is dedicated to you.

Contents

Chapter 1

Introduction

The need to exchange digital information outside the typical wired office environment is growing. For example, a class of students may need to interact during a lecture; business associates serendipitously meeting in an airport may wish to share files; or disaster recovery personnel may need to coordinate relief information after a hurricane or flood. Each of the devices used by these information producers and consumers can be considered a *node* in an *ad hoc network*.

In a typical ad hoc network, mobile nodes come together for a period of time to exchange information. While exchanging information, the nodes may continue to move, and so the network must be prepared to adapt continually. In the applications we are interested in, networking infrastructure such as repeaters or base stations will frequently be either undesirable or not directly reachable, so the nodes must be prepared to organize themselves into a network and establish routes among themselves without any outside support. The idea of ad hoc networking is sometimes also called *infrastructureless networking* [76], since the mobile nodes in the network dynamically establish routing among themselves to form their own network "on the fly."

1.1. Overview

In the simplest scenarios, nodes may be able to communicate directly with each other, for example, when they are within wireless transmission range of each other. However, ad hoc networks must also support communication between nodes that are only indirectly connected by a series of wireless hops through other nodes. For example, in Figure 1.1, nodes **A** and **C** must enlist the aid of node **B** to relay packets between them in order to communicate. In general, an ad hoc network is a network in which every node is potentially a router, and every node is potentially mobile.

The presence of wireless communication and mobility make an ad hoc network unlike a traditional wired network and requires that the routing protocols used in an ad hoc network be based on new and different principles. Routing protocols for traditional wired networks are designed to support tremendous numbers of nodes, but they assume that the relative position of the nodes will generally remain unchanged. In a mobile ad hoc network, however, there may be fewer nodes among which to route, but the network topology changes can be drastic and frequent as the individual mobile nodes move. My thesis is that:

> Efficient routing in an ad hoc network requires that the routing protocol operate in an on-demand fashion, and requires that the routing protocol limit the number of nodes that must be informed of topology changes. Ad hoc networks running such a protocol can be designed and implemented, and they perform well enough to support useful applications.

In this dissertation, I argue that there are two keys to designing a routing protocol that operates successfully given the challenges of an ad hoc network. First, the protocol must be fundamentally *on-demand*, meaning that it reacts to changes in the environment *only* when necessary. Second, the protocol must limit the number of nodes that are required to share consistent state information, since it is extremely expensive or impossible to maintain a distributed data structure in a consistent state across all the nodes in a rapidly changing ad hoc network.

In the chapters that follow, I will describe the Dynamic Source Routing protocol (DSR) [55, 54, 52, 51] I developed to embody these principles, and will explain why the mechanisms included in the protocol were selected from the large set of candidates I investigated. I show that DSR delivers excellent routing performance across a wide range of ad hoc network environments, and I dissect DSR into its component mechanisms to show how they combine to give DSR that performance. This dissection also exposes insights that enable the lessons learned from DSR to be generalized to other protocols. Finally, I describe two ways in which my research group and I have demonstrated the real-world practicality of ad hoc networks running the DSR protocol: first by creating an 8-node *full-scale testbed implementation of DSR*, and second by developing an *emulation system* that allows arbitrary applications to be subjected to the network conditions inside any ad hoc network scenario.

The remainder of this thesis is organized as follows. The rest of this chapter describes the challenges that a routing protocol for ad hoc networks must overcome and gives a brief history of investigations into the problem. Chapter 2 presents a brief description of DSR, which is my solution to these challenges. Chapter 3 briefly describes the simulation environment I used to conduct many of the studies described in this thesis, and it explains several of the metrics against which I evaluated the protocols being studied. In Chapter 4, I present four fundamental challenges to on-demand routing, and show how DSR meets these challenges, by dissecting DSR into the on-demand mechanisms that comprise it. Chapter 5 compares the performance of DSR with three other protocols, and establishes how on-demand mechanisms contribute to DSR's excellent performance. Chapter 7 describes the implementation of DSR in a full-scale ad hoc network testbed and the lessons learned from the experience. Finally, Chapters 8 and 9 conclude the thesis with a presentation of related work and a summary of the results of the thesis.

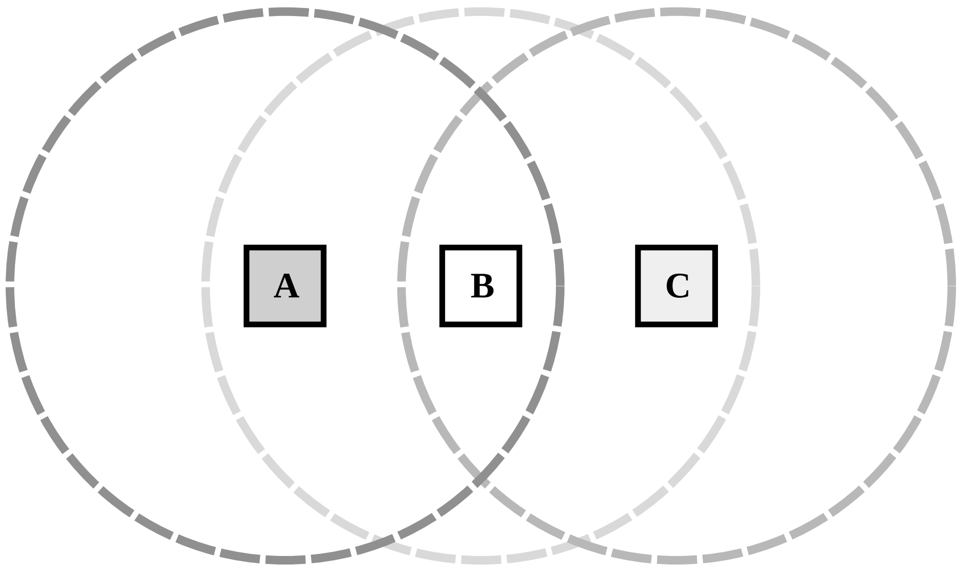


Figure 1.1 An ad hoc network of three nodes, where nodes **A** and **C** must discover the route through **B** in order to communicate. The circles indicate the nominal range of each node's radio transceiver. Nodes **A** and **C** are not in direct transmission range of each other, since **A**'s circle does not cover **C**.

1.2. Problem Statement: Routing in Ad Hoc Networks

The basic routing problem is that of finding an ordered series of intermediate nodes that can transport a packet across a network from its source to its destination by forwarding the packet along this series of intermediate nodes. In traditional hop-by-hop solutions to the routing problem, each node in the network maintains a routing table: for each known destination, the routing table lists the next node to which a packet for that destination should be sent.

The routing table at each node can be thought of as a view into part of a distributed data structure that, when taken together, describes the topology of the network. The goal of the routing protocol is to ensure that the overall data structure contains a consistent and correct view of the actual network topology. If the routing tables at some nodes were to become inconsistent, then packets can loop in the network. If the routing tables were to contain incorrect information, then packets can be dropped. The problem of maintaining a consistent and correct view becomes harder as there is an increase in the number of nodes whose information must be consistent, and as the rate of change in the actual topology increases.

The challenge in creating a routing protocol for ad hoc networks is to design a single protocol that can adapt to the wide variety of conditions that can be present in any ad hoc network over time. For example, the bandwidth available between two nodes in the network may vary from more than 10 Mbps to 10 Kbps or less. The highest speeds are achieved when using high-speed network interfaces with little interference, and the extremely low speeds may arise when using low-speed network interfaces or when there is significant interference from outside sources or other nodes' transmitters. Similar to the potential variability in bandwidth, nodes in an ad hoc network may alternate between periods during which they are stationary with respect to each other and periods during which they change topology rapidly. Conditions across a single network may also vary, so while some nodes are slow moving, others change location rapidly.

The routing protocol must perform efficiently in environments in which nodes are stationary and bandwidth is not a limiting factor. Yet, the same protocol must still function efficiently when the bandwidth available between nodes is low and the level of mobility and topology change is high. Because it is often impossible to know *a priori* what environment the protocol will find itself in, and because the environment can change unpredictably, the routing protocol must be able to adapt automatically.

Most routing protocols include at least some *periodic* behaviors, meaning that there are protocol operations that are performed regularly at some interval regardless of outside events. These periodic behaviors typically limit the ability of the protocols to adapt to changing environments. If the periodic interval is set too short, the protocol will be inefficient as it performs its activities more often than required to react to changes in the network topology. If the periodic interval is set too long, the protocol will not react sufficiently quickly to changes in the network topology, and packets will be lost.

Periodic protocols can be designed to adjust their periodic interval to try to match the rate of change in the network [11], but this approach will suffer from the overhead associated with the tuning mechanism and the lag between a change in conditions and the selection of a new periodic interval. In the worst case, which consists of bursts of topology change followed by stable periods, adapting the periodic interval could result in the protocol using a long interval during the burst periods and a short interval in the stable periods. This worst case may be fairly common, for example, as when a group of people enter a room for a meeting, are seated for the course of the meeting, and then stand up to leave at the end.

The alternative to a periodic routing protocol is one that operates in an *on-demand* fashion. On-demand protocols are based on the premise that if a problem or inconsistent state can be detected before it causes permanent harm, then all work to correct a problem or maintain consistent state can be delayed until it is known to be needed. They operate using the same "lazy" philosophy as optimistic algorithms [23, 24].

The Dynamic Source Routing protocol (DSR) completely avoids periodic behavior, and uses source routing to solve the routing information consistency problem.

First, DSR is completely on-demand, which allows the overhead of the protocol to automatically scale directly with the need for reaction to topology change. This scalability dramatically lowers the overhead of the protocol by eliminating the need for any periodic activities, such as the route advertisement and neighbor detection packets that are present in other protocols.

Second, DSR uses source routes to control the forwarding of packets through the network. The key advantage of a source routing design is that intermediate nodes do not need to maintain consistent global routing information, since the packets themselves already contain all the routing decisions. Beyond this, every packet that carries a source route carries a description of a path through the network. Therefore, with a cost of no additional packets, every node overhearing a source route learns a way to reach all nodes listed on the route.

While the on-demand mechanisms built into DSR are intended to improve the network's performance, they also have potentially significant liabilities. For example, by deferring work until it must be performed in order to complete some desired action, the time taken to complete the desired action will increase as none of the work has been precomputed. In the network, this means that packets may be delayed or lost while the network performs the tasks needed to deliver them. Chapter 4 looks at a number of these issues in detail and shows how the DSR mechanisms are able to mitigate the liabilities.

This thesis concentrates on achieving high-performance unicast routing in multi-hop wireless ad hoc networks. There are two related problems that this thesis will not directly address: quality of service (QoS) routing and multicast routing. Time-sensitive and multi-media data traffic, such as voice and video, will typically not function properly unless the senders and receivers of the data are provided with promises that the quality of service supplied by the network will fall within some prearranged bounds, or, at the very least, are provided with information about the quality of service that traffic across the network can expect. Providing this quality of service information requires the involvement of the routing protocol, since no other layer of the network stack has a view of the multiple paths available across the network. While multicast routing can be achieved at layers above the routing layer, it too benefits from access to the information available at the routing layer, and it deserves support from the routing protocol. More extensive comments on these subjects can be found in Chapter 8 on related work.

1.3. A Brief History of Packet Radio Networks

The first packet radio networks date from the very earliest days of the wireless radio, when radio station operators would manually forward telegram-like messages between stations in order to further the distance over which messages could be carried. These networks, like the Marconi Short Beam network [73], used "manually configured" routes and the intuition of the human operators to route messages through the network. This solution worked well, as the topology was very static and the links very reliable.

The first development of packet radio for computer communications was in 1970 as part of the ALOHANET project operated by the University of Hawaii [59, 36]. ALOHANET consisted of a radio network used to connect together university computers on each of the major Hawaiian islands. The network was single-hop however, meaning that only nodes directly in reach of each other could communicate. From the ALOHANET project, packet radio grew in two main directions, non-military amateur radio networks and tactical military networks.

1.3.1. Amateur Packet Radio Networks

Towards the end of the 1970s, Amateur radio operators became interested in packet radio as a means to achieve three goals: to exchange messages with other operators outside their radio horizon, to achieve better message delivery in the presence of radio interference that made voice communication impossible, and to offer new services to other amateur operators (e.g., BBS bulletin-board service) [57, 61].

Nodes in the amateur radio network are known as Terminal Node Controllers (TNCs), and the first TNCs were built in 1978 in Montreal, Canada [56]. The Vancouver Amateur Digital Communication Group followed this with a TNC kit available in 1980. The most widely used TNC was developed by the Tucson Amateur Packet Radio Corp in 1982, which claims over 100,000 TNCs have been sold [57, 19].

The amateur packet radio network today still uses a variety of different protocols. Examples include [56]:

- The NET/ROM protocol that originally allowed a user to remotely log into a directly reachable TNC and, from there, access the TNC's neighbor list and recursively log into TNC's further and further away. This ability is roughly equivalent to remote-shell (rsh) or telnet today, but limited to the abilities to only log into a directly connected machine and to only execute commands supported by the TNC. Later versions of NET/ROM [32] supported automatic routing using a variation of a distance vector protocol.

- A variant of the X.25 network protocol called AX.25. AX.25 is a true packet-based protocol that has been modified to use radio call signs as source and destination addresses.

- ROSE, which added routing tables to the AX.25 network, but required all routes to be manually configured into the route tables and updated by hand as new nodes come on-line or drop off-line.

- The TCP/IP protocols as used on wired networks.

Of these protocols used in amateur packet radio networks, only the NET/ROM and TCP/IP suites provide the functionality of DSR and the other protocols described in this thesis. The remainder of the protocols require direct human intervention to deal with any change in routing, be it caused by a new node appearing, a node disappearing, or a node moving.

1.3.2. Military Packet Radio Networks

In contrast to the amateur packet radio networks, which often focused on the hardware, military packet radio networks from their very beginning developed hardware, software, and protocols that could adapt to the changing topologies and environments that were expected on a battlefield. Freeversyser and Leniner [33] provide a good overview of the United States' military packet radio efforts.

Growing out of ALOHANET, the DARPA-sponsored Packet Radio Network (PRNET) project extended the single hop packet radio into a mult-hop packet radio network in a series of development efforts stretching from 1972 to 1983 [59]. Unlike most amateur packet radio networks, the PRNET project designed and tested protocols in environments where the nodes were expected to be mounted on mobile platforms, such as trucks. As a result, the protocols had to adapt automatically to changes in topology, although routes were expected to remain stable for "at least a few minutes, if not longer" [59, p. 1480].

As of 1987, PRNET supported 138 nodes, either packet radios or attached hosts, and it used a flat distance vector protocol for routing. Chapter 5 conducts a comparison of the performance of a distance vector protocol with that of DSR. PRNET used its own non-IP routing header on packets, but could encapsulate IP data packets and connect to the Internet via gateways. PRNET did not address scalability to large numbers of node, high rates of mobility, quality of service, or multicast [58]. When the PRNET project ended, the Survivable Adaptive Networks (SURAN) project began, and ran from 1983 to 1992 [64]. SURAN was followed by the Global Mobile Information Systems (GloMo) project [65] that ran from 1995 to 2000.

Contemporaneously with the work on DSR described in this thesis, the United States Army began the Near Term Digital Radio project (NTDR) [100] to develop a tactical packet radio for deployment in battlefield settings. The goal was to support 400 nodes in a metropolitan sized area. NTDR uses a two-level routing hierarchy. Nodes are first organized into clusters by an elected clusterhead. The clusterhead then participates in a link-state routing protocol with other clusterheads to form the network backbone. The use of clustering reduces the amount of information that must be propagated by the link-state protocol. NTDR also provided multicast and limited support for quality of service, in the form of header bits that indicate handling precedence [110]. Published data on the performance of NTDR is not available, and this thesis did not evaluate a similar enough protocol to enable a comparison between DSR and NTDR. As an example of the importance of multi-hop ad hoc wireless networks, however, NTDR equiped radios have been a part of the US Army's First Digital Division's "go-to-war" equipment since 1998, and they are used by the armed forces of a number of other countries as well [28].

1.4. Chapter Summary

Now that low-cost high-speed wireless transceivers are available for commonly used computing equipment, there is significant industry interest in supporting wireless networking. Most work to date has focused on single-hop connectivity, although when combined with protocols like DSR, which could even be implemented directly inside the network interface card, the new hardware opens the door to the general usage of multi-hop wireless ad hoc networks.

The question that drives this thesis is, "Is it possible to build a protocol that works in an ad hoc network, and that scales dynamically across a range of environments — successfully delivering packets in the worst environments, and still taking advantage of better environments to lower its overhead?" This thesis gives an existence proof, in the form of DSR, that it is possible to build an adaptive routing protocol that operates successfully in a wide variety of ad hoc network environments.

Chapter 2

DSR Overview

This chapter gives a brief overview of the operation of the DSR protocol, providing only enough detail that the reader can understand the analysis of DSR in the following chapters. DSR remains a evolving protocol, as the lessons from our experimentation with it are folded back into it. Version 3 of the Internet Draft defining DSR [16] provides a detailed description of the version of DSR used in the experiments described in this thesis, and it is attached as an appendix to this thesis.

The Dynamic Source Routing protocol (DSR) [55, 54, 49, 51] is based on source routing, which means that the originator of each packet determines an ordered list of nodes through which the packet must pass while traveling to the destination. The key advantage of a source routing design is that intermediate nodes do not need to maintain up-to-date routing information in order to route the packets that they forward, since the packet's source has already made all of the routing decisions. This fact, coupled with the entirely on-demand nature of the protocol, eliminates the need for any type of periodic route advertisement or neighbor detection packets.

The DSR protocol consists of two basic mechanisms: Route Discovery and Route Maintenance. Route Discovery is the mechanism by which a node **S** wishing to send a packet to a destination **D** obtains a source route to **D**. To reduce the cost of Route Discovery, each node maintains a Route Cache of source routes it has learned or overheard. Route Maintenance is the mechanism by which a packet's originator **S** detects if the network topology has changed such that it can no longer use its route to the destination **D** because some of the nodes listed on the route have moved out of range of each other. Figure 2.1 shows the basic operation of the DSR protocol.

2.1. Source Routing

The routes that DSR discovers and uses are *source routes.* That is, the sender learns the complete, ordered sequence of network hops necessary to reach the destination, and, at a conceptual level, each packet to be routed carries this list of hops in its header. The key advantage of a source routing design is that intermediate nodes do not need to maintain up-to-date routing information in order to route the packets that they forward, since the packets themselves already contain all the routing decisions.

Aggregating information about the network topology at the source of each packet allows the node that cares most about the packet, namely its source, to expend the appropriate amount of effort to deliver the packet. It also enables the explicit management of the resources in the ad hoc network [70]. In some sense, DSR has an even stronger "end-to-end philosophy" than the Internet itself. Intermediate nodes in a DSR network maintain even less state than nodes in the core of the Internet, which must maintain up-to-date routing tables for all destinations in the network.

Basing the routing protocol on source routes also has two additional benefits. First, the protocol can be trivially proved to be loop-free, since the source route used to control the routing of a packet is, by definition, of finite length, and it can be trivially checked for loops. Second, each source route is a statement that a particular path is believed to exist through the network. As source routes travel through the network riding on control packets, such as ROUTE REQUESTs or ROUTE REPLYs, or the data packets whose forwarding

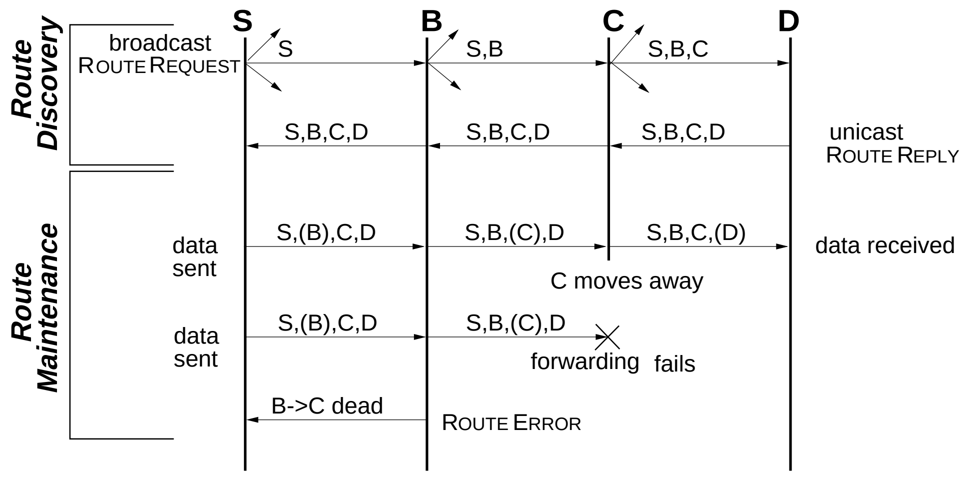


Figure 2.1 Basic operation of the DSR protocol showing the building of a source route during the propagation of a ROUTE REQUEST, the source route's return in a ROUTE REPLY, its use in forwarding data, and the sending of a ROUTE ERROR upon forwarding failure. The next hop is indicated by the address in parentheses.

they control, any node overhearing a source route can incorporate the information it contains into its Route Cache. At the cost of no overhead above that used to carry out the normal operation of the protocol, the protocol itself spreads topology information among the nodes in the network. The information carried by a source route on a data packet also has the useful property that the more frequently heard routes and the most recently heard routes are the most likely to contain accurate information, since those routes are currently being tested by the packets flowing along them.

Although DSR uses source routes, and each packet is routed based on a discovered source route, recent improvements to DSR have made it so that most packets do not need to incur the overhead of carrying an explicit source route header [16, 70]. This allows DSR to achieve the benefits of information aggregation, loop-freedom, and source routing with out the significant overhead cost in terms of bytes shown in Section 5.4.1.

2.2. Route Discovery

Route Discovery works by flooding a request through the network in a controlled manner, seeking a route to some target destination. In its simplest form, a source node **A** attempting to discover a route to a destination node **D** broadcasts a ROUTE REQUEST packet that is re-broadcast by intermediate nodes until it reaches **D**, which then answers by returning a ROUTE REPLY packet to **A**. Many optimizations to this basic mechanism are used to limit the frequency and spread of Route Discovery attempts. The controlled flood approach used by DSR works well in wired networks, but it is particularly well-suited to the nature of many wireless networks, where the communication channel between nodes is often inherently broadcast. A single transmission of a ROUTE REQUEST is all that is needed to repropagate the REQUEST to all of a node's neighbors.

Figure 2.2 illustrates a simple Route Discovery. Before originating the ROUTE REQUEST, node **A** chooses a *request_id* to place into the REQUEST such that the pair <originating address,request_id> is

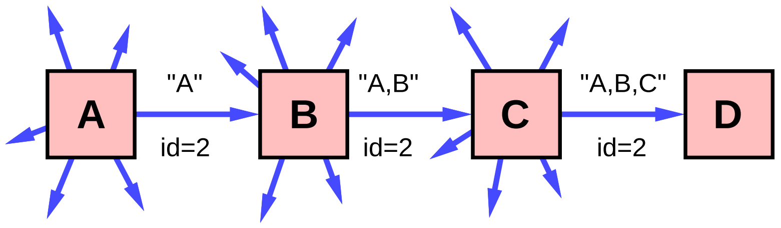


Figure 2.2 Basic operation of Route Discovery.

globally unique.[1] As the REQUEST propagates, each host adds its own address to a route being recorded in the packet, before broadcasting the REQUEST on to its neighbors (any host within range of its wireless transmission). When receiving a REQUEST, if a host has recently seen this request_id or if it finds its own address already recorded in the route, it discards that copy of the REQUEST and does not propagate that copy further.

As a result of the duplicate check in the recorded source route, the algorithm for Route Discovery explicitly prohibits ROUTE REQUESTS from looping in the network. This is an important correctness property and is responsible for the loop-free property of DSR. The use of request_ids represents a simple optimization that results in the ROUTE REQUESTs primarily spreading outwards from the originator, as shown in Figure 2.3, and curtails the number of REQUEST packets that circle around the originator. As an optimization, the protocol will still function correctly if the request_ids are either not used or not cached long enough to prevent lateral movement of an outward propagating ROUTE REQUEST, though the overhead will be higher.

While propagating a ROUTE REQUEST, nodes obey the normal rules for processing the Hop-Count or Time-to-Live field in the IP header of the packet carrying the ROUTE REQUEST. This mechanism can be

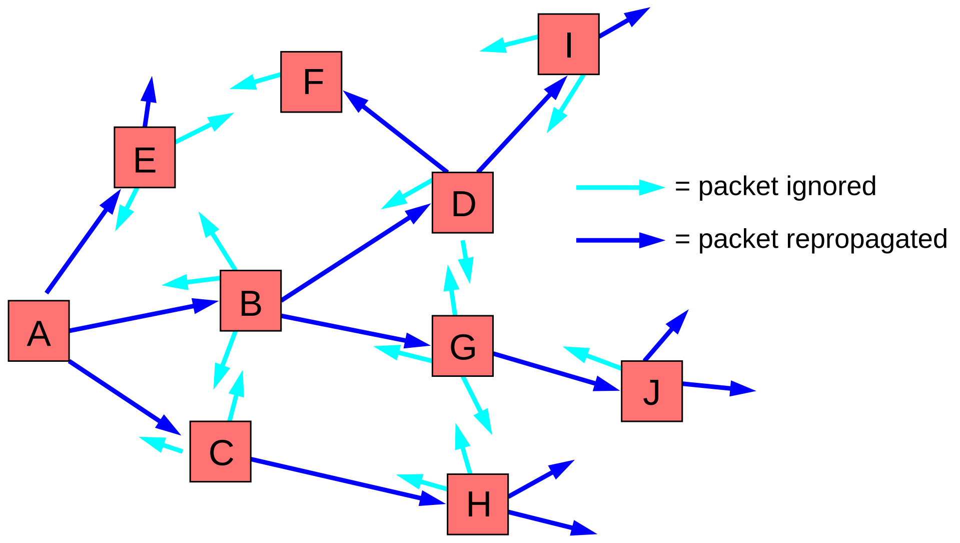


Figure 2.3 The use of request_ids constrains the propagation of ROUTE REQUESTs into an organized outward-moving wave-front.

[1]The Internet-Draft [16] explains in detail how node crashes and request_id duplication are handled.

used to implement a wide variety of “expanding ring” search strategies for the target, in which the hop limit is gradually increased in subsequent retransmissions of the ROUTE REQUEST for the target. A simple, two-phase expanding ring search is described in Section 2.5.1.

2.3. Route Maintenance

When sending or forwarding a packet to some destination **D**, Route Maintenance is used to detect if the network topology has changed such that the route used by this packet has broken. Each node along the route, when transmitting the packet to the next hop, is responsible for detecting if its link to the next hop has broken. In many wireless MAC protocols, such as IEEE 802.11 [43], the MAC protocol retransmits each packet until a link-layer acknowledgment is received, or until a maximum number of transmission attempts have been made. Alternatively, DSR may make use of a *passive acknowledgment* [58] or may request an explicit network-layer acknowledgment. When the retransmission and acknowledgment mechanism detects that the next link is broken, the detecting node returns a ROUTE ERROR packet to the original sender **A** of the packet. The sender **A** can then attempt to use any other route to **D** that is already in its route cache, or can invoke Route Discovery again to find a new route for subsequent packets.

2.4. Route Cache

All the routing information needed by a node participating in an ad hoc network using DSR is stored in a Route Cache. Each node in the network maintains its own Route Cache, to which it adds information as it learns of new links between nodes in the ad hoc network, for example through packets carrying either a ROUTE REPLY or a source route. Likewise, the node removes information from the cache as it learns previously existing links in the ad hoc network have broken, for example through packets carrying a ROUTE ERROR or through the link-layer retransmission mechanism reporting a failure in forwarding a packet to its next-hop destination. The Route Cache is indexed logically by destination node address, and supports the following operations:

void Insert(Route RT) Inserts information extracted from source route RT into the Route Cache.

Route Get(Node DEST) Returns a source route from this node to DEST (if one is known).

void Delete(Node FROM, Node TO) Removes from the route cache any routes which assume that a packet transmitted by node FROM will be received by node TO.

There is tremendous room for innovation inside the interface defined for the Route Cache, and this is intentional. An implementation of DSR may choose for its Route Cache whatever cache replacement and cache search strategy are most appropriate for its particular network environment. For example, some environments may choose to return the shortest route to a node (the shortest sequence of hops), while others may select an alternate metric for the Get() operation.

I have experimented with many different types of Route Cache, and found that several general principles are helpful.

- The Route Cache should support storing more than one source route for each destination.
- If a node **S** is using a source route to some destination **D** that includes intermediate node **N**, **S** should shorten the route to destination **D** when it learns of a shorter route to node **N** than the one that is listed as the prefix of its current route to **D**. However, the cache should still retain the ability to revert to the older, longer route to **N** if the shorter one does not work.

- The Route Cache replacement policy should allow routes to be categorized based upon "preference", where routes with a higher preferences are less likely to be removed from the cache. For example, a node could prefer routes for which it initiated a Route Discovery over routes that it learned as the result of promiscuous snooping on other packets. In particular, a node should prefer routes that it is presently using over those that it is not.

For the studies performed as part of this thesis, I primarily used a cache called the *mobicache*. The mobicache is composed of two separate caches called the *primary cache* and *secondary cache*. On a Get() operation, the primary cache is searched first, followed by the secondary cache if there is not a hit in the primary. When there is a cache hit in the secondary cache, the retrieved route is promoted to the primary cache. Routes learned as result of Route Discoveries, or other actions deliberately taken by the node, are inserted into the primary cache. Routes learned opportunistically are inserted into the secondary cache, and they will be promoted to the primary cache only if used. Each cache runs an independent replacement strategy based on a round-robin scheme. The division of the cache into primary and secondary caches was made to prevent opportunistic routes from competing for cache space with routes of known value to the node.

Myself and others have also experimented with numerous other cache strategies, including some that aggressively attempt to shorten the cached routes based on learned information [51], and some based on link-state ideas [41]. The performance achieved by DSR is significantly effected by the cache algorithms use, but this thesis demonstrates how the fundamental on-demand nature of DSR delivers good performance even when used with a simple cache. A very interesting area for future work is refining the cache algorithms.

2.5. DSR Optimizations

2.5.1. Optimizations to Route Discovery

Nonpropagating ROUTE REQUESTS. When performing Route Discovery, nodes first send a ROUTE REQUEST with the maximum propagation limit (hop limit) set to zero, prohibiting their neighbors from rebroadcasting it. At the cost of a single broadcast packet, this mechanism allows a node to query the route caches of all its neighbors for a route and optimizes the case in which the destination node is adjacent to the source. If the nonpropagating ROUTE REQUEST fails to elicit a reply within a 30 ms time limit, a propagating ROUTE REQUEST with a hop limit set to the maximum value is sent. The 30 ms timeout was chosen based on the distribution of REPLY latencies shown in Figure 4.2.

Replying from cache. If a node receives a ROUTE REQUEST for a destination **D** to which it has a route, the node may generate a ROUTE REPLY based on its cached information instead of rebroadcasting the ROUTE REQUEST. This optimization is intended to both reduce the latency of ROUTE REPLYs and prevent ROUTE REQUESTs from flooding through the entire network.

Gratuitous ROUTE REPLIES. When a node operating in promiscuous receive mode overhears a packet for which it is *not* the next hop, it scans through the list of addresses in the unprocessed portion of the source route looking for its own address. If its address is listed in this part of the source route, the node knows that packet could bypass the unprocessed hops preceding it in the source route. The node then sends a gratuitous ROUTE REPLY message to the packet's source, giving it the shorter route without these hops. Upon receiving the ROUTE REPLY, the originator will insert the shorter route into its route cache. The Route Cache can store more than one route to a destination, so the shorter route will not necessarily overwrite the longer route already in the cache. If the shorter route is found not to work, the originator can immediately revert to the longer route.

Preventing ROUTE REPLY Storms. The ability for nodes to reply to a ROUTE REQUEST not targeted at them by using their Route Caches can result in a ROUTE REPLY storm. If a node broadcasts a Route

Request for a node that its neighbors have in their Route Caches, each neighbor may attempt to send a ROUTE REPLY, thereby wasting bandwidth and increasing the rate of collisions in the area. When the target the Route Discovery is a node with which many nodes communicate, such as a server, these simultaneous replies from the mobile nodes receiving the broadcast may create packet collisions among some or all of these replies and may cause local congestion in the wireless network. In addition, it will often be the case that the different replies will indicate routes of different lengths, since some nodes will be closer to the server than others.

If nodes are able to promiscuously listen to the channel, they can reduce the number of ROUTE REPLYs sent to the originator of the REQUEST by deferring their REPLY for a time period based on the length of the source route in their REPLY. If a node with a deferred REPLY hears the originator of the Route Discovery use a source route to the target shorter than the one it is deferring, the node knows that the originator already has a shorter route to the destination, and it can cancel its deferred ROUTE REPLY.

2.5.2. Optimizations to Route Maintenance

Salvaging. When an intermediate node forwarding a packet discovers that the next hop in the source route for the packet is unreachable, it examines its route cache for another route to the same destination. If a route exists, the node replaces the broken source route on the packet's header with the route from its cache and retransmits the packet. If a route does not exist in its cache, the node drops the packet — it does not send a ROUTE REQUEST. In either case, the node attempting to perform salvaging returns a ROUTE ERROR to the source of the data packet.

The intermediate node itself does not initiate a Route Discovery to attempt to heal the broken link because it is likely to be wasteful of network resources. The originating node is likely to have an alternate route to the destination, so any work done to heal the broken route is unneeded extra overhead. If the design of DSR were to change so that intermediate nodes did attempt Route Discovery to heal the break, DSR would need a new type of ROUTE REQUEST packet that can request a route to any of several destinations. Since the intermediate node does not know which of the nodes between itself and the final destination is the best node at which to rejoin the old route, it should send a ROUTE REQUEST targeting all of them. Simply requesting a route to what had been the next node on the broken route may result in a very suboptimal repair.

Gratuitous ROUTE ERRORS. When a source **S** receives a ROUTE ERROR for a packet that it originated, **S** propagates this ROUTE ERROR to its neighbors by piggybacking it on its next ROUTE REQUEST. In this way, stale information in the caches of nodes around **S** will not generate ROUTE REPLYs that contain the same invalid link for which **S** received a ROUTE ERROR.

2.5.3. Optimizations to Caching Strategies

Snooping. When a node forwards a data packet, it "snoops" on the unprocessed portion of the source route and adds to its cache the route from itself to the final destination listed in the source route.

Tapping. Nodes operate their network interfaces in *promiscuous* mode, disabling the interface's 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. This optimization allows a node to prime its route cache with potentially useful information, while causing no additional use of the limited network bandwidth.

2.6. Intended Scope for the DSR Protocol

Figure 2.4 depicts the most basic axes that can be used to characterize the environments in which routing protocols for ad hoc networks might be deployed. The first axis is the number of nodes that the network

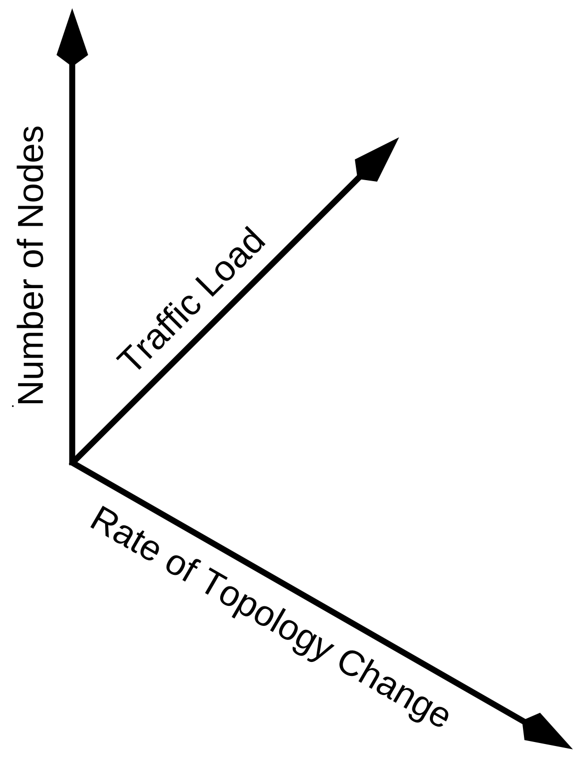


Figure 2.4 The basic axes along which routing protocols for ad hoc networks are stressed.

must be able to route among. The second axis is the rate at which the topology of the network changes. The third axis is the traffic load on the network. Traffic load is most accurately measured as the expected link utilization of the network, as it is the stability of the link bandwidth and the ratio of the offered load in bits-per-second to the link bandwidth that is most important to protocol behavior. For example, consider the difference between a 10Kbps load place on a network with 9.6Kbps links to the same load applied to a network with 1Mbps links. The formal load place a much greater strain on the routing protocol than the latter load, since congestion may will prevent any routing packets from being sent.

As the environment moves away from the origin of the three axes, the routing problem becomes harder. Increasing the number of nodes, increasing the rate of topology change (e.g., increasing the node's mobility), or increasing the offered traffic load to the network all challenge the routing protocol in different ways. It is not surprising that different routing protocols excel in different parts of the space of challenges.

DSR, as it is currently implemented, is optimized for operation in a large region that roughly covers the lower part of the space of environments. It functions with extremely low overhead across a very wide range of mobility rates and rates of topology change, from a static topology to one with constant motion. It also functions with very high performance across traffic loads varying from extremely low bit-rates measured in packets per minute to traffic loads that saturate the radio medium.

Due to its use of flood-fill broadcasts in Route Discovery, however, DSR as described in this thesis is limited in terms of the maximum size of the network it can support with high performance. Using the techniques described in this thesis, DSR is most applicable to networks of 500 nodes or fewer. Above that size, it becomes harder to maintain good containment (Section 3.3.2) of Route Discoveries when searching for reachable nodes, and Route Discoveries for unreachable nodes begin to have serious deleterious effect on the network. Broch and myself have begun working on techniques to support hierarchy inside DSR networks [17] that may enable DSR to achieve the same performance in large networks that it does in small networks. The techniques improve the containment of Route Discovery and Route Maintenance by leveraging the natural hierarchy often found in the organizations most likely to use ad hoc networks.

Chapter 3

Evaluation of Protocols for Ad Hoc Networks

This chapter describes a methodology for evaluating protocols in an ad hoc network environment, concentrating on the simulation system and the metrics I used for analyzing and comparing protocols later in this thesis.

The value of simulation in studies of protocols is that it allows near perfect experimental control: experiments can be designed at will and then rerun while varying an experimental variable and holding all other variables constant. With simulation, it is also possible to test the behavior of networks with more nodes than physical equipment is available for, or networks with equipment that does not even exist yet.

The drawback of simulation is that inherently runs the risk of oversimplification. It is not possible to exactly replicate the entire world inside a computer model, so when creating a simulation some factors must be statistically or otherwise approximated. The failure to properly capture the behavior of first-order factors can lead to dramatically incorrect results.

Working with Josh Broch and the other members of the Monarch Project, we have created a detailed packet-level simulator that supports the modeling of arbitrary movement patterns, a variety of MAC and physical layers, and the Address Resolution Protocol (ARP) [88]. Our simulation environment is significantly more detailed than those that preceded it, and the results presented in later chapters demonstrate significantly different conclusions than some earlier studies precisely because of this extra detail. However, our simulator is still not perfect. For example, the propagation model in our simulator, while more detailed than others, still fails to model a type of packet loss that appears to be significant, as shown in results from the full-scale testbed reported in Chapter 7. As a compromise between simulation and full-scale implementation, in Chapter 6 I present several types of emulation that can be used to balance the risks of simulation with its benefits.

3.1. Extensions to ns-2 Simulator

ns-2 is a discrete event simulator developed by the University of California at Berkeley and the VINT project [30]. Prior to our work on it, it had established itself as a prominent environment for studying TCP and other protocols over networks like the conventional wired Internet. However, it did not provide the support needed for accurately simulating the physical aspects of multi-hop wireless networks or the MAC protocols needed in such environments. Our extensions to *ns-2* as described below have now been adopted by the VINT project as the general support for modeling ad hoc networks in *ns-2* and included into their mainstream releases.

Berkeley has also released *ns-2* code that provides some support for modeling wireless LANs, but this code can not be used for studying multi-hop ad hoc networks as it does not support the notion of node position; there is no spatial diversity (all nodes are in the same collision domain), and it can only model directly connected nodes.

Significantly more details on our extensions to *ns-2* are available in our documentation of the extensions [69]. Additionally, many helpful scripts, Frequently-Asked-Questions answers, and other information is available from the Monarch web site [91].

3.1.1. Propagation Model

To accurately model the attenuation of radio waves between antennas close to the ground, radio engineers typically use a model that attenuates the power of a signal as $1/r^2$ at short distances (r is the distance between the antennas), and as $1/r^4$ at longer distances. The crossover point is called the *reference distance*, and is typically around 100 meters for outdoor low-gain antennas 1.5m above the ground plane operating in the 1–2GHz band [94]. Following this practice, the signal propagation model used by the simulator combines both a free space propagation model and a two-ray ground reflection model. When a transmitter is within the reference distance of the receiver, the free space model where the signal attenuates as $1/r^2$is used. Outside of this distance, the simulator uses the ground reflection model where the signal falls off as $1/r^4$.

3.1.2. Node Model

The model for each mobile node has a position and a velocity and moves around on a topography that is specified using either a digital elevation map or a flat grid. The position of a mobile node can be calculated as a function of time, and is used by the radio propagation model to calculate the propagation delay from one node to another and to determine the power level of a received signal at each mobile node.

Each mobile node has one or more wireless network interfaces, with all interfaces of the same type (on all mobile nodes) linked together by a model of a single physical channel. When the model of a network interface transmits a packet, it passes the packet to the appropriate physical channel object. This object then computes the propagation delay from the sender to every other interface on the channel and schedules a "packet reception" event for each. This event notifies the model of the receiving interface that the first bit of a new packet has arrived. At this time, the power level at which the packet was received is compared to two different values: the carrier sense threshold and the receive threshold. If the power level falls below the carrier sense threshold, the packet is discarded as noise. If the received power level is above the carrier sense threshold but below the receive threshold, the packet is marked as a packet in error before being passed to the MAC layer. Otherwise, the packet is simply handed up to the MAC layer.

Once the model of the MAC layer receives a packet, it checks to insure that its receive state is presently "idle." If the receiver is not idle, one of two things can happen. If the power level of the packet already being received is at least 10 dB greater than the received power level of the new packet, we assume capture, discard the new packet, and allow the receiving interface to continue with its current receive operation. Otherwise, a collision occurs and both packets are dropped.

If the MAC layer is idle when an incoming packet is passed up from the network interface, it simply computes the transmission time of the packet and schedules a "packet reception complete" event for itself. When this event occurs, the MAC layer verifies that the packet is error-free, performs destination address filtering, and passes the packet up the protocol stack.

3.1.3. Medium Access Control

The link layer of our simulator includes a simulation model of the complete IEEE 802.11 standard [43] Medium Access Control (MAC) protocol Distributed Coordination Function (DCF) in order to accurately model the contention of nodes for the wireless medium. DCF is similar to MACA [60] and MACAW [10] and is designed to use both *physical carrier sense* and *virtual carrier sense* mechanisms to reduce the probability of collisions due to hidden terminals. Virtual carrier sense attempts to "sense" the presence of carrier near the intended receiver of a packet before transmitting. If either carrier sense mechanism indicates that the wireless medium is busy, the node defers before transmitting, using a binary exponential backoff.

The virtual carrier sense mechanism uses two short packets before the intended data packet to acquire the channel: a Request-to-Send (RTS) and a Clear-to-Send (CTS). When a node **A** wants to transmit a packet to node **B**, it first transmits an RTS to **B**, and upon receipt of this packet, node **B** responds with a CTS to **A** if both its physical and virtual carrier sense mechanisms indicate that the medium is idle. When the CTS is received by **A**, **A** transmits the data packet, and when **B** receives the data packet, it returns a link-layer Acknowledgment (ACK) packet to **A**. All packets sent by 802.11 carry in their headers a *Duration* field that indicates how much longer nodes that receive the packet should consider virtual carrier to be present, and nodes listen to and obey the virtual carrier implied by the Duration field in all packets that they receive, even if not addressed to them.

If node **A** fails to receive a CTS after transmitting an RTS, or if it fails to receive an ACK after transmitting a data packet, it will retry up to four times before giving up. If it gives up, the data packet is handed back up to the network layer as "undeliverable." The routing protocol in the network layer is then free to retry the transmission, select another route for the packet, or discard the packet.

As an exception to the RTS/CTS/Data/ACK exchange described above, 802.11 does not use this mechanism for the transmission of a data packet to a broadcast destination address. As MAC-layer broadcasts are not transmitted to a specific destination, the channel cannot be reserved with an RTS/CTS exchange. Similarly, these packets cannot be acknowledged and are thus less reliable than unicast packets 5.4.3.

3.1.4. Address Resolution

Since the routing protocols all operate at the network layer using IP addresses, a simulation model of ARP [88], following the design of the BSD Unix implementation [111], was included in the simulation and used to resolve IP addresses to link layer addresses. The broadcast nature of an ARP REQUEST packet (Section 5.4.3) and the interaction of ARP with on-demand protocols (Section 5.4.4) make ARP an important detail of the simulation.

3.1.5. Validation of the Propagation Model and MAC Layer

The propagation model in the Monarch Project's simulator uses standard equations and techniques, and it was verified by an expert in radio propagation modeling. We analyzed the power and simulated radio behavior as a function of distance between small groups of nodes to ensure that the propagation, capture, and carrier sense models were working as designed.

The 802.11 MAC implementation was studied in a variety of scenarios and independently verified by two members of the Monarch Project. We experimentally tested that when all nodes are in range of each other, no data packets experience collisions (regardless of offered traffic load), and that each node is able to make progress sending packets. This verified that the carrier sense, RTS/CTS, and back-off mechanisms of 802.11 were working correctly.

3.2. Simulation Methodology

Figure 3.1 shows the basic methodology of our simulation experiments. Each run of the simulator accepts as input a *scenario file* that describes the exact motion of each node and the exact sequence of packets originated by each node, together with the exact time at which each change in motion or packet origination is to occur. The detailed trace file created by each run was stored to disk, and analyzed using a variety of scripts, particularly one called `totals.pl` that counts the number of packets successfully delivered and the length of the paths taken by the packets, as well as additional information about the the internal functioning of each protocol. This data was further analyzed in matlab to produce the graphs shown in this thesis.

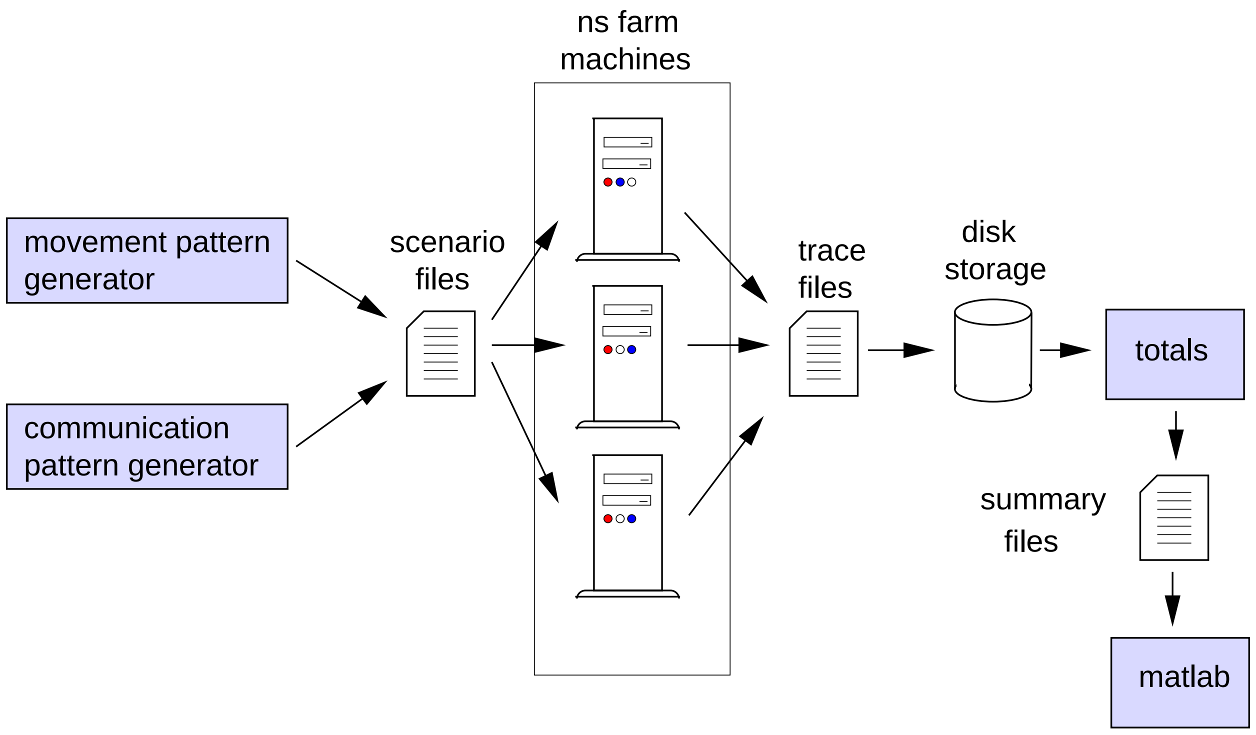


Figure 3.1 Overview of our simulation methodology.

3.2.1. Movement Models

Most of the studies reported in this thesis are made with nodes that move according to the *Random Waypoint* model [51]. Random Waypoint scenarios are characterized by a *pause time* value that affects how often nodes move during the scenario, which in turn affects the amount of topology change. The scenarios are generated by first picking a value for the *pause time* of the scenario, and then following this algorithm:

> Each node begins the simulation at some randomly chosen location inside the simulation site. After the simulation starts, the node remains stationary for *pause time* seconds. It then selects a random destination in the site and moves to that destination at a speed distributed uniformly between 0 and some maximum speed. Upon reaching the destination, the node pauses again for *pause time* seconds, selects another destination, and proceeds there as previously described, repeating this behavior for the duration of the simulation.

The Random Waypoint algorithm can be thought of as modeling nodes as actors that go to a place, perform some task there, and then go on to the next place to perform a task. Random Waypoint motion is trivial to compute and represent, and the algorithm produces “more natural” motion than algorithms which involve bouncing off walls or moving by Brownian motion. Under Random Waypoint, nodes’ motions are uncorrelated, and nodes are as likely to transit long distances before pauses as short distance. Due to the variation in distance each node travels, after the first pause time, nodes do not necessarily all pause at the same time.

3.2.2. `ad-hockey` Tool for Visualization

While quantitative comparison between protocols is best achieved by comparing the performance of the protocols over a range of scenarios, as described above, protocol designers also need tools that help them understand the behavior of their protocols.

There is tremendous value in analyzing the trace files produced by *ns-2* via a variety of simple techniques. For example, I designed the trace file format to make it easy to slice the trace file using `egrep` to show all the packets handled or actions taken a node, or to track an individual packet as it passes through the network. However, even these simple techniques require that *ns-2* be run on some scenario, and creating scenarios by hand is tedious. More importantly, these simple techniques do not leverage the visual ability of humans to "grok" a network when the information is displayed in two-dimensional graphic form, rather than the single dimensional form of a trace slice.

In order to support my design of the DSR protocol, I created *ad-hockey* [68]. *ad-hockey* is a Perl/Tk program that can assist in the creation of scenario files for use by the CMU Monarch extensions to *ns-2* and the visualizations of the simulation trace files. This tool has proven generically useful, both for experimenting with DSR and other protocols for ad hoc networks and as a visualization system for the types of emulation described in Chapter 6.

3.3. Evaluation Metrics

In evaluating DSR and the other protocols studied in this thesis, I used several sets of metrics. To characterize the basic performance of the protocols, I used a set of high-level summary metrics that are of interest to network users. To understand the internal functioning of the protocols, I used other sets of metrics: some of which are protocol specific and described as needed in the text, and some of which are general to all on-demand routing protocols and described below.

3.3.1. Summary Metrics

The following three metrics capture the most basic overall performance of DSR and the other protocols studied in this thesis:

- **Packet delivery ratio**: The ratio between the number of packets originated by the "application layer" sources and the number of packets received by the sinks at the final destination.
- **Routing overhead**: The total number of routing packets transmitted during the simulation. For packets sent over multiple hops, *each* transmission of the packet (each hop) counts as one transmission. Routing packets are those that are originated by the routing protocol and do not also include user data. For protocols like DSR, which include both routing data and user data in the same packet, all the bytes of routing data in the packets are counted as routing overhead as described in more detail below.
- **Path optimality**: The difference between the number of hops a packet took to reach its destination and the length of the shortest path that physically existed through the network when the packet was originated.

Packet delivery ratio is important as it describes the loss rate that will be seen by the transport protocols, which in turn effects the maximum throughput that the network can support. This metric characterizes both the completeness and correctness of the routing protocol.

Routing overhead is an important metric for comparing these protocols, as it measures the scalability of a protocol, the degree to which it will function in congested or low-bandwidth environments, and its efficiency in terms of consuming node battery power. Protocols that send large numbers of routing packets can also increase the probability of packet collisions and may delay data packets in network interface transmission queues. I did not include the number of IEEE 802.11 MAC packets or ARP packets in routing overhead, since the routing protocols I studied could be run over a variety of different medium access or address resolution protocols, each of which would have different overhead.

I also evaluated routing overhead in terms of total number of bytes of routing information transmitted. If a packet contains only routing information, the entire size of the packet, including IP headers, is counted as byte overhead. For packets that carry both user data and routing data, such as the source routes used by DSR, we counted the overhead as only the number of bytes in the source route and its associated subheader.

In the absence of congestion or other "noise," path optimality measures the ability of the routing protocol to efficiently use network resources by selecting the shortest path from a source to a destination. Path optimality is calculated as the difference between the shortest path found internally by the simulator when the packet was originated, and the number of hops the packet actually took to reach its destination. Because packets can be stored in the buffers of nodes while the nodes move, it is possible for the length of the optimal path for a packet to take through the network to change between the time the packet is originated and when it is received by the destination. As a result, packets are occasionally received by the destination after traveling fewer hops than the optimal path computed when the packet was originated. The effect is negligible in most experiments reported in this thesis, but does become more pronounced as the packet buffer time, and hence the packet delay, increases.

3.3.2. Measuring Route Discovery

Two additional metrics, *containment* and *discovery cost*, proved useful to evaluate the cost of on-demand Route Discovery.

- **Containment:** Containment is defined as the percentage of nodes that do not receive a particular ROUTE REQUEST. For a nonpropagating ROUTE REQUEST (Section 2.5.1), containment is equivalent to measuring the percentage of nodes in the network which are not neighbors (within transmission range) of the node originating the request. For a propagating ROUTE REQUEST, containment measures how far out the request propagates before running into either the edge of the network or a band of nodes with cached information about the target that is wide enough to stop further propagation. Values of containment approaching 1 indicate that a ROUTE REQUEST was well contained and interrupted very few nodes, whereas containment values approaching 0 indicate that most of the nodes in the network had to process the request.

- **Discovery cost:** The cost of a single Route Discovery is defined as

  $$1 + \mathrm{FwReq} + \mathrm{OgRep} + \mathrm{FwRep},$$

  where 1 represents the transmission of the original request, *FwReq* is the number of ROUTE REQUEST forwards, *OgRep* is the number of ROUTE REPLY originations, and *FwRep* is the number of ROUTE REPLY forwards. For each Route Discovery, this metric measures the number of routing packets (requests and replies) that were transmitted to complete the discovery. The average discovery cost is calculated as

  $$\frac{\mathrm{OgReq} + \sum \mathrm{FwReq} + \sum \mathrm{OgRep} + \sum \mathrm{FwRep}}{\mathrm{OgReq}},$$

  where *OgReq* is the number of ROUTE REQUEST originations, and *FwReq*, *OgRep*, and *FwRep* are summed over all Route Discoveries.

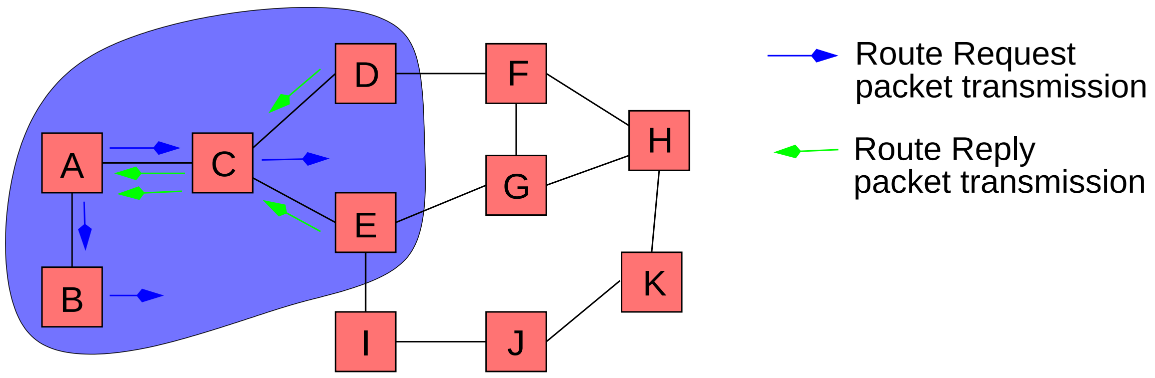

A
B
C
D
E
F
G
H
I
J
K
Route Request packet transmission
Route Reply packet transmission

Figure 3.2 Example Route Discovery with a discovery cost of 8 and a containment of 6/11 or 54%.

Figure 3.2 is an example of how containment and discovery cost are calculated for a route discovery on a hypothetical topology with lines indicating the nodes that can communicate directly. The shaded area depicts the nodes which are involved in the route discovery: receiving a ROUTE REQUEST and either returning a ROUTE REPLY or repropagating the REQUEST. Since only 5 of the 11 nodes are involved, the containment of this discovery is 54%. The cost of this discovery was 8 transmissions. Four transmissions were made to propagate the ROUTE REQUEST outwards, and four transmissions were made to return the ROUTE REPLYS.

Both the containment and discovery cost metrics of Route Discovery are sensitive to a third parameter that characterizes the topology over which the discovery is running: the average *degree* of the nodes in the network. The degree of a node is the number of direct neighbors the node has, and the average degree measures how tightly interconnected the network is. As the degree of interconnectivity goes up, it is harder to contain a ROUTE REQUEST to one part of the network. In addition, the "branching factor" of a propagating ROUTE REQUEST increases, which causes more nodes to receive and process it. Thus, we would expect containment to decrease and discovery cost to increase in environments where the average node degree increases.

Chapter 4

Simulation Analysis of DSR

The intent of this thesis is to evaluate the usefulness of on-demand mechanisms in routing protocols for ad hoc networks. At first glance this may be an easy proposition, since the mechanisms appear to align well with the requirements of the environment in which many ad hoc networks will operate. However, on-demand mechanisms bring with them many potential liabilities. In this chapter, we pose four challenges to the value of on-demand mechanisms, and then refute the challenges using data from simulations of DSR.

- *What effect does on-demand routing have on packet latency?* An on-demand routing protocol attempts to discover a route to a destination only when it is presented with a packet for forwarding to that destination. This discovery must be completed before the packet can be sent, which adds to the latency of delivering the packet. Indeed, some mechanisms to reduce the overhead cost of discovering a new route may result in an increase in latency for some Route Discovery attempts.

- *What is the overhead cost of on-demand routing behavior?* Without additional information, a protocol using on-demand routing must search the entire network for a node to which it must send packets, but does not know how to reach. Optimizations to the protocol may reduce the cost of initiating communication, but discovering a new route is likely to remain a costly operation.

- *How long does it take the protocol to recover from a broken route?* In an ad hoc environment, node movement and wireless propagation effects cause topology changes and link breakages. If a link that is part of an actively used route breaks, the potential exists for packets to be dropped until a new route is discovered and put into use. The latency of recovering from a broken route is therefore a critical factor in determining the ability of the routing protocol to successfully deliver packets, as is the ability of the routing protocol to save packets already traveling through the network when the break occurs.

- *When caching the results of on-demand routing decisions, what is the level and effect of caching and cache correctness on the routing protocol?* Any on-demand routing protocol must utilize some type of routing cache, either distributed or centralized, in order to avoid the need to re-discover each routing decision for each individual packet. However, the cache itself may contain out-of-date information indicating that links exist between nodes that are no longer within wireless transmission range of each other. This stale data represents a liability that may degrade performance rather than improve it.

While we have conducted our experiments with DSR, these challenges are general to any wireless ad hoc network routing protocol using on-demand behavior. The DSR protocol provides a good source of examples for this study, since it is based *entirely* on on-demand behavior. However, our answers to these questions can be generalized to other protocols using related mechanisms, and they validate the use of on-demand mechanisms in DSR and other protocols.

4.1. Methodology

In order to analyze the effects of the mechanisms that comprise DSR, we simulated different variations of the DSR protocol running on an identical network with an identical workload. Simulation enabled us to study a large number of points in the DSR design space and to directly compare the results of the simulations, since we were able to hold constant factors such as the communication and movement pattern between runs of the simulator. We conducted the experiments using the *ns-2* network simulator [30] extended with our support for realistic modeling of mobility and wireless communication as described in Section 3.1. Each simulation used 50 nodes and simulated 900 seconds of real time.

4.1.1. Node Movement and Communication Pattern

We analyzed the effect of two different movement models: constant node motion and no node motion. All tables and graphs in this chapter are from the constant motion case, since without motion the data are uninteresting: DSR does a single Route Discovery for each destination, and then aside from congestion-caused packet drops leading to Route Errors, there is no more protocol activity for the duration of the run. For both constant motion and no motion models, the nodes start at a uniformly distributed location. In the constant motion case, all nodes move according to the *random waypoint* algorithm described in Section 3.2.1 with speeds chosen uniformly from 0 to 20 m/s.

We chose to model node communication using a uniform node-to-node communication pattern with constant bit rate (CBR) traffic sources sending data in 512-byte packets at a rate of 4 packets per second. A total of 20 CBR connections were modeled in each simulation run, with each node being the source of 0, 1, or 2 connections. In all runs, the 20 connections were spread in this way over a total of 14 different originating nodes (which we label nodes 1 through 14). All CBR connections were started at times uniformly distributed during the first 180 seconds of simulated time and then remained active through the entire simulation. We chose the parameters of the communication pattern to stress the ability of the routing protocol to discover and maintain routes during the experiments, but to avoid causing extreme congestion.

In order to average out the effects of particular motion or communication patterns, we generated 10 different scenarios — 5 with constant node motion, and 5 with no node motion. We ran all experiments over the same set of scenarios.

4.1.2. Simulated Sites

In an attempt to generate results that would be representative of some potential, real-world scenarios that DSR might encounter, we ran our simulations over two different types of simulated sites:

- A flat site with length much greater than its width. We used dimensions of 1500m $\times$ 300m, and we call this the *rectangular site*.
- A flat site with area approximately equal to that of the rectangular site, but in which the length and width are equal. We used dimensions of 670m $\times$ 670m, and we call this the *square site*.

Altering the shape of the simulated site in this way creates networks with qualitatively different topologies and throughput bottlenecks, thereby exercising DSR in different ways.

The rectangular site causes a roughly linear arrangement of the nodes. The results seen here could be applied to situations where the physical paths between nodes are very constrained, such as when the mobile nodes are vehicles driving along a road. The lengths of the routes taken by packets in this site are typically longer than the corresponding route lengths in the square site. Since the site is also narrow, a packet being transmitted down the site is typically overheard by all the nodes spatially located between the sender and destination. This makes networks in the rectangular site more prone to congestion bottlenecks since there is effectively no spatial diversity in the narrow dimension.

The square site models situations in which nodes can move freely around each other, and where there is a small amount of path and spatial diversity available for the routing protocol to discover and use. Since its total area is approximately the same as that of the rectangular site, the average node density between the two sites is constant, but the average route length used in the square site is less than in the rectangular site.

4.2. On-Demand Routing and Latency

The use of on-demand behavior in routing protocols for multi-hop wireless ad hoc networks can result in increased packet latency, since, by the nature of the on-demand philosophy, operations are put off until the time when forward progress cannot be made without performing them. The time taken to perform the operation is therefore directly visible.

The best example of this behavior in DSR, and a large source of excess latency, is Route Discovery. For example, when some node **A** wants to send a packet to a node **B**, and it does not currently have a route to **B**, it must first perform a Route Discovery for node **B** before sending the packet. Because **A** must buffer its packet until it has a route to **B**, the entire time that Route Discovery takes to obtain a route to **B** adds directly to the time it will take for the packet to be delivered. This makes the latency of Route Discovery critical in any environment in which packets must be delivered in a timely fashion.

To review briefly the description of Route Discovery in Section 2.5.1, DSR uses an instance of the general class of expanding ring searches consisting of two phases:

- **Phase I: Nonpropagating ROUTE REQUEST**. Node **A** transmits a ROUTE REQUEST with a maximum propagation limit of one hop. Any node receiving the ROUTE REQUEST that does not have a route to **B** simply discards the request.

- **Phase II: Propagating ROUTE REQUEST**. If after 30 ms the nonpropagating ROUTE REQUEST has failed to return a route to **B**, node **A** will then transmit a propagating ROUTE REQUEST. Each node other than **B** that receives this request will either return a ROUTE REPLY based on information in its route cache, or will rebroadcast the ROUTE REQUEST, propagating it further through the network. Should **B** itself receive the request, it will return a ROUTE REPLY consisting of the source route collected in the ROUTE REQUEST.

In order to calculate the latency of a Route Discovery, we measure the time from when a node sends a ROUTE REQUEST until it receives the *first* ROUTE REPLY that answers the request. We use the first reply to calculate latency since as soon as it arrives, the node can begin transmitting data packets — it need not wait for multiple replies to be returned before sending.

As a result of the structure of this Route Discovery algorithm, the latency of Route Discovery is determined by the length of time it takes a ROUTE REQUEST to travel across the network to a node who can answer it with a ROUTE REPLY plus the time taken by the REPLY to travel back to the originator of the Route Discovery. This, in turn, implies that the latency of Route Discovery depends on the speed with which packets propagate through the network and the distribution of nodes who have the routing information needed to answer the request, since this determines the average distance between the requester and the replier.

The next sections analyze the speed with which packets can be forwarded through the network, and the factors that affect the distribution of routing information in the network.

4.2.1. The Propagation Speed of Route Requests

Examining the latency of Route Discovery when nodes do not answer ROUTE REQUESTs on behalf of other nodes makes visible the worst case speed with with which ROUTE REQUESTs and ROUTE REPLYs travel through the network. With the replying from cache optimization disabled, Route Discoveries can be answered only by the targeted node, which creates the greatest level of congestion and requires the packets to travel the furthest.

Each point in the scatter plot of Figure 4.1 depicts the latency of a particular Route Discovery versus the sum of the number of hops over which the ROUTE REQUEST and matching ROUTE REPLY had to travel. The scatter plot makes visible the distribution of individual samples, and the abscissas have been uniformly jittered to make the density of the points visible. The least squares fit line shown on the graph has a slope of 14.5 ms/hop.

The spread of latencies recorded reflect the realistic nature of our simulator. The minimum time required to forward a small packet, such as a ROUTE REQUEST or ROUTE REPLY, over a single hop is approximately 600 μs. This assumes the packet immediately acquires the media when offered to the network. The minimum per-hop forwarding time for a 512 byte data packet is approximately 3 ms. However, since the effective carrier-sense range on the simulated radios is 550 m, on average each packet sent must defer to other packets being transmitted elsewhere in the network. The scatterplot is the superposition of the dense mass of points along the bottom of the distribution with a linear dependence of latency on hop count, representing Route Discovery in uncongested cases, and a exponentially distributed scatter of points representing congested cases.

4.2.2. The Distribution of Routing Information in the Network

The distribution of useful routing information in the network can be inferred by the type of ROUTE REPLY packet received by a node in response to sending a ROUTE REQUEST. The REPLYs may be categorized as follows:

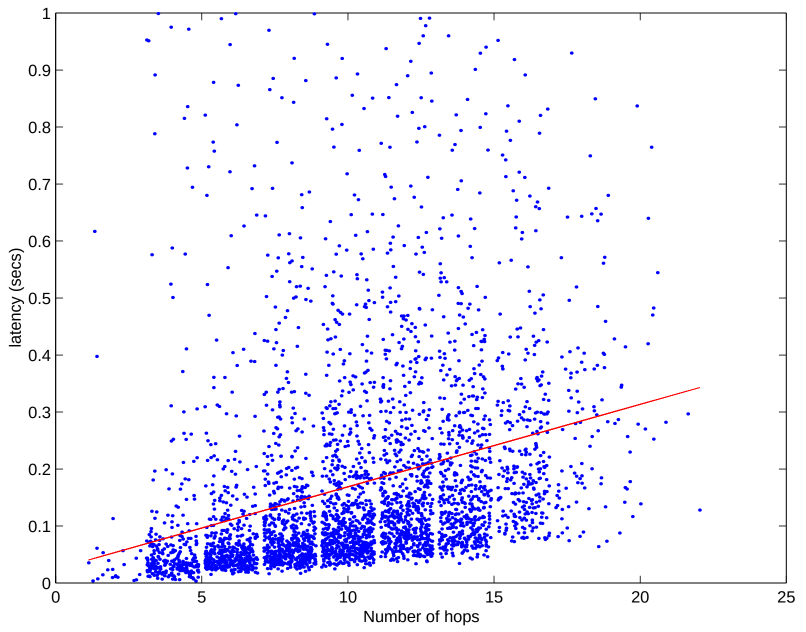


Figure 4.1 Scatter plot of forwards vs. latency. Forwards includes the number of hops from the originator of the Route Discovery to the target and from the target back to the originator. (No cache replies, rectangular site, constant motion)

- **Cache Replies**. Replies constructed from cached routing information by a node other than the target of a Route Discovery.

- **Target replies**. Replies originated by a node in response to receipt of a ROUTE REQUEST targeting it. We say that target replies are based on *fresh* routing information because the route from requester to target has just been traversed by the corresponding ROUTE REQUEST.

- **Neighbor replies**. A ROUTE REPLY returned in response to a ROUTE REQUEST with a maximum propagation limit (TTL) of 1. Neighbor replies must originate from a direct neighbor of the requester (since the request cannot propagate), but may be either cache replies or target replies depending on whether or not the respondent is the target of the Route Discovery.

When the reply from cache optimization is enabled, the reception of a Target reply means that there were no nodes closer to the originator than the final target with information about the target's whereabouts. Reception of a Neighbor reply means one of two things. In the first case, the target itself was directly reachable by the originator, and so the distribution of routing information in the network is irrelevant for this particular Route Discovery. In the second case, routing information was so well distributed that an immediate neighbor had cached information with a route to the target.

Table 4.1 summarizes the latency required for the initiator of a Route Discovery to receive the first ROUTE REPLY in response to its ROUTE REQUEST. This data is broken down into the mean, minimum, and maximum latencies for each of the three different types of ROUTE REPLYs listed above. In this section, we analyze the latency for each type of ROUTE REPLY and examine the effect of changing the shape of the site on the latency.

The Latency of Neighbor Replies

Figure 4.2 shows a detailed view of the latency of the first ROUTE REPLY packet received for nonpropagating ROUTE REQUESTs. The graph shows only those replies that were returned within 100 ms. After 30 ms, a nonpropagating request is considered unsuccessful and a propagating ROUTE REQUEST is transmitted and so only replies returned within the 30 ms timeout are useful in terms of preventing a propagating Route Discovery. Since neighbor replies with latency greater than 30 ms may be interpreted as the first reply to the propagating ROUTE REQUEST, we have included replies with a latency of up to 100 ms in our analysis. Table 4.1 summarizes all of the data, including the outliers, and presents the distribution of the data. The mean of the distribution is 7.1 ms, discarding as outliers all replies that arrived after 100 ms.

Table 4.1 Latency of first ROUTE REPLY by type
(All-Opt DSR, rectangular site, constant motion)

		Latency			
	# Replies	Mean	Min	99 percentile	Max
Neighbor Replies	3136	7.1ms	1.3ms	457ms	17.8s
Cache Replies	1524	45.9ms	1.3ms	752ms	2.4s
Target Replies	12	87.6ms	23.6ms	458ms	458.3ms

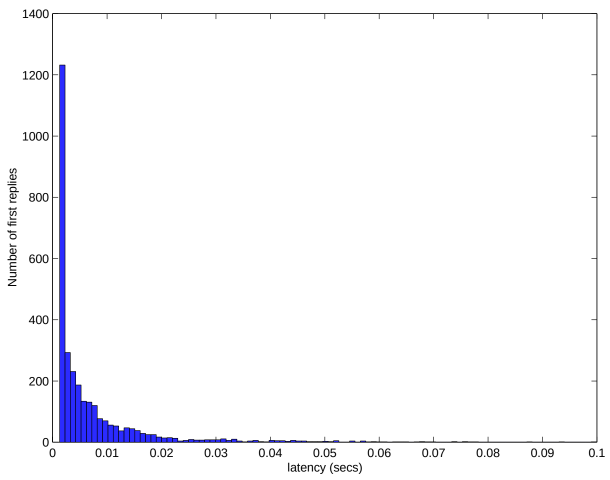


Figure 4.2 Latency of neighbor replies (All-Opt DSR, rectangular site, constant motion). Mean is 7.1 ms.

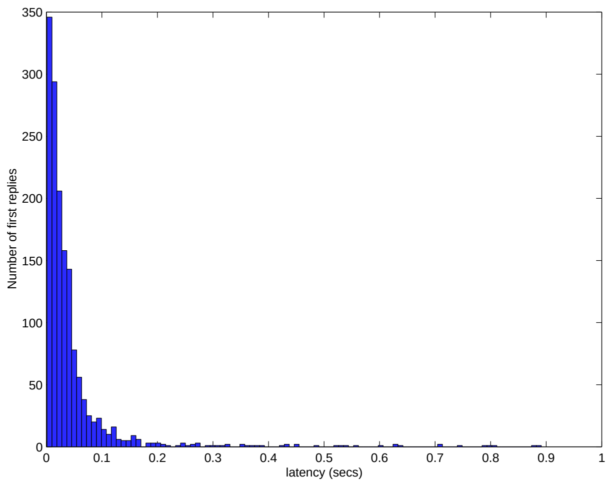


Figure 4.3 Latency of Cached ROUTE REPLY packets (All-Opt DSR, rectangular site, constant motion). Mean is 45.9 ms.

The Latency of Cache Replies

When a nonpropagating request fails to obtain a route, the node performing Route Discovery sends a propagating ROUTE REQUEST. Each node that hears this request will either transmit a ROUTE REPLY from its cache or forward the request as described above. Figure 4.3 shows a detailed view of the latency of ROUTE REPLY packets sent from cached route information received in response to a propagating ROUTE REQUEST. Again discarding outliers, the mean of this distribution is 45.9 ms (Table 4.1), nearly seven times larger than the mean latency for neighbor replies (7.1 ms). This difference between mean neighbor reply latency and the latencies of cache replies indicates that successful nonpropagating requests are very effective in reducing latency. However, the difference does not show exactly how effective they are since the cache replies in Table 4.1 are elicited by propagating requests which are only sent after a nonpropagating request has already failed.

In order to factor out the effect of nonpropagating ROUTE REQUESTS on the latency of cache replies, we removed the mechanism that sends nonpropagating requests and ran this modified version of DSR on the same set of scenarios. The results of this experiment are reported in Table 4.2 and Figure 4.4. When only propagating ROUTE REQUESTs are sent, the mean latency for cache replies dropped to 21 ms, less than half of the latency measured when nonpropagating requests were enabled. From this, we conclude

Table 4.2 Latency of first ROUTE REPLY (No nonpropagating ROUTE REQUESTs, rectangular site, constant motion)

	Mean Latency	Min Latency	Max Latency
Cache Replies	21.0ms	1.3ms	8.0s
Target Replies	33.3ms	3.8ms	63.0ms

that nonpropagating ROUTE REQUESTs offer an average savings of about 14 ms in latency (a factor of three decrease).

The Latency of Target Replies

The third row of Table 4.1 in Section 4.2.2 summarizes the latency of replies from the target of a Route Discovery. The mean of 87.6 ms is an increase of approximately 40 ms over the time for cache replies. This increased latency is in general due to the increased number of hops over which the ROUTE REQUEST and matching ROUTE REPLY must propagate to reach the target, relative to what is needed to reach the first node with a route cache entry that can return a reply from its cache.

In almost all cases, the first ROUTE REPLY received in response to a ROUTE REQUEST is not a target reply, but rather is a cache reply. In our simulations, the first ROUTE REPLY came from the target node only a total of 12 times over all All-Opt DSR scenarios studied. This small number of target replies indicates that DSR's caching mechanism and its ability to send replies from the cache are very effective in reducing the latency of Route Discovery, but this limits our ability to directly evaluate the latency of target replies, and the latency for target replies shown in Table 4.1 represents only these 12 samples.

As another way of measuring the latency of target replies, we disabled both nonpropagating ROUTE REQUESTs and replying from cache in DSR and ran this modified version of the protocol over the same scenarios as above. Table 4.3 summarizes the target reply latencies from this experiment, and Figure 4.5 shows a detailed view of this latency. The mean latency for target ROUTE REPLYs in this experiment (the only type enabled) was 403.1 ms. This indicates that the optimizations added to DSR (replies from cache and non-propagating route requests) significantly decrease the latency of Route Discovery. The reason latency is higher with the optimizations turned off is the decreased containment (Section 3.3.2) of ROUTE REQUESTs and the longer paths that ROUTE REPLYs and REQUESTs must take.

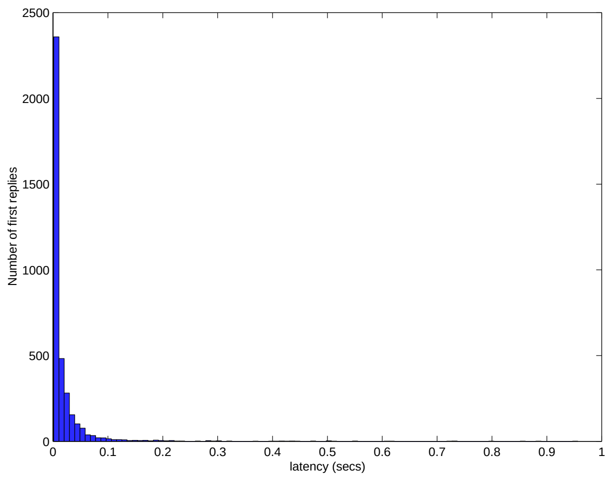


Figure 4.4 Latency of cached ROUTE REPLY packets (No nonpropagating ROUTE REQUESTs, rectangular site, constant motion). Mean is 21.0 ms.

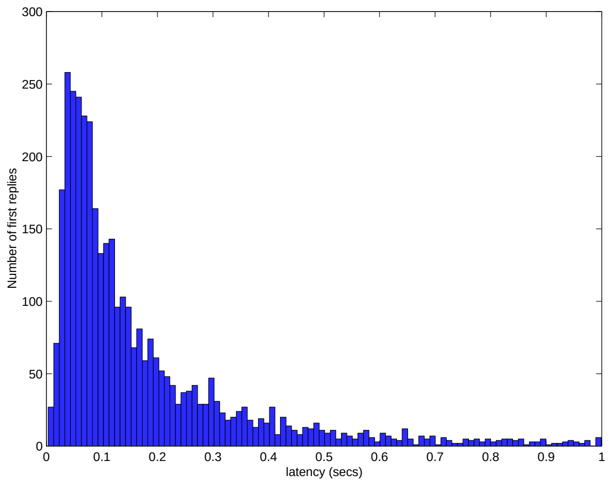


Figure 4.5 Latency of ROUTE REPLY packets (No nonpropagating ROUTE REQUESTs and no cache replies, rectangular site, constant motion). Mean is 403.1 ms.

Table 4.3 Latency of first ROUTE REPLY (No nonpropagating ROUTE REQUESTs and no cache replies, rectangular site, constant motion)

	Mean Latency	Min Latency	Max Latency
Target Replies	403.1ms	3.5ms	42.1s

The Effect of a Different Topology

In the square site, we expect two factors to work together to decrease the latency of Route Discovery as compared to the rectangular site. First, the mean path length is less, so forwarding latency related to path length should be less. Second, the first ROUTE REPLY received should come from nodes closer to the initiator of the Route Discovery, because routing information is better distributed throughout the network. Routing information will be distributed more widely in the square site than the rectangular one because the average degree of each node (the number of direct, 1-hop neighbors of each node) is higher, which results in each node overhearing more of the traffic sent by other nodes.

The average degree of a node in the square site is 16.5 as compared to 11.5 in the rectangular site. As shown in Table 4.4, the mean first ROUTE REPLY latency for all three types of REPLYs decreases relative to the latencies in the rectangular site (Table 4.1), supporting the hypothesis that there are more nodes "close" to the originator of a Route Discovery and that routing information is more spread out through the network.

The node degrees observed in these experiments are within the range of values seen in simulations of a widely deployed multi-hop wireless access network, and are probably typical of those that will be seen any widely deployed network.

4.2.3. Summary of Latency Results

The individual per-hop forwarding latency of all packets is dictated by the length of the interface queue and by the media access time, since the packets at each node must wait their turn for transmission. The latency of a single ROUTE REQUEST propagating outwards or a ROUTE REPLY propagating inwards is dominated by effects due to path length: the greater the number of hops over which the ROUTE REQUEST or ROUTE REPLY packet must travel, the longer the time the packet takes to reach its destination. However, *the greatest influence on the latency of a Route Discovery is the distribution of information in nodes' caches,* since it is the path length to the closest node that can generate a ROUTE REPLY that determines the latency of Route Discovery.

The latency of the slowest Route Discoveries seen during an experiment are dominated by the effects of congestion. The maximum latencies reported in this section are in the tens of seconds — we have verified in each case that the REQUEST or REPLY was held up in a series of long interface queues as the packet

Table 4.4 Latency of first ROUTE REPLY by type (All-Opt DSR, square site, constant motion)

	Mean Latency	Min Latency	Max Latency
Neighbor Replies	4.7ms	1.3ms	117.1ms
Cache Replies	29.1ms	1.3ms	1.3s
Target Replies	36.6ms	29.4ms	54.4ms

propagated, each queue draining very slowly due to media contention around the node. Given the long latencies that some Route Discoveries will experience, protocol designers might want to include even the routing protocol packets in a QoS scheme that allows a packet's originator to specify that the packet should be dropped after some time period. Route Discovery packets could then be set to expire shortly after the originator plans to retransmit the ROUTE REQUEST.

4.3. The Cost of On-Demand Route Discovery

Although on-demand routing protocols can reduce routing overhead by not disseminating routing information throughout the network on a periodic basis, the actual cost of performing on-demand Route Discovery can be significant. When a node **A** transmits a ROUTE REQUEST, the request flood-fills through the network, potentially disturbing each node in the network and consuming valuable bandwidth and battery power. Each node that receives the request must either transmit a ROUTE REPLY based on information in its route cache or forward the request further.

Using the containment and discovery cost metrics defined in Section 3.3.2, we examine the overall cost of Route Discovery in DSR. We focus on how the use of route caches and nonpropagating requests effect Route Discovery among nodes at the rectangular site, and then compare those results with data collected at the square site.

Our intuition of how Route Discovery works predicts that both the containment and discovery cost metrics of Route Discovery should be sensitive to the average *degree* of the nodes in the network. The degree of a node is the number of direct neighbors the node has, and measures how tightly interconnected the network is. As the degree of interconnectivity goes up, it is harder to contain a ROUTE REQUEST to one part of the network. In addition, the "branching factor" of a propagating ROUTE REQUEST increases which causes more nodes to receive and process it. Thus, we expect containment to decrease and discovery cost to increase in environments where the average node degree increases.

4.3.1. Overall Route Discovery Cost in DSR With All Optimizations

We began by studying the behavior of Route Discovery in scenarios using the rectangular site, where the average node degree was measured to be 11.5 neighbors. Route Discovery behaved as described in Sections 2 and 4.2, with All-Opt DSR sending a nonpropagating ROUTE REQUEST before transmitting a propagating ROUTE REQUEST if the nonpropagating request failed to provide a route within 30 ms. Table 4.5 summarizes the discovery costs of propagating and nonpropagating requests broken out by the number of

Table 4.5 Summary of Route Discovery costs
(All-Opt DSR, rectangular site, constant motion)

	Nonpropagating	Propagating	Total
Request Og	957	316	1,273
Request Fw	0	6,115	6,115
Reply Og	3,912	3,215	7,127
Reply Fw	0	7,002	7,002
Containment	77.0%	41.0%	68.0%
Cost	5.09	52.69	16.90

times a routing packet of the given type was originated (Og) or forwarded (Fw). When propagating and nonpropagating requests are used together in All-Opt DSR, the total containment for all requests is 68%, and the average discovery cost metric is nearly 17. The two types of ROUTE REQUESTs have very different behavior and costs, however, and so it is informative to analyze them separately.

An average of 316 propagating requests were initiated during each of the simulations. Examining only these requests, on average each request was forwarded 20 times and caused 10 ROUTE REPLY packets to be returned, yielding a containment metric of 41%. This means that most propagating requests involve a little more than half of the nodes in the network. The average discovery cost of a propagating request was nearly 53 transmissions. The fact that the origination of a single propagating ROUTE REQUEST results in the transmission of more than 50 DSR packets, even when the containment metric is 40%, indicates that Route Discovery has the potential to be a very expensive operation when not well contained.

In comparison to the propagating ROUTE REQUESTs, the nonpropagating requests have a containment metric of 77% and the average discovery cost drops to only 5 transmissions. A propagating request costs nearly 10 times more than a nonpropagating request because the total cost of a nonpropagating request is the one transmission of the ROUTE REQUEST plus one transmission for each of the neighbors that generates a ROUTE REPLY.

4.3.2. Effects of Replying from Cache

As described in Section 2.5.1, DSR contains Route Discoveries by allowing a node to short-circuit the outward propagation of a ROUTE REQUEST packet when it has a route to the request's target in its route cache. In order to discern the effect of replying from cache on the cost of Route Discovery, we re-ran all the scenarios with a modified version of DSR where replying from cache was disabled.

As shown in Table 4.6, the overall containment metric decreases to 10% when replying from cache is turned off. The cost of each discovery likewise increases by more than a factor of five to 102 packet transmissions per Route Discovery. The dramatic decrease in containment and increase in overhead argue strongly that allowing nodes to reply from their route cache is vital to limiting the cost of discovery. Even though the discovery cost quintupled, the packet delivery ratio of DSR with cache replies disabled was the same as in All-Opt DSR. While this is an encouraging result for on-demand Route Discovery, this may not be significant, as a different offered traffic load or communication pattern could suffer more from the dramatic increase in overhead.

Table 4.6 Summary of Route Discovery costs (No reply from cache and no nonpropagating ROUTE REQUESTs, rectangular site, constant motion)

	Propagating
Request Og	871
Request Fw	39,471
Reply Og	8,413
Reply Fw	39,554
Containment	10%
Cost	102.52

Table 4.7 Summary of Route Discovery costs (no nonpropagating ROUTE REQUESTs, rectangular site, constant motion)

	Propagating
Request Og	776
Request Fw	8,812
Reply Og	8,077
Reply Fw	8,894
Containment	58 %
Cost	34.19

4.3.3. Effects of Nonpropagating ROUTE REQUESTS

A second mechanism that may be used to reduce the cost of Route Discovery is an expanding ring search. Specifically, we consider the algorithm used by All-Opt DSR where each propagating ROUTE REQUEST is preceded by a nonpropagating ROUTE REQUEST, i.e., with a maximum propagation limit (TTL) of 1. To evaluate whether the nonpropagating ROUTE REQUEST accomplishes its intended purpose of allowing the initiator of the Route Discovery to quickly and inexpensively query the route caches of each of its neighbors, we experimented with a version of DSR in which the sending of nonpropagating requests has been disabled.

When DSR is modified to send only propagating ROUTE REQUESTs (Table 4.7), the overall containment metric decreases from 68% to 58%, and the cost of a single Route Discovery doubles from 16 to 34 packets. These data argue strongly in favor of a two-phase Route Discovery that uses both nonpropagating and propagating requests, even though the tradeoff is an increase in latency when the nonpropagating request fails (Section 4.2).

4.3.4. Effects of Node Degree on Route Discovery

As previously mentioned, nodes in the rectangular site have an average node degree of 11.5. To evaluate the effect of node degree on Route Discovery, we performed the same experiments using the square site where nodes have an average node degree of 16.5. This increase in the number of neighbors stems from the fact that the rectangular site is only 300m wide and hence, much of the area covered by a node's transmission range lies outside the space where nodes are allowed to move to.

Our intuitive model of Route Discovery predicted that changing to the square site where nodes have greater degree should both decrease containment and increase discovery cost. As Table 4.8 shows, however, containment does decrease by 6% from 68% to 62%, but the discovery cost remains relatively *unchanged*. Analyzing the nonpropagating and propagating requests separately makes the reasons for the discrepancy more apparent.

The reason the overall metrics do not follow our expectations is that the increased node degrees in the square site not only increase the number of nodes that overhear each ROUTE REQUEST packet (thus increasing the cost of the request), but also increase the number of nodes that overhear source routes used by their neighbors. This causes routing information about each destination to be spread more widely throughout the network than at the rectangular site. The greater spatial distribution of routing information allows nonpropagating requests to succeed more often in the square site, and prevents a greater number

Table 4.8 Summary of Route Discovery costs
(All-Opt DSR, square site, constant node motion)

	Nonpropagating	Propagating	Total
Request Og	217	27	244
Request Fw	0	798	798
Reply Og	1,931	387	2,318
Reply Fw	0	584	584
Containment	67.0%	18.0%	62.0%
Cost	9.87	66.05	16.11

of the expensive propagating route requests. In the rectangular site, 24% of the ROUTE REQUESTs were propagating requests, while in the square site, only 12% of the Route Discoveries required a propagating ROUTE REQUEST. This difference makes the cost and containment metrics of nonpropagating requests dominate the overall overhead for the square site, thereby giving the appearance that the discovery cost and containment metrics are roughly equivalent on the two sites. However, Route Discovery tends to be more expensive and less well-contained in the square site.

4.4. On-Demand Route Repair

Here we consider the amount of time it takes a sender to "recover" when a route that it is using breaks. As illustrated in Figure 4.6, the *latency of a route repair* is defined as as the length of time between when the first packet is lost while attempting to forward across the broken link and when a packet with a higher sequence number is successfully received at the destination.

Table 4.9 shows the overall effect of a link breakage on an active connection in terms of the time to repair the breakage, and the number of packets lost. For 60 percent of the breakages, the salvaging optimization (Section 2.5.2) successfully prevents any packets from being lost. When packets are lost, most loss bursts

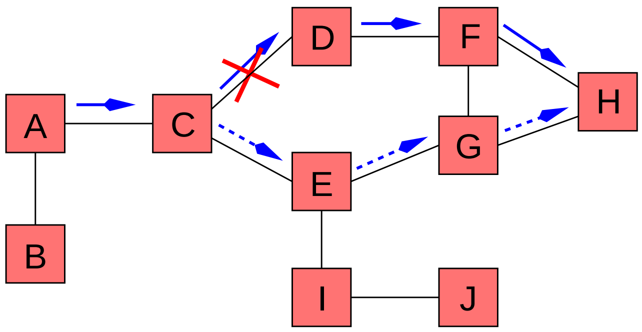


Figure 4.6 Latency of route repair is measured as the length of time between when the first packet is lost while attempting to forward across the broken link and the time the next packet is successfully received at the destination.

Table 4.9 Overall effect of the breakage of an active link on a connection.

	Quantiles			
	25%	50%	75%	99%
repair latency	26ms	106ms	403ms	11.8s
missing packets	0	0	2	51

are fairly small. The next two tables independently examine the cases where salvaging is successful and where loss occurs.

4.4.1. Repair Latency When Salvaging is Successful

In the data set used for this analysis, there were 10,179 link breakages affecting actively used routes. For 6,015 of these breakages, an alternate route to the destination was available in the Route Cache of the forwarding node that discovered the breakage, and this route successfully delivered the packet to the final destination without any packet losses. Table 4.10 shows the latency of route repair in these cases where salvaging was able to prevent any packet loss. For these cases, the latency of repair is just the time for the packet to traverse the hops on the salvage source route. The distribution has a long tail due to the congestion effects commonly present around the sources of broken links, as noted in Section 4.4.3.

4.4.2. Repair Latency When Packets Are Lost

Table 4.11 shows the latency and packet loss for route repairs for the remaining 4,000 cases in which at least one packet was lost. The table shows that loss bursts are small, with more then 50 percent of the loss bursts having 2 packets or fewer. In many cases, the large bursts at the tail of the distribution represent network partitions or periods of extreme congestion.

The repair latency numbers in Table 4.11 must be read with an understanding that, due to an artifact of the traffic sources used, the minimum latency for route repair when a packet is lost is nearly 250 ms. The CBR traffic sources only offer a packet to the network every 250 ms, and the repair latency timer is not started until a packet actually fails to traverse the broken link. The result is that, when a packet is lost, it will be nearly 250 ms until the network next has a chance to successfully deliver a packet and stop the repair latency timer. Values of repair latency less than 250 ms reflect the time it takes packets to traverse from the originator across a potentially congested network to the broken link where they are lost (an example is illustrated by the timeline in Figure 4.7). Less frequently, repair latencies less than 250 ms are due to temporary partitions or congestion that have caused a burst of packets, all with the same route, to be queued in the network, with the first packet being lost but following packets being salvaged successfully.

Table 4.10 The latency of route repair for cases where salvaging was able to prevent all packet loss.

	Quantiles			
	25%	50%	75%	99%
repair latency	17ms	36ms	107ms	1.7s

Table 4.11 Latency and packet loss for route repairs where at least one packet was lost.

	Quantiles			
	25%	50%	75%	99%
repair latency	185ms	429ms	1.19s	11.3s
missing packets	1	2	6	50

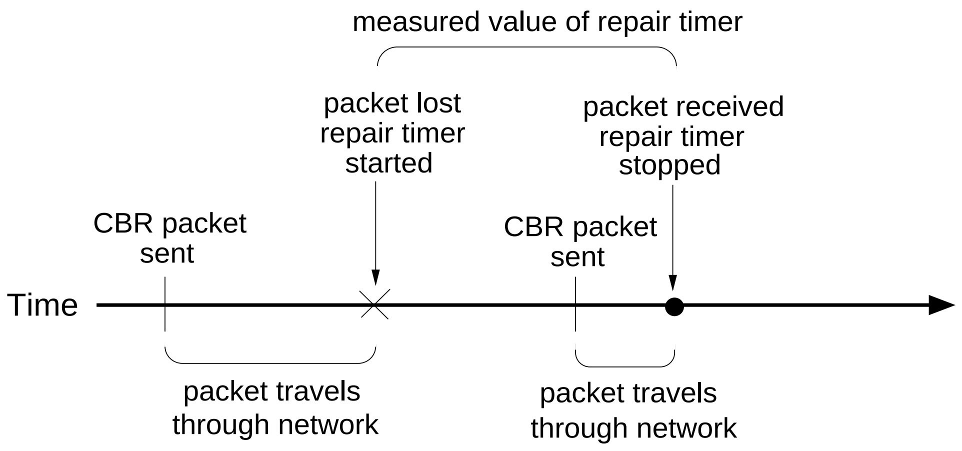

measured value of repair timer
packet lost repair timer started
packet received repair timer stopped
CBR packet sent
CBR packet sent
Time
packet travels through network
packet travels through network

Figure 4.7 Illustration of how the repair latency timer can take on values less than the interval between CBR packets.

4.4.3. Latency of ROUTE ERRORS

As described in Chapter 2, a node originating packets will detect that a route it is using has broken only after it tries to send a packet along the broken route, and receives a ROUTE ERROR in response. The ROUTE ERROR is sent by the node that, upon attempting to forward the packet over a link in the source route, decides that the link is broken. The forwarding node declares the link broken after making several attempts to transmit the packet to the next hop, and failing to receive a passive or explicit acknowledgment of success.

For All-OptDSR in the rectangular site, Table 4.12 shows the time taken for a ROUTE ERROR to propagate from the node that detected a link breakage to the originator of the packet that contained the broken source route. The median latency per hop is 8.6 ms, while the mean is 26.6 ms. The difference between the two is due to the heavy tail caused by congestion. The average link breakage in this environment occurred 1.8 hops from the packet's originator.

The propagation time for ROUTE ERRORs is significantly longer than for other types of packets. This is due to the fact that the reason a ROUTE ERROR is being sent is that something is wrong in the network. Either topology change, congestion, or external interference is preventing the exchange of packets and causing what had been working links to appear as broken. It should be no surprise that it takes longer than usual to successfully get a packet out of such a trouble spot! However, it does serve as a cautionary tale against basing a reaction to mitigate a problem on the ability to receive an explicit notice of the problem from a node inside the problem region. A similar effect has been noted in wired networks using Explicit Congestion Notification to retard packet sources when parts of the network become congested.

4.5. Cache Consistency in On-Demand Protocols

All routing protocols which use on-demand Route Discovery must include some kind of route caching system, since the originator of a packet cannot afford the cost of doing a Route Discovery operation for every packet it wishes to send. Once the originator discovers a route through the network, it must remember the route in some kind of cache for use in sending future packets. DSR, in particular, makes even greater use of the route cache, using it not only to cache routes for the purpose of originating packets, but also for the purpose of allowing nodes to answer ROUTE REQUESTs targeted at other nodes, as explained in Section 2.5.1.

When a cache is added to the system, however, the issue arises of how stale cache data is handled. In the context of a route cache, we call a route stale if any link in the route is broken, since a packet sent using the route will encounter a forwarding error when it attempts to traverse the broken link. Removing stale data from the cache of a node originating packets is critical, since any packet the node sends with a stale route will result in a ROUTE ERROR being returned, with a reasonable chance of the packet being dropped.

Allowing nodes to use the data in their route caches to issue ROUTE REPLY packets carries even greater risk, as nodes with stale cache data may return stale, incorrect routes that will pollute both that particular Route Discovery and potentially other nodes' caches.

Table 4.12 Latency of ROUTE ERRORS (All-OptDSR, rectangular site, constant motion)

	number of hops ROUTE ERROR traveled							
	1	2	3	4	5	6	7	8
median (ms)	8.75	17.37	24.64	29.21	37.75	80.82	58.13	336.1
# ERRORS	8602	3985	1978	929	367	123	18	4

For the purposes of analyzing cache behavior, we label the nodes as either *originator nodes* or *forwarder nodes*. Originator nodes are those that are the sources of packets in a connection, while forwarder nodes do not themselves originate data packets, but only act as routers to carry others' communications. As described in Section 4.1, there are 14 originator nodes (which may also serve as forwarders for other connections) and 36 forwarder nodes (which are not the originator of any connections) in the communication pattern used in our simulations. The route cache of an originator node behaves very differently from that of a forwarder node, since the originator actively invokes Route Discovery to maintain good routes to the nodes to which it sends packets, and it is the target of ROUTE ERRORS whenever it sends a packet with a broken route. Forwarder nodes must learn and correct routes opportunistically.

4.5.1. Contents of the Route Cache

The most basic measure of how well a node's route cache performs is the percentage of cache lookups that actually find a *good* cached route in the node's cache. This measure is a function of the percentage of good links in a node's cache and the overall cache hit rate. Figure 4.8 shows the average number of links in each node's cache, broken out into number of good links and bad links, for our constant node motion scenarios. On average, 84% of the links in caches were found to be good while sampling once per second. As expected, all of the bad links in the caches are due to node motion, and there are 0% bad links in the caches of nodes in the scenarios with no node motion. Originating nodes have a slightly greater number of links in their caches, due to the fact that they receive ROUTE REPLY packets from nodes throughout the network when they perform Route Discovery, whereas forwarding nodes only hear ROUTE REPLYs that travel past them on the way to an originating node. That the difference is slight argues that Route Discovery is successfully priming the caches of the forwarding nodes.

To determine if the routes in the nodes' caches are useful, we evaluated the cache hit rate for each node in our simulations and show this data in Figure 4.9. As described in Section 2.5.1, nodes use their cache

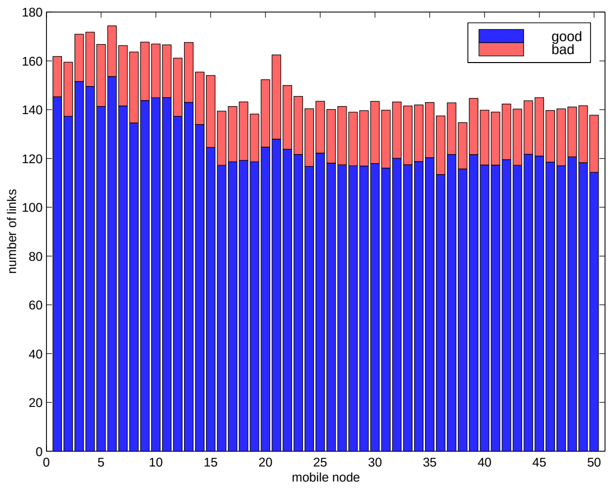


Figure 4.8 Average number of links in each node's cache over simulation runs for All-Opt DSR (rectangular site, constant motion). Nodes 1 through 14 are originating nodes.

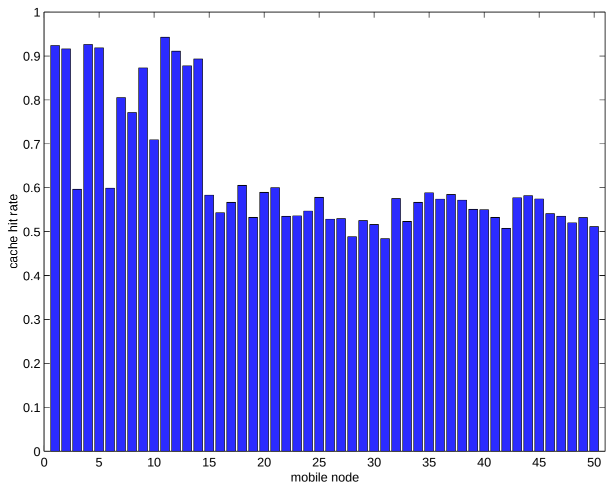


Figure 4.9 Cache hit rate for All-Opt DSR (rectangular site, constant motion). Nodes 1 through 14 are originating nodes.

when originating packets, when deciding whether to return a cache reply in response to a received ROUTE REQUEST, and when salvaging. Originating nodes have a significantly higher hit rate than forwarding nodes, since the vast majority of the cache lookups that originating nodes perform are for packets to destinations for which they have performed Route Discovery. The hit rate of forwarding nodes is sensitive to both the geometry of the space and the communication pattern, since a node's hit rate will go up if the ROUTE REQUESTs that it overhears are for destinations in close spatial proximity, and down as it processes ROUTE REQUESTs for nodes far away from it. Forwarding nodes in the rectangular site have an average hit rate of only 55%, which, though low, is consistent with the philosophy of on-demand Route Discovery: there is no reason to have a route to a destination unless you are originating packets to it. In the square site, the average hit rate for forwarding nodes increases to 62%, since nodes are fewer hops apart and there is greater shared knowledge as nodes overhear a greater number of useful source routes that they can then add to their route caches (Section 2.5.3).

4.5.2. The Quality of Route Replies

Nodes in the network use their caches to generate ROUTE REPLY packets in response to other nodes' Route Discoveries, in order to limit the propagation of ROUTE REQUESTs (Section 2.5.3). Given that there may be some stale data in nodes' caches, the question arises as to the extent of the cache pollution that the Route Discovery process might cause as bad replies from cache are returned to the initiator of a Route Discovery and overheard by forwarding nodes. We collected statistics for this over our scenarios using All-Opt DSR and report these results in Table 4.13. Using the nominal 250 m transmission range of the radios modeled in our simulations to determine if the links are good, 40% of the ROUTE REPLYs received by the initiator of a Route Discovery contain routes that would not work if used.

That 40% of ROUTE REPLYs contain broken routes is not surprising considering how cached information is learned, and why a Route Discovery is initiated. Route Discovery is performed when a node wants to send a packet to a destination to which it does not have any routes. This occurs because either the node has never had a route to the destination, or because its last working route to the destination just broke, many times as the result of some major network topology change that altered the available paths to reach the destination. Since many nodes will have cached routes to that destination with links in common, the breakage of any of those links will cause many of the cached replies generated for that destination to be bad. In our simulations of All-OptDSR, the overwhelming majority of ROUTE REPLY packets are based on cached data, and only 59% of those replies carry correct routes. Though 84% of the links in the nodes' caches are good, the probability of a route being good given that it has been retrieved from a forwarding node's cache in our simulations is only 56%. Even replies from the target itself are not 100% correct, since routes can change while the REPLY propagates back to the requester.

However, the fact that many nodes share the same bad information allows these bad links to be quickly removed from their caches by Route Maintenance, especially if the Gratuitous ROUTE ERRORS (Section 2.5.2) optimization is being used. The rapid removal, in turn, allows DSR to maintain its high packet delivery ratio.

If replying from cache is disabled and only propagating ROUTE REQUESTs are sent, a total of 8,413 ROUTE REPLY messages are received by initiators, of which 93% are good. While disabling the replying from cache mechanism significantly increases the number of working routes an initiator receives, it also increases overhead and latency as described in Sections 4.2 and 4.3.

The fact that DSR continues to have a very high packet delivery ratio (Figure 5.4) even when 40% of the routes returned by Route Discovery are incorrect, argues that Route Maintenance is performing well so that the discovery and dissemination of bad routes does not result in the continued use of bad routes.

Table 4.13 The correctness of source routes returned by Route Discovery.

	# Replies	% Good
From target	119	95.2 %
From cache	6809	59.4%
Total	6928	60.0%

4.5.3. Correlation between Age and Quality of Routes

A standard strategy in caching is to assume that the data learned most recently is more likely to be correct and useful than data that has been in the cache for a while — this *temporal locality* assumption is the basis behind the common LRU cache replacement policy. DSR deliberately does not include such a temporal policy because of its philosophy that the on-demand nature of Route Maintenance allows it to store routes until they are found to be bad, at which time they are cheaply erased.

We define the *age* of a link in the cache to be the elapsed time since it was last entered into the cache. If the link already appears in the cache when it is entered, its age is reset to zero. The age of a link in a node's cache can be interpreted as indicating the length of time since the node had some indication the link was at that time still valid because it saw the link being used by another node The age of a cached link is not reset when a link is retrieved from the cache, since retrieval does not suggest that the link is valid.

To determine whether link age is a useful predictor of the probability that a link is good, we conducted an experiment where an omniscient evaluation system inside the simulator examined all the links in each node's cache once per second of simulation time. Each cached link was classified as good or bad, depending on whether the two endpoints of the link where actually within communication range of each other when the evaluation was performed. Figure 4.10 shows the average age of the good links and the bad links in the nodes caches at each of the 900 seconds in the simulation. The scenarios were generated by the random waypoint movement model and used a uniform node-to-node communication pattern. Since the routing

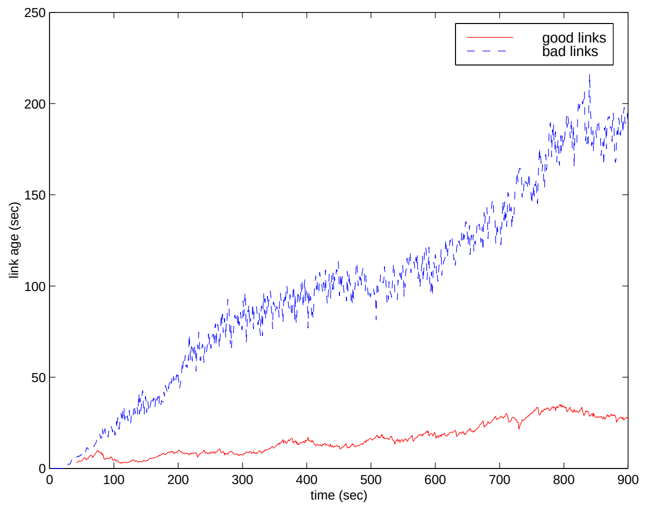


Figure 4.10 The average age of the good and bad links as a function of time in the route caches (rectangular site, constant motion)

protocol was not involved in determining whether the links were good or bad, the figure includes bad links in the nodes' caches that the on-demand Route Maintenance might never encounter (since those routes might never be retrieved from the cache).

With the communication and movement patterns used in our scenarios, there is a clear distinction between the age of good and bad links. As the simulation progress, the average age of a bad link in the cache grows linearly, while the average age of a good link stays below 45 seconds. On average, every link older than 50 seconds was bad. Although we have not collected enough data to confirm it, we suspect this occurs because movement is uniformly distributed, and it takes 50 seconds for a node moving at the simulation's average node speed of 10m/s (half the maximum of 20m/s) to cross the nominal diameter of another node's transmission range (500m).

The difference in age between good and bad links suggests that cache entries should be expired if older than some threshold, and Figure 4.10 shows how trivial the discrimination function could be. In general, however, simple passage of time is not what makes a link go bad — link goodness or badness should correlate directly with the amount of relative node motion, since that is what breaks links. In the random waypoint movement model we use in our scenarios, the distance two nodes have moved apart correlates closely with time, which contributes to the correlation between link age and quality. However, we believe that in other movement and communication patterns with a higher degree of *spatial locality*, in which nodes are more likely to communicate with nodes near them than farther away, or are more likely to move together with the other nodes with which they are communicating, a much lower correlation between link age and link badness would exist.

In summary, the data here presented show that for particular environments or applications in which the probability of link breakage is known to increase with the passage of time, the introducing an LRU expiration policy to the Route Cache would improve the performance of DSR. It is an open question for future longitudinal research to determine whether the environments in which ad hoc networks are actually used exhibit more spatial locality and correlated motion or, inversely, uncorrelated motion causing a correlation between the probability of link failure and the passage of time.

4.6. Chapter Summary

This chapter presents a detailed examination of the fine-grain performance of on-demand routing protocols, and it shows how on-demand mechanisms are able to meet the four challenges presented at the beginning of the chapter, using DSR as an example to study. DSR provides a good source of examples for this study, since it is based entirely on on-demand behavior, and thus the results are not affected by any periodic or background activity within the protocol.

To summarize DSR's response to the four challenges:

- The latency of Route Discovery is reasonable, and it follows the expected trends. Requests for source routes to nodes that have not been previously heard from see a latency proportional the distance between the requester and requestee. However, using the Route Caches to answer ROUTE REQUESTS is very effective at breaking this linear dependence, and Route Discoveries see an average latency of 40ms when replying from cache is enabled. The variation in Route Discovery latency is primarily due to congestion, some which may have been induced by the Route Discovery itself. Future work should be able to analyse the duration, level and effect of congestion caused by Route Discovery.

- The cost of Route Discovery can be controlled by localization. This chapter studied two forms of localization: the use network knowledge in the form of Route Caches, and the use of fiat in the form of expanding ring search. The results show that both forms of localization are effective individually. When used together, they reduce the mean latency of Route Discovery in our 50 node network from 403 ms to well under 40 ms and decrease the total cost from 102 packets per discovery to 17 packets, while increasing containment from 10 percent to 68 percent.

- Route repair does not typically result in long strings of packet losses. The salvaging optimization prevented any loss for 60 percent of link breakages, and when packet loss did occur the loss bursts were often small. More than 75 percent of route breakages resulted in two or fewer packets being lost.

- The Route Cache is able to acquire useful information about the overall network topology solely by extracting routing information from packets that pass through and near it. Nodes were able to learn about sufficiently many destinations to achieve a cache hit rate of 55%, whether the nodes were initiating Route Discovery themselves or not. Efficient Route Maintenance is critical in all systems with route caches, as we found that 16% of the links in the nodes' caches were stale, and up to 41% of the ROUTE REPLIES sent based on cached data contained broken routes. Mechanisms like those used in DSR can meet the challenge however, as more than 90% of the data packets were successfully delivered even at constant node motion. This demonstrates that on-demand Route Maintenance is very effective at detecting and cleaning up stale routing information once a node attempts to use it.

Many of the techniques and lessons learned from this work can be applied to the other on-demand routing protocols, such as AODV, TORA, and ZRP. In particular, by adding a mechanism to share routing information among nodes in the network (e.g., having a subset of the data packets each record the route it takes through the network), protocols such as these three could make greater use of their Route Caches to control the overhead cost of Route Discovery. (In the case of TORA and AODV, these Route Caches are distributed in the form of routing tables at each node, by abstractly they serve the same function as a DSR Route Cache.)

A more general conclusion of this chapter is that there is significant value in dissecting a routing protocol into its constituent parts to determine how the parts interact to produce the overall performance of the protocol. We believe this is the first work to examine this type of fine-grain protocol performance in detail using a realistic simulation, and we encourage other ad hoc network routing protocol designers to subject their protocols to the same type of dissection.

Chapter 5

Comparative Analysis of DSR's Performance

This chapter describes a study that provides a realistic, quantitative analysis comparing the performance of a variety of multi-hop wireless ad hoc network routing protocols. We present results of detailed simulations showing the relative performance of four proposed routing protocols for ad hoc networks: DSDV [83], TORA [79, 81], DSR [49, 51, 16], and AODV [85, 84].

These four protocols were chosen for study because they cover a wide range of design choices:

- The use of periodic advertisements (TORA, DSDV) versus on-demand route discovery (DSR, AODV).
- The use of feedback from the MAC layer to indicate a failure to forward a packet to the next hop (DSR, AODV) versus periodic advertisement (DSDV) versus sophisticated neighbor discovery (TORA).
- The use of fundamentally different routing algorithms, such as distributed Bellman-Ford (DSDV, AODV) versus localized link-reversal relaxation (TORA) versus a pruned breadth-first search of connectivity links (DSR).
- The use of fundamentally different forwarding algorithms, such as hop-by-hop routing (DSDV, TORA, AODV) versus source routing (DSR).

5.1. Protocols to be Compared

In this section, we briefly describe the key features of the DSDV, TORA, DSR, and AODV protocols studied in our simulations. We also describe the particular parameters that we chose when implementing each protocol.

The protocols were carefully implemented according to their specifications published as of April 1998 and based on clarifications of some issues from the designers of each protocol and on our own experimentation with them. In particular, during the process of implementing each protocol and analyzing the results from early simulation runs, we discovered some modifications for each protocol that improved its performance. The key improvements to each protocol are highlighted in the respective protocol descriptions below. We also made the following improvements to all of the protocols:

- To prevent synchronization, periodic broadcasts and packets sent in response to the reception of a broadcast packet were jittered using a random delay uniformly distributed between 0 and 10 milliseconds.
- To insure that routing information propagated through the network in a timely fashion, routing packets being sent were queued for transmission at the head of the network interface transmit queue, whereas all other packets (ARP and data) were inserted at the end of the interface transmit queue.
- Each of the protocols used link breakage detection feedback from the 802.11 MAC layer that indicated when a packet could not be forwarded to its next hop, with the exception of DSDV as explained in Section 5.1.1.

5.1.1. Destination Sequenced Distance Vector (DSDV)

DSDV [83] is a hop-by-hop distance vector routing protocol requiring each node to periodically broadcast routing updates. The key advantage of DSDV over traditional distance vector protocols is that it guarantees loop-freedom.

Basic Mechanisms

Each DSDV node maintains a routing table listing the "next hop" for each reachable destination. DSDV tags each route with a sequence number and considers a route R more favorable than R' if R has a greater sequence number, or if the two routes have equal sequence numbers but R has a lower metric. Each node in the network advertises a monotonically increasing even sequence number for itself. When a node **B** decides that its route to a destination **D** has broken, it advertises the route to **D** with an infinite metric and a sequence number one greater than its sequence number for the route that has broken (making an odd sequence number). This causes any node **A** routing packets through **B** to incorporate the infinite-metric route into its routing table until node **A** hears a route to **D** with a higher sequence number.

Implementation Decisions

We did not use link layer breakage detection from the 802.11 MAC protocol in obtaining the DSDV data presented in this chapter, because after implementing the protocol both with and without it, we found the performance significantly worse *with* the link layer breakage detection. The reason is that if a neighbor **N** of a node **A** detects that its link to **A** is broken, it will broadcast a triggered route update containing an infinite metric for **A**. The sequence number in this triggered update will be one greater than the last sequence number broadcast by **A**, and therefore is the highest sequence number existing anywhere in the network for **A**. Each node that hears this update will record an infinite metric for destination **A** and will propagate the information further. This renders node **A** unreachable from all nodes in the network until **A** broadcasts a newer sequence number in a periodic update. **A** will send this update as soon as it learns of the infinite metric being propagated for it, but large numbers of packets can be dropped in the meantime.

Our implementation uses both full and incremental updates as required by the protocol's description. However, the published description of DSDV [83] is ambiguous about specifying when triggered updates should be sent. One interpretation is that the receipt of a new sequence number for a destination should cause a triggered update. We call this approach DSDV-SQ (sequence number). The advantage of this approach is that broken links will be detected and routed around as new sequence numbers propagate around the broken link and create alternate routes. The second interpretation, which we call simply DSDV, is that only the receipt of a new metric should cause a triggered update, and that the receipt of a new sequence number is not sufficiently important to incur the overhead of propagating a triggered update.

We implemented both DSDV-SQ and DSDV and found that while DSDV-SQ is much more expensive in terms of overhead, it provides a much better packet delivery ratio in most cases. The second scheme (DSDV) is much more conservative in terms of routing overhead, but because link breakages are not detected as quickly, more data packets are dropped. All of the results presented in this chapter use DSDV-SQ, with the exception of Section 5.4.2, which compares DSDV-SQ with DSDV.

Table 5.1 lists the constants used in our DSDV-SQ simulation.

5.1.2. Dynamic Source Routing (DSR)

DSR is described in detail in Chapter 2. For this study, we implemented the following optimizations from among those described in Chapter 2.

Table 5.1 Constants used in the DSDV-SQ simulation.

Periodic route update interval	15 s
Periodic updates missed before link declared broken	3
Initial triggered update weighted settling time	6 s
Weighted settling time weighting factor	7/8
Route advertisement aggregation time	1 s
Maximum packets buffered per node per destination	5

Although the DSR protocol supports unidirectional routes, IEEE 802.11 requires an RTS/CTS/Data/ACK exchange for all unicast packets, limiting the routing protocol to using only bidirectional links in delivering data packets. We implemented DSR to discover only routes composed of bidirectional links by requiring that a node return all ROUTE REPLY messages to the requestor by reversing the path over which the ROUTE REQUEST packet came. If the path taken by a ROUTE REQUEST contained unidirectional links, then the corresponding ROUTE REPLY will not reach the requestor, preventing it from learning the unidirectional link route.

In Route Discovery, a node first sends a ROUTE REQUEST with the maximum propagation limit (hop limit) set to zero, prohibiting its neighbors from rebroadcasting it. At the cost of a single broadcast packet, this mechanism allows a node to query the route caches of all its neighbors for a route and also optimizes the case in which the destination node is adjacent to the source. If this nonpropagating search times out, a propagating ROUTE REQUEST is sent.

Nodes operate their network interfaces in *promiscuous* mode, disabling the interface's 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. This optimization allows nodes to learn potentially useful information, while causing no additional overhead on the limited network bandwidth.

Furthermore, when a node overhears a packet not addressed to itself, it checks the unprocessed portion of the source route in the packet's header. If the node's own address is present, it knows that this source route could bypass the unprocessed hops preceding it in the route. The node then sends a gratuitous ROUTE REPLY message to the packet's source, giving it the shorter route without these hops.

Finally, when an intermediate node forwarding a packet discovers that the next hop in the source route is unreachable, it examines its route cache for another route to the destination. If a route exists, the node replaces the broken source route on the packet with the route from its cache and retransmits the packet. If a route does not exist in its cache, the node drops the packet and does not begin a new Route Discovery of its own.

Table 5.2 lists the constants used in our DSR simulation.

Table 5.2 Constants used in the DSR simulation.

Time between retransmitted ROUTE REQUESTS (exponentially backed off)	500 ms
Size of source route header carrying *n* addresses	$4n + 4$ bytes
Timeout for nonpropagating search	30 ms
Time to hold packets awaiting routes	30 s
Max rate for sending gratuitous REPLYS for a route	1/s

5.1.3. Ad Hoc On-Demand Distance Vector (AODV)

AODV [84] is essentially a combination of both DSR and DSDV. It borrows the basic on-demand mechanism of Route Discovery and Route Maintenance from DSR, plus the use of hop-by-hop routing, sequence numbers, and periodic beacons from DSDV.

Basic Mechanisms

When a node **S** needs a route to some destination **D**, it broadcasts a ROUTE REQUEST message to its neighbors, including the last known sequence number for that destination. The ROUTE REQUEST is flooded in a controlled manner through the network until it reaches a node that has a route to the destination. Each node that forwards the ROUTE REQUEST creates a *reverse route* for itself back to node **S**.

When the ROUTE REQUEST reaches a node with a route to **D**, that node generates a ROUTE REPLY that contains the number of hops necessary to reach **D** and the sequence number for **D** most recently seen by the node generating the REPLY. Each node that participates in forwarding this REPLY back toward the originator of the ROUTE REQUEST (node **S**), creates a *forward route* to **D**. The state created in each node along the path from **S** to **D** is hop-by-hop state; that is, each node remembers only the next hop and not the entire route, as would be done in source routing.

In order to maintain routes, AODV normally requires that each node periodically transmit a HELLO message, with a default rate of once per second. Failure to receive three consecutive HELLO messages from a neighbor is taken as an indication that the link to the neighbor in question is down. Alternatively, the AODV specification briefly suggests that a node may use physical layer or link layer methods to detect link breakages to nodes that it considers neighbors [84].

When a link goes down, any upstream node that has recently forwarded packets to a destination using that link is notified via an UNSOLICITED ROUTE REPLY containing an infinite metric for that destination. Upon receipt of such a ROUTE REPLY, a node must acquire a new route to the destination using Route Discovery as described above.

Implementation Decisions

We initially implemented AODV using periodic HELLO messages for link breakage detection as described in the AODV specification [84]. For comparison, we also implemented a version of AODV that we call AODV-LL (link layer), instead using *only* link layer feedback from 802.11 as in DSR, completely eliminating the standard AODV HELLO mechanism. Such an approach saves the overhead of the periodic HELLO messages, but does somewhat change the fundamental nature of the protocol; for example, all link breakage detection in AODV-LL is only on-demand, and thus a broken link cannot be detected until a packet needs to be sent over the link, whereas the periodic HELLO messages in standard AODV allow broken links to be detected before a packet must be forwarded. Nevertheless, we found our alternate version AODV-LL to perform significantly better than standard AODV, and so we report measurements from that version here.

In addition, we also changed our AODV implementation to use a shorter timeout of 6 seconds before retrying a ROUTE REQUEST for which no ROUTE REPLY has been received (RREP_WAIT_TIME). The value given in the AODV specification was 120 seconds, based on the other constants specified there for AODV. However, a ROUTE REPLY can only be returned if each node along the discovered route still has a reverse route along which to return it, saved from when the ROUTE REQUEST was propagated. Since the specified timeout for this reverse route information in each node is only 3 seconds, the original ROUTE REPLY timeout value of 120 seconds unnecessarily limited the protocol's ability to recover from a dropped ROUTE REQUEST or ROUTE REPLY packet.

Table 5.3 lists the constants used in our AODV-LL simulation.

Table 5.3 Constants used in the AODV-LL simulation.

Time for which a route is considered active	300 s
Lifetime on a ROUTE REPLY sent by destination node	600 s
Number of times a ROUTE REQUEST is retried	3
Time before a ROUTE REQUEST is retried	6 s
Time for which the broadcast id for a forwarded ROUTE REQUEST is kept	3 s
Time for which reverse route information for a ROUTE REPLY is kept	3 s
Time before broken link is deleted from routing table	3 s
MAC layer link breakage detection	yes

5.1.4. Temporarily Ordered Routing Algorithm (TORA)

TORA [21, 79, 81] is a distributed routing protocol based on a "link reversal" algorithm [34] that finds and maintains routes via local relaxation of link direction. It is designed to discover routes on demand, provide multiple routes to a destination, establish routes quickly, and minimize communication overhead by localizing algorithmic reaction to topological changes when possible. Route optimality (shortest-path routing) is considered of secondary importance, and longer routes are often used to avoid the overhead of discovering newer routes.

The actions taken by TORA can be described in terms of water flowing downhill towards a destination node through a network of tubes that models the routing state of the real network. The tubes represent links between nodes in the network, the junctions of tubes represent the nodes, and the water in the tubes represents the packets flowing towards the destination. Each node has a height with respect to the destination that is computed by the routing protocol. If a tube between nodes **A** and **B** becomes blocked such that water can no longer flow through it, the height of **A** is set to a height greater than that of any of its remaining neighbors, such that water will now flow back out of **A** (and towards the other nodes that had been routing packets to the destination via **A**).

Basic Mechanisms

At each node in the network, a logically separate copy of TORA is run for each destination. When a node needs a route to a particular destination, it broadcasts a QUERY packet containing the address of the destination for which it requires a route. This packet propagates through the network until it reaches either the destination, or an intermediate node having a route to the destination. The recipient of the QUERY then broadcasts an UPDATE packet listing its height with respect to the destination. As this packet propagates through the network, each node that receives the UPDATE sets its height to a value greater than the height of the neighbor from which the UPDATE was received. This has the effect of creating a series of directed links from the original sender of the QUERY to the node that initially generated the UPDATE.

When a node discovers that a route to a destination is no longer valid, it adjusts its height so that it is a local maximum with respect to its neighbors and transmits an UPDATE packet. If the node has no neighbors of finite height with respect to this destination, then the node instead attempts to discover a new route as described above. When a node detects a network partition, it generates a CLEAR packet that resets routing state and removes invalid routes from the network.

TORA is layered on top of IMEP, the Internet MANET Encapsulation Protocol [20], which is required to provide reliable, in-order delivery of all routing control messages from a node to each of its neighbors, plus notification to the routing protocol whenever a link to one of its neighbors is created or broken. To reduce overhead, IMEP attempts to aggregate many TORA and IMEP control messages (which IMEP refers to as *objects*) together into a single packet (as an *object block*) before transmission. Each block carries a sequence number and a response list of other nodes from which an ACK has not yet been received, and only those nodes ACK the block when receiving it; IMEP retransmits each block with some period, and continues to retransmit it if needed for some maximum total period, after which time, the link to each unacknowledged node is declared down and TORA is notified. IMEP can also provide network layer address resolution, but we did not use this service, as we used ARP [88] with all four routing protocols. For link status sensing and maintaining a list of a node's neighbors, each IMEP node periodically transmits a BEACON (or "BEACON-equivalent") packet, which is answered by each node hearing it with a HELLO (or "HELLO-equivalent") packet.

Implementation Decisions

IMEP must queue objects for some period of time to allow possible aggregation with other objects, but the IMEP specification [20] does not define this time period, and we discovered that the overall performance of the protocol was very sensitive to the choice of this value. After significant experimentation, we chose as the best balance between packet overhead and routing protocol convergence, to aggregate HELLO and ACK packets for a time uniformly chosen between 150 ms and 250 ms, and to not delay TORA routing messages for aggregation. The reason for not delaying these messages is that the TORA link reversal process creates short-lived routing loops that exist from the time that the link-reversal starts until the time that all nodes that need to be aware of the reversal receive the corresponding UPDATE (Section 5.3.2). Delaying the transmission of TORA routing messages for aggregation, coupled with any queuing delay at the network interface, allows these routing loops to last long enough that significant numbers of data packets are dropped.

The TORA and IMEP specifications [81, 20] do not define the precise semantics of reliable object delivery required by TORA, but experimentation showed that very strong semantics must be provided in order to prevent the creation of long-lived routing loops. In particular, all TORA objects must be delivered reliably and in order, without any duplication. Additionally, all neighboring nodes in the ad hoc network must have a consistent picture of the network with regard to each destination. This implies that anytime a node **A** decides its link to a neighbor **B** has gone down, **B** *must* also decide that the link to **A** has gone down.

We have implemented IMEP to provide this functionality, although the retransmission timeout and total number of attempts are not specified by IMEP [20]. We chose a retransmission period of 500 ms and a total timeout of 1500 ms, although based upon our observations, an adaptive retransmission timer should be added to the protocol. In-order delivery is enforced by, at each receiver node **B**, only passing an object block from some node **A** to TORA if the block has the sequence number that IMEP at **B** next expects from **A**. Blocks with lower sequence numbers may generate another ACK but are otherwise dropped. Blocks with higher sequence numbers are queued until the missing blocks arrive or until the maximum 1500 ms total timeout expires, at which point **B** can be certain the object will never be retransmitted. By this point, **A** will have declared its link to **B** down, since it will not have received an ACK for the missing packet. To give the routing protocol at **B** a picture consistent with that seen by the protocol at **A**, the IMEP layer at **B** notifies its routing protocol that the link to **A** is down, then notifies it the link is back up, and then processes the queued packets.

Finally, we improved IMEP's method of link status sensing by reducing it to a point that functions with minimum overhead yet still maintains all of the required link status information. During our experimentation with IMEP, we found that nodes need only send BEACON messages when they are disconnected from all other nodes. Suppose two nodes **A** and **B**, both of which have neighbors, transmit a single HELLO listing each of their neighbors once per BEACON period. If a bi-directional link exists between **A** and **B**, both nodes will overhear each other's HELLOs and all other transmissions, causing each node to create a link to the other with "incoming" status. In subsequent HELLO messages, **A** and **B** will list each other, confirming that a bi-directional link exists between them.

Table 5.4 lists the constants used in our TORA simulation.

5.2. Experimental Design

The overall goal of our experiments was to measure the ability of the routing protocols to react to network topology change while continuing to successfully deliver data packets to their destinations. To measure this ability, our basic methodology was to apply a variety of workloads to a simulated network. In effect, each data packet originated by some sender tests whether the routing protocol can find a route to the destination of that packet at that time. We were not attempting to measure the protocols' performance on a particular workload taken from real life, but rather to measure the protocols' performance under a range of conditions.

We compared the protocols on the basis of simulations run using our extended *ns-2* simulator (Section 3.1). Simulation is the tool best suited for performing this analysis because it allows us to precisely control the environment in which each protocol runs, and to measure the output accurately without perturbing the system. In addition, simulation allows us to independently vary input parameters and measure each protocol's sensitivity to those parameters.

The protocol evaluations are based on the simulation of 50 wireless nodes forming an ad hoc network, moving about over a rectangular (1500m $\times$ 300m) flat space for 900 seconds of simulated time. We chose a rectangular space in order to force the use of longer routes between nodes than would occur in a square space with equal node density. Observations of the behavior of DSR in square spaces are reported in Section 4.2.2 and Section 4.3.4. The physical radio characteristics of each mobile node's network interface, such as the antenna gain, transmit power, and receiver sensitivity, were chosen to approximate the Lucent WaveLAN [107] direct sequence spread spectrum radio.

In order to enable direct, fair comparisons between the protocols, it was critical to challenge the protocols with identical loads and environmental conditions. We pre-generated 210 different scenario files (Section 3.2) with varying movement patterns and traffic loads, and then ran all four routing protocols against each of these scenario files. Since each protocol was challenged in an identical fashion, we can directly compare the performance results of the four protocols. We evaluated the protocols using the Packet Delivery Ratio, Routing Overhead and Path Optimality metrics defined in Section 3.3.

We ran the simulator on a collection of 300 MHz and 333 MHz Intel Pentium II-based PCs under the FreeBSD version of Unix. Each run of the simulator, simulating 900 seconds (15 minutes) of elapsed time, required between 5 and 30 minutes to execute, depending on the protocol and number of traffic sources sending. Approximately 50 million total events were simulated during each run.

Table 5.4 Constants used in the TORA simulation.

BEACON period	1 s
Time after which a link is declared down if no BEACON or HELLO packets were exchanged	3 s
Time after which an object block is retransmitted if no acknowledgment is received	500 ms
Time after which an object block is not retransmitted and the link to the destination is declared down	1500 ms
Min HELLO and ACK aggregation delay	150 ms
Max HELLO and ACK aggregation delay	250 ms

5.2.1. Movement Model

Nodes in the simulation move according to the Random Waypoint model described in Section 3.2.1. We ran our simulations with movement patterns generated for 7 different pause times: 0, 30, 60, 120, 300, 600, and 900 seconds. Each simulation ran for 900 seconds of simulated time, so a pause time of 0 seconds corresponds to continuous motion, and a pause time of 900 (the length of the simulation) corresponds to no motion.

Because the performance of the protocols is very sensitive to movement pattern, we generated scenario files with 70 different movement patterns, 10 for each value of pause time. All four routing protocols were run on the same 70 movement patterns.

We experimented with two different maximum speeds of node movement. This study primarily reports data from simulations using a maximum node speed of 20 meters per second (average speed 10 meters per second), but it also compares this data to simulations using a maximum speed of 1 meter per second.

5.2.2. Communication Model

As the goal of our simulation was to compare the performance of each routing protocol, we chose our traffic sources to be constant bit rate (CBR) sources. When defining the parameters of the communication model, we experimented with sending rates of 1, 4, and 8 packets per second, networks containing 10, 20, and 30 CBR sources, and packet sizes of 64 and 1024 bytes.

Varying the number of CBR sources was approximately equivalent to varying the sending rate. Hence, for these simulations we chose to fix the sending rate at 4 packets per second, and used three different communication patterns corresponding to 10, 20, and 30 sources.

When using 1024-byte packets, we found that congestion, due to lack of spatial diversity, became a problem for all protocols and one or two nodes would drop most of the packets that they received for forwarding. As none of the studied protocols performs load balancing, and the goal of our analysis was to determine if the routing protocols could consistently track changes in topology, we attempted to factor out congestive effects by reducing the packet size to 64 bytes. This smaller packet size still provides a good test of the routing protocols, since we are still testing their ability to determine routes to a destination with the same frequency (a total of 40, 80, or 120 times per second).

All communication patterns were peer-to-peer, and connections were started at times uniformly distributed between 0 and 180 seconds. The three communication patterns (10, 20, and 30 sources), taken in conjunction with the 70 movement patterns, provide a total of 210 different scenario files for each maximum node movement speed (1 m/s and 20 m/s) with which we compared the four routing protocols.

We did not use TCP sources because TCP offers a conforming load to the network, meaning that it changes the times at which it sends packets based on its perception of the network's ability to carry packets. As a result, both the time at which each data packet is originated by its sender and the position of the node when sending the packet would differ between the protocols, preventing a direct comparison between them.

In early studies, we also observed that TCP behavior is dramatically affected by the number of hops packets must take to reach their destinations. Because the chance of a packet being lost to congestion or routing error goes up with the number of hops it must make to reach its destination, we observed that TCP connections to nearby nodes showed dramatically higher throughput than connections made to nodes further away. The high throughput achieved by these "local" connections may have even created congestion around these nodes, further damaging the ability of "far away" connections to achieve good throughput and obscuring the behavior of the routing protocols. In a proposal for future work [70], we outline several methods that can be used to enhance DSR and other on-demand routing protocols to combat this distance penalty.

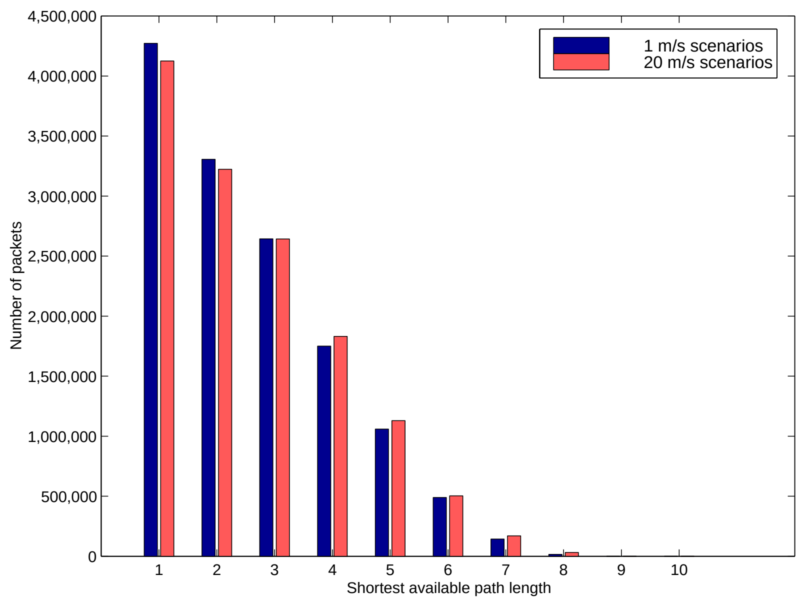


Figure 5.1 Distribution of the shortest path available to each application packet originated over all scenarios.

5.2.3. Scenario Characteristics

To characterize the challenge our scenarios placed on the routing protocols, we measured the lengths of the routes over which the protocols had to deliver packets, and the total number of topology changes in each scenario.

When each data packet is originated, an internal mechanism of our simulator (separate from the routing protocols) calculates the shortest path between the packet's sender and its destination. The packet is labeled with this information, which is compared with the number of hops actually taken by the packet when received by the intended destination. The shortest path is calculated based on a nominal transmission range of 250m for each radio and does not account for congestion or interference that any particular packet might see.

Figure 5.1 shows the distribution of shortest path lengths for all packets over the 210 scenario files at 1 m/s and 20 m/s that we used. The height of each bar represents the number of packets for which the destination was the given distance away when the packet was originated. The average data packet in our simulations had to cross 2.6 hops to reach its destination, and the farthest reachable node to which the routing protocols had to deliver a packet was 8 hops away.

Table 5.5 shows the average number of link connectivity changes that occurred during each of the simulations runs for each value of pause time. We count one link connectivity change whenever a node goes into or out of direct communication range with another node. For the specific scenarios we used, the 30-second pause time scenarios at 1 m/s actually have a slightly higher average rate of link connectivity change than the 0-second pause time scenarios, due to an artifact of the random generation of the scenarios. This artifact is also visible as a slight bump at a pause time of 30 seconds in the performance graphs we present for 1 m/s in Section 5.3.

We confirmed this effect by collecting additional simulation data for pause times of 10, 15, 20, 25, 35, 40, 45, and 50, and generating additional scenario files for 30-second pause time, which did not show the same bump. We present the data for the original 30-second pause time scenarios here, though, so as not to artificially bias the data by manually selecting scenarios.

5.2.4. Validation of the Routing Protocol Simulation Models

To ensure that the implementation of the protocols inside the simulator were faithful to the protocol's specifications, each protocol implementation was studied and verified by at least two members of the Monarch Project, and two independent implementations were made of both AODV and DSDV.

The results of each simulation were internally consistent. That is, the percentage of packets originated by the "application layer" sources that were not logged as either received or dropped at the end of the simulation was less than 0.01% (approximately 10 packets per simulation). These packets were almost certainly in transit at the end of the simulation, as the simulation originates between 40 and 120 packets per simulated second and terminates at exactly 900 seconds without a cool-down period.

The results of our simulations are, in fact, different from the few previously reported studies of some of these protocols. We explain the reasons for these differences in Sections 5.3, 5.4, and 8.2.

5.3. Simulation Results

As noted in Section 5.2.1, we conducted simulations using two different node movement speeds: a maximum speed of 20 m/s (average speed 10 m/s) and a maximum speed of 1 m/s. We first compare the four protocols based on the 20 m/s simulations, and then in Section 5.3.5 present data for 1 m/s for comparison. For all simulations, the communication patterns were peer-to-peer, with each run having either 10, 20, or 30 sources sending 4 packets per second.

5.3.1. Comparison Overview

Figures 5.2 and 5.3 highlight the relative performance of the four routing protocols on our traffic loads of 20 sources.

All of the protocols deliver a greater percentage of the originated data packets when there is little node mobility (i.e., at large pause time), converging to 100% delivery when there is no node motion. DSR and AODV-LL perform particularly well, delivering over 95% of the data packets regardless of mobility rate. In these scenarios, DSDV-SQ fails to converge at pause times less than 300 seconds.

Table 5.5 Average number of link connectivity changes during each 900-second simulation as a function of pause time.

Pause Time	# of Connectivity Changes	
	1 m/s	20 m/s
0	898	11857
30	908	8984
60	792	7738
120	732	5390
300	512	2428
600	245	1270
900	0	0

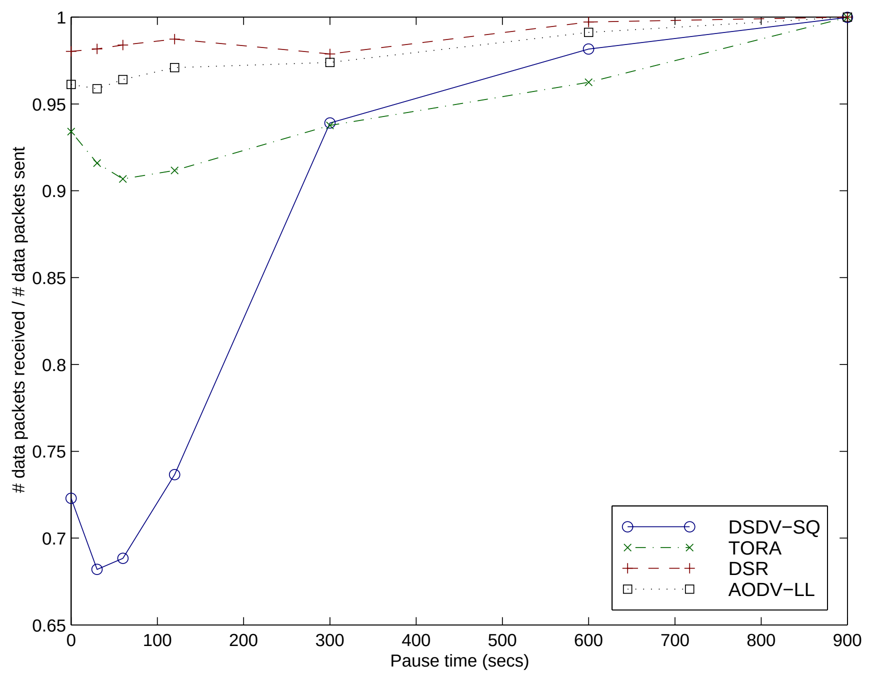


Figure 5.2 Comparison between the four protocols of the fraction of application data packets successfully delivered (packet delivery ratio) as a function of pause time. Pause time 0 represents constant mobility.

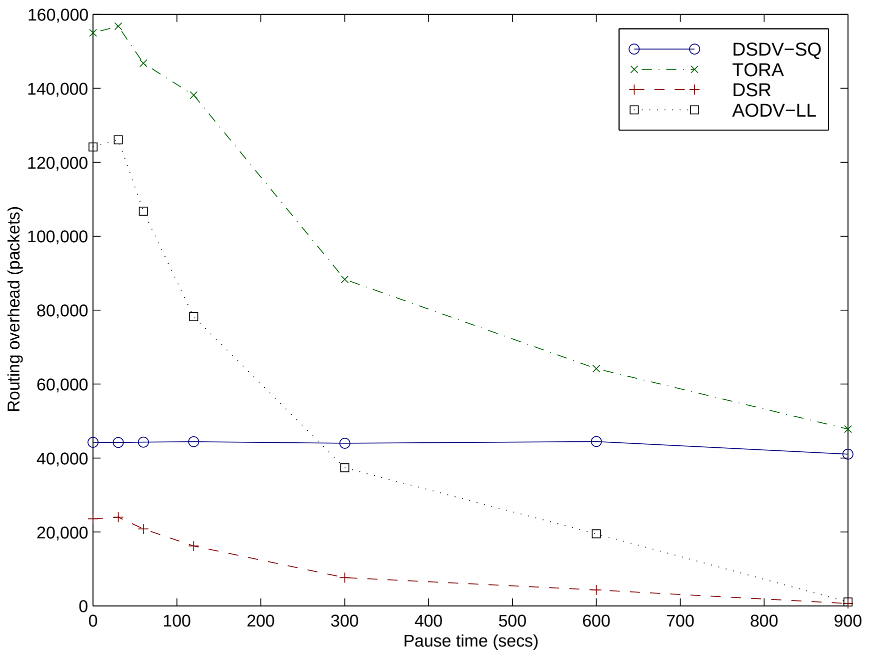


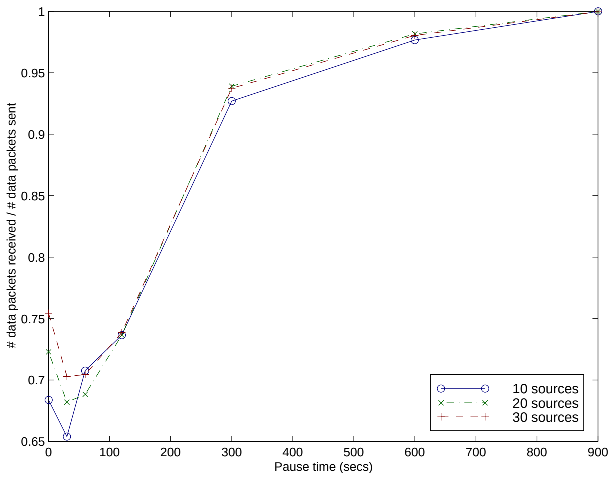
Figure 5.3 Comparison between the four protocols of the number of routing packets sent (routing overhead) as a function of pause time. Pause time 0 represents constant mobility.

The four routing protocols impose vastly different amounts of overhead, as shown in Figure 5.3. Nearly an order of magnitude separates DSR, which has the least overhead, from TORA, which has the most. The basic character of each protocol is demonstrated in the shape of its overhead curve. TORA, DSR, and AODV-LL are all on-demand protocols, and their overhead drops as the mobility rate drops. As DSDV-SQ is a largely periodic routing protocol, its overhead is nearly constant with respect to mobility rate.

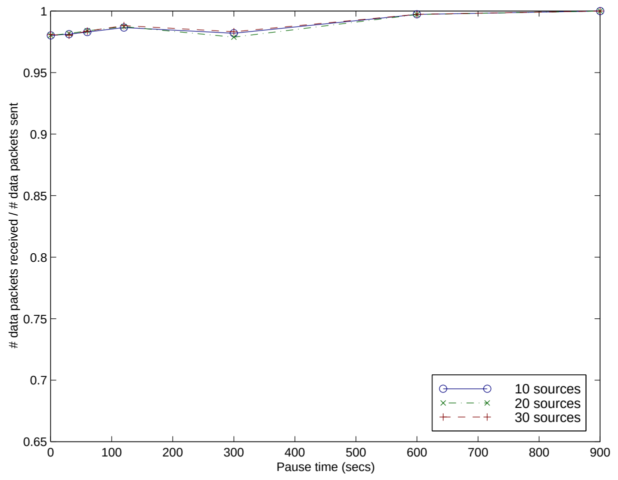
The TORA results shown in Figures 5.2 and 5.3 at pause time 600 are the average of only 9 scenarios, as the overhead for the tenth scenario was much higher than the others due to significant congestion caused by the routing protocol. The complete results are included below and explained in Section 5.3.3.

5.3.2. Packet Delivery Ratio Details

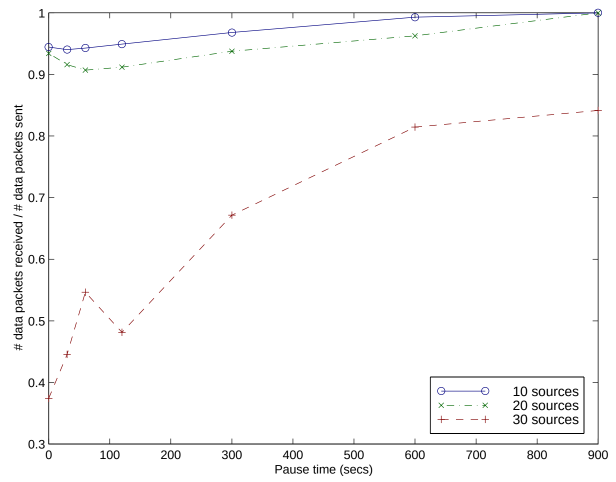
Figure 5.4 shows the fraction of the originated application data packets each protocol was able to deliver, as a function of both node mobility rate (pause time) and network load (number of sources). For DSR



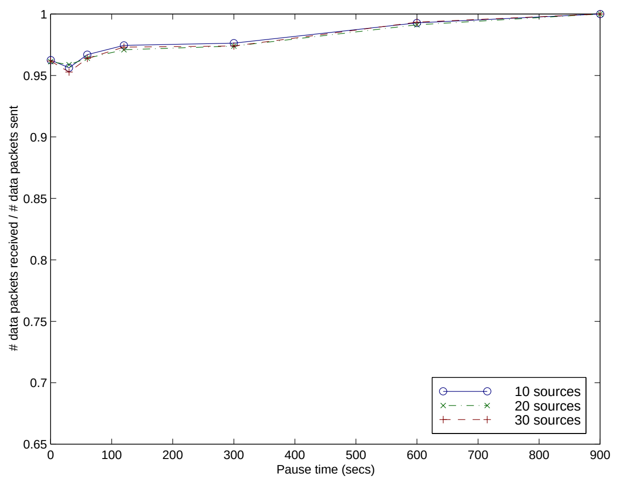
(a) DSDV-SQ



(b) DSR



(c) TORA



(d) AODV-LL

Figure 5.4 Packet delivery ratio as a function of pause time. TORA is shown on a different vertical scale for clarity (see Figure 5.2).

and AODV-LL, packet delivery ratio is independent of offered traffic load, with both protocols delivering between 95% and 100% of the packets in all cases.

DSDV-SQ fails to converge below pause time 300, where it delivers about 92% of its packets. At higher rates of mobility (lower pause times), DSDV-SQ does poorly, dropping to a 70% packet delivery ratio. Nearly all of the dropped packets are lost because a stale routing table entry directed them to be forwarded over a broken link. As described in Section 5.1.1, DSDV-SQ maintains only one route per destination and consequently, each packet that the MAC layer is unable to deliver is dropped since there are no alternate routes.

TORA does well with 10 or 20 sources, delivering between 90% and 95% of originated data packets even at the highest rate of node mobility (pause time 0). The majority of the packet drops are due to the creation of short-lived routing loops that are a natural part of its link-reversal process. Consider a node **A** routing packets to **D** via **B** and **C** as shown in Figure 5.5. If **C**'s link to **D** breaks, **C** will reverse its link to **B**, transmit an UPDATE to notify its neighbors it has done this, and begin routing packets to **D** via **B**. Until **B** receives the UPDATE, data packets to **D** will loop between **B** and **C**. Our implementation of TORA detects when the next-hop of a packet is the same as the previous-hop and drops the data packet, since experiments showed that allowing these packets to loop until their TTL expires or the loop resolves causes more packets to be dropped overall, as the looping data packets interfere with the ability of other nearby nodes to successfully exchange the broadcast UPDATE packet that will resolve their routing loop.

With 30 sources, TORA's average packet delivery ratio drops to 40% at pause time 0, although upon examination of the data we found that variability was extremely large, with packet delivery ratios ranging from 8% to 91%. In most of these scenarios, TORA fails to converge because of increased congestion, as explained in Section 5.3.3. A revision to the IMEP specification [20] released after this study was performed claims to improve the reliable control message delivery semantics provided by IMEP, which might eliminate some of the behaviors seen here. However, these new mechanisms add more packet overhead to TORA/IMEP which, as shown in Section 5.3.3, is already higher than the other protocols studied here.

5.3.3. Routing Overhead Details

Figure 5.6 shows the number of routing protocol packets sent by each protocol in obtaining the delivery ratios shown in Figure 5.4. DSR and DSDV-SQ are plotted on a the same scale as each other, but AODV-LL and TORA are each plotted on different scales to best show the effect of pause time and offered load on overhead. TORA, DSR, and AODV-LL are on-demand routing protocols, so as the number of sources increases, we expect the number of routing packets sent to increase because there are more destinations to which the network must maintain working routes.

DSR and AODV-LL, which use *only* on-demand packets and very similar basic mechanisms, have almost identically shaped curves. Both protocols exhibit the desirable property that the incremental cost of additional sources decreases as sources are added, since the protocol can use information learned from one route discovery to complete a subsequent route discovery.

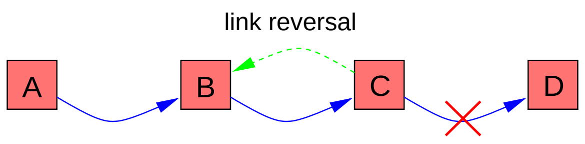
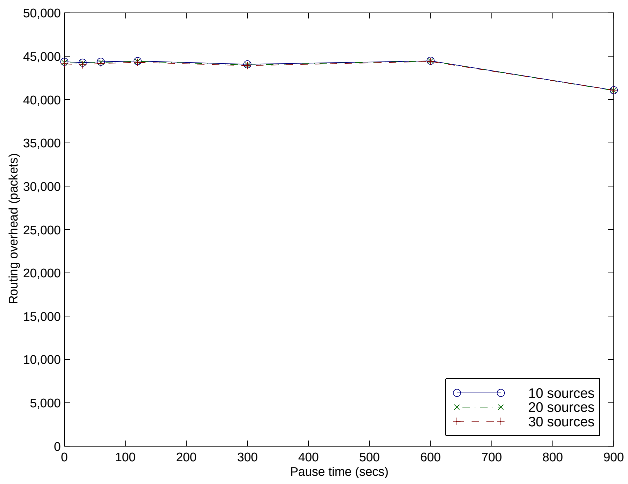
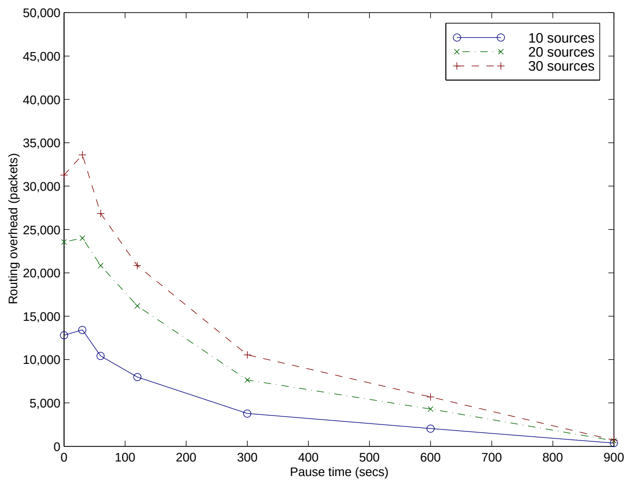


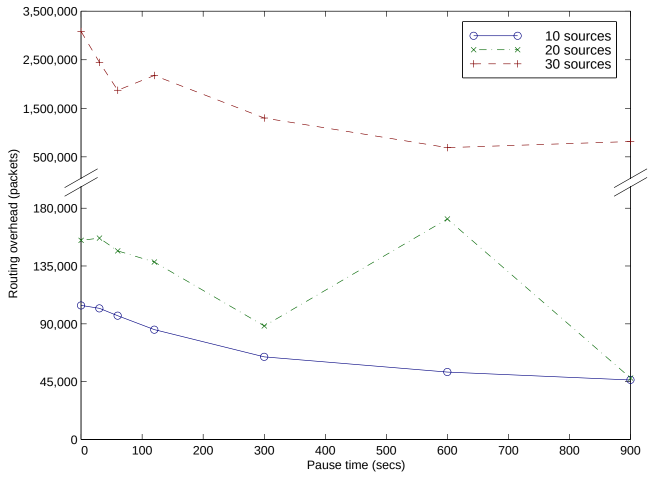
Figure 5.5 The creation of short-lived routing loops during link reversal in TORA.



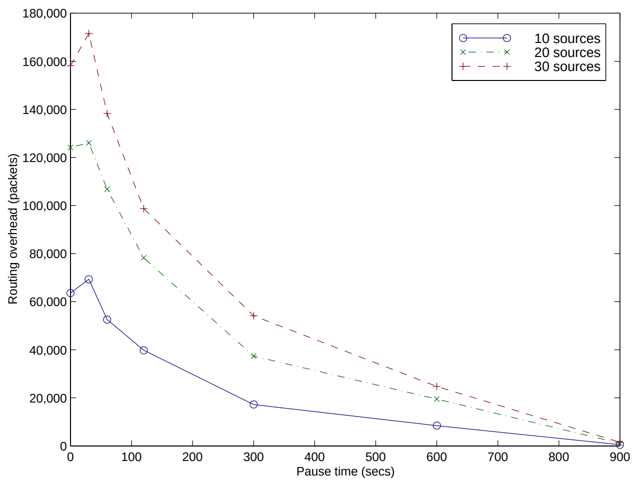
(a) DSDV-SQ



(b) DSR



(c) TORA



(d) AODV-LL

Figure 5.6 Routing overhead as a function of pause time. TORA and AODV-LL are shown on different vertical scales for clarity (see Figure 5.3).

However, the absolute overhead required by DSR and AODV-LL are very different, with AODV-LL requiring about 5 times the overhead of DSR when there is constant node motion (pause time 0). This dramatic increase in AODV-LL's overhead occurs because each of its route discoveries typically propagates to every node in the ad hoc network. For example, at pause time 0 with 30 sources, AODV-LL initiates about 2200 route discoveries per 900-second simulation run, resulting in around 110,000 ROUTE REQUEST transmissions. In contrast, DSR limits the scope and overhead of ROUTE REQUEST packets by using caching from forwarded and promiscuously overheard packets and using non-propagating ROUTE REQUESTs as described in Section 5.1.2, which results in DSR sending only 950 non-propagating requests and 300 propagating requests per simulation run.

TORA's overhead is the sum of a constant mobility-independent overhead and a variable mobility-dependent overhead. The constant overhead is the result of IMEP's neighbor discovery mechanism, which requires each node to transmit at least 1 HELLO packet per BEACON period (1 second). For 900-second simulations with 50 nodes, this results in a minimum overhead of 45,000 packets. The variable part of the overhead consists of the routing packets TORA uses to create and maintain routes, multiplied by the number of retransmission and acknowledgment packets IMEP uses to ensure their reliable, in-order delivery.

In many of our scenarios with 30 sources, TORA essentially underwent congestive collapse. As shown in Figure 5.7, a positive feedback loop developed in TORA/IMEP wherein the number of UPDATE routing packets sent caused numerous MAC-layer collisions, which in turn caused data, ACK, UPDATE, and HELLO packets to be lost. The loss of these packets caused IMEP to erroneously believe that links to its neighbors were breaking, even in the pause time 900 scenarios when all nodes were stationary. TORA reacted to the perceived link breakages by sending more UPDATEs, which closed the feedback loop by directly causing more congestion. Most importantly, each UPDATE sent required reliable delivery, which increased the system's exposure to additional erroneous link failure detections, since the failure to receive an ACK from retransmitted UPDATEs was treated as a link failure indication. In the worst runs, TORA generated over 10 million objects, which IMEP aggregated into 1.6 million packets requiring reliable delivery. In the few 30-source runs where congestion did not develop, the overhead varied from 639,000 packets at pause time 0 to 47,000 packets at pause time 900.

DSDV-SQ has approximately constant overhead, regardless of movement rate or offered traffic load. This constant behavior arises because each destination **D** broadcasts a periodic update with a new sequence number every 15 seconds. With 50 unsynchronized nodes in the simulation, at least one node broadcasts a periodic update during each second. DSDV-SQ considers the receipt of a new sequence number for a node to be important enough to distribute immediately (Section 5.1.1), so each node that receives **D**'s periodic update generates a triggered update. These triggered updates flood the network, as each node receiving one learns a new sequence number and so also generates a triggered update. Each node limits the rate at which it sends triggered updates to one per second, but since there is at least one new sequence number

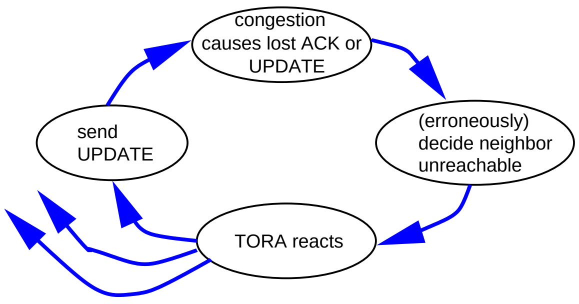

congestion causes lost ACK or UPDATE
(erroneously) decide neighbor unreachable
TORA reacts
send UPDATE

Figure 5.7 The positive feedback loop created by IMEP's reliable broadcast in a shared-media wireless network.

per second, every node transmits triggered updates at the maximum permitted rate. Therefore, although the base periodic action of DSDV-SQ is once per 15 seconds, the effective rate of a group of nodes is one update per node per second, yielding an overhead of 45,000 packets for a 900-second, 50-node simulation.

5.3.4. Path Optimality Details

As described in Section 3.1, an internal mechanism of our simulator knows the length of the shortest possible path between all nodes in the network at any time and labels all packets with this path length when they are originated. Figure 5.8 shows the difference between this shortest path length and the length of the paths actually taken by data packets. A difference of 0 means the packet took a shortest path, and a difference greater than 0 indicates the number of extra hops the packet took.

Both DSDV-SQ and DSR use routes very close to optimal. TORA and AODV-LL each have a significant tail, taking up to 4 or more hops longer than optimal for some packets, although TORA was not designed to find shortest paths. For space reasons, Figure 5.8 aggregates the data from all pause times into one graph. When the data are broken out by pause time, DSDV-SQ and DSR do very well regardless of pause time, with no statistically significant change in optimality of routing with respect to node mobility rate.

TORA and AODV-LL, on the other hand, each show a significant difference with respect to pause time in the length of the routes they use relative to the shortest possible routes. When node mobility is very low, they use routes that are significantly closer to the shortest possible routes than when nodes are moving. As an explicit design choice, TORA does not attempt to find shortest path routes, and so it is not surprising that it has worse path optimality as pause time decreases. AODV-LL has worse path optimality than DSR because

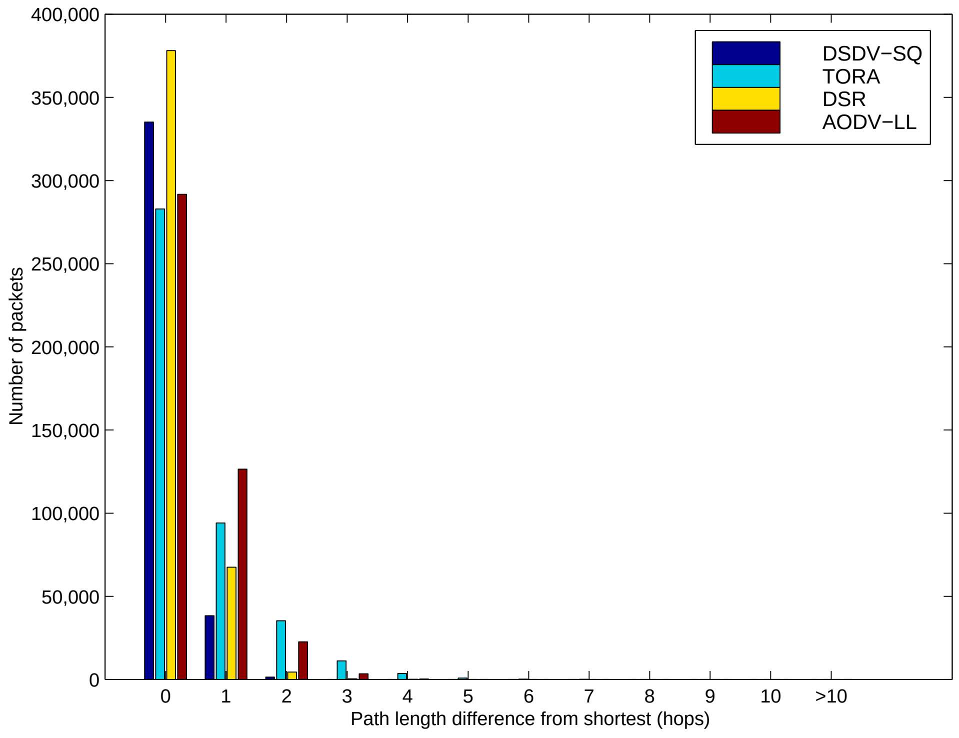


Figure 5.8 Difference between the number of hops each packet took to reach its destination and the optimal number of hops required. Data is for 20 sources.

it does include an optimization like DSR's Gratuitous ROUTE REPLYs that discovers and uses shorter routes as they become available. The difference is path optimality is greatest when node mobility is highest, as that is when the greatest need for route shortening exists. When the nodes are stationary, DSR and AODV-LL will discover routes of equal length.

5.3.5. Lower Speed of Node Movement

In order to explore how the protocols scale as the rate of topology change varies, we changed the maximum node speed from 20 m/s to 1 m/s and re-evaluated all four protocols over scenario files using this lower movement speed. Figures 5.9 and 5.10 show the results of this experiment when using 20 sources. All of the protocols deliver more than 98.5% of their packets at this movement speed. Unlike in the 20 m/s scenarios, where DSDV-SQ could not converge, it delivers excellent performance in the 1 m/s scenarios.

Even at this slower rate of movement, each of the routing protocols generated very different amounts of overhead. Neither DSR nor AODV-LL were seriously challenged by this set of scenarios, as the overhead increases only mildly as pause time decreases. The separation between DSR and AODV-LL, however, has grown from a factor of 5 to nearly a factor of 10 because DSR's caching is even more effective at lower speeds where the cached information goes stale more slowly.

Due to its largely periodic nature, DSDV-SQ continues to have a constant overhead of approximately 41,000 packets, while TORA's overhead is dominated by the link/status sensing mechanism of IMEP, which amounts to one packet per node per second, or a total of 45,000 packets per simulation (Section 5.3.3).

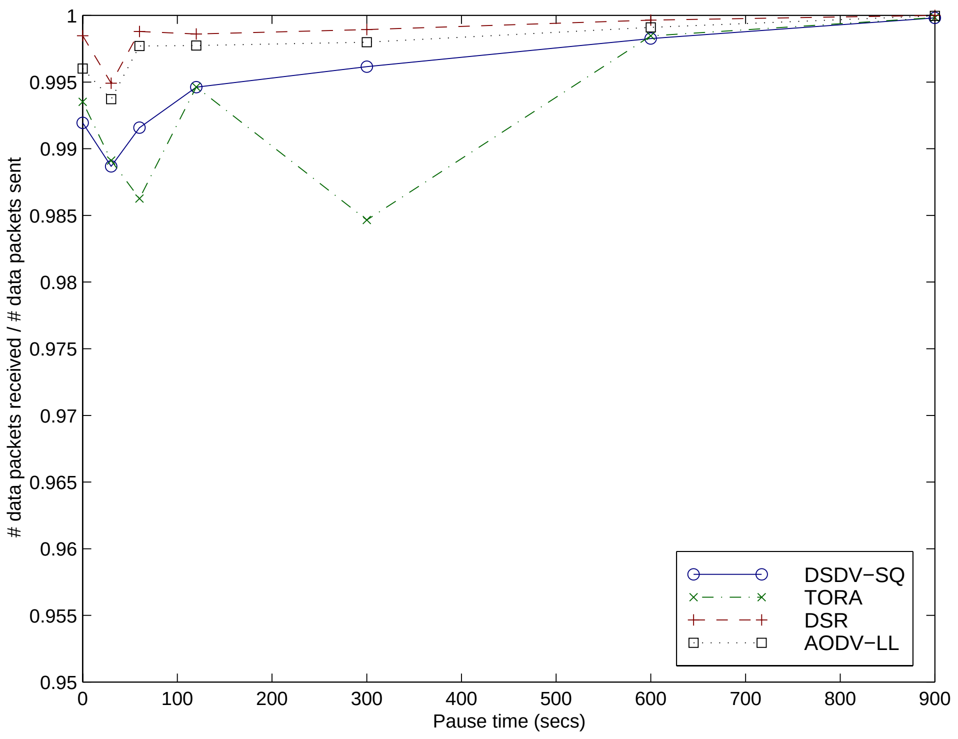


Figure 5.9 Comparison of the fraction of application data packets successfully delivered as a function of pause time. Speed is 1 m/s.

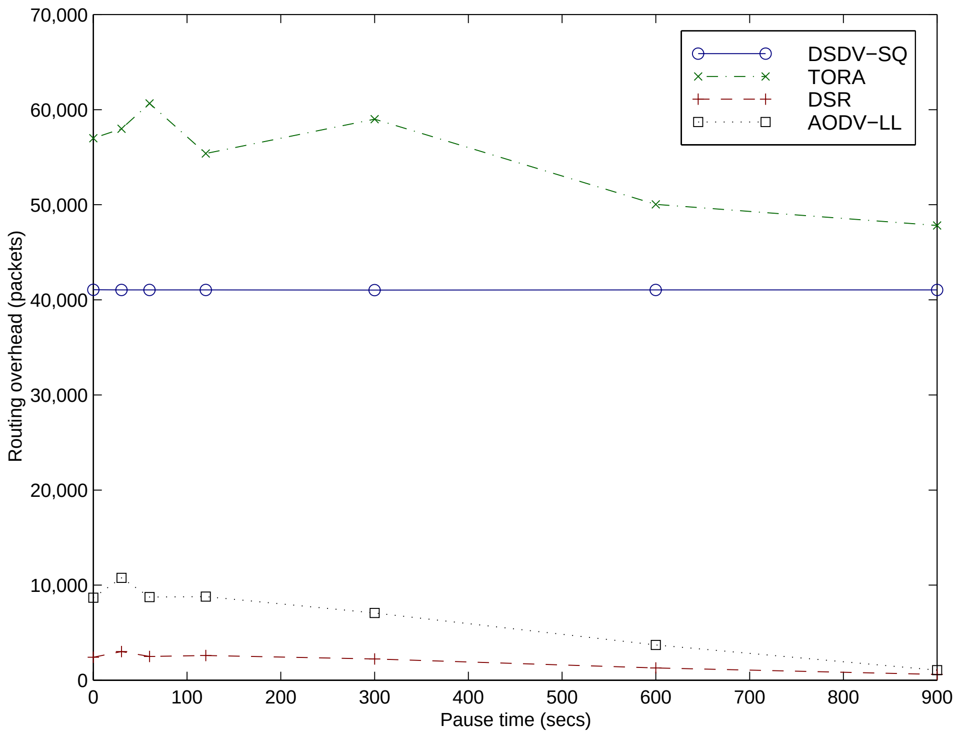
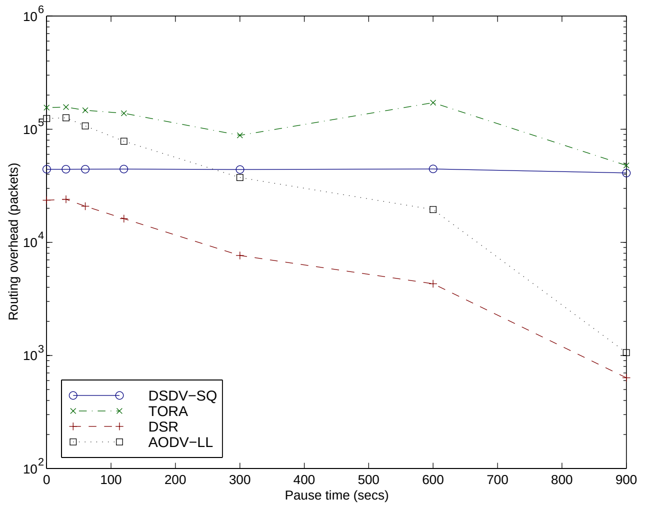
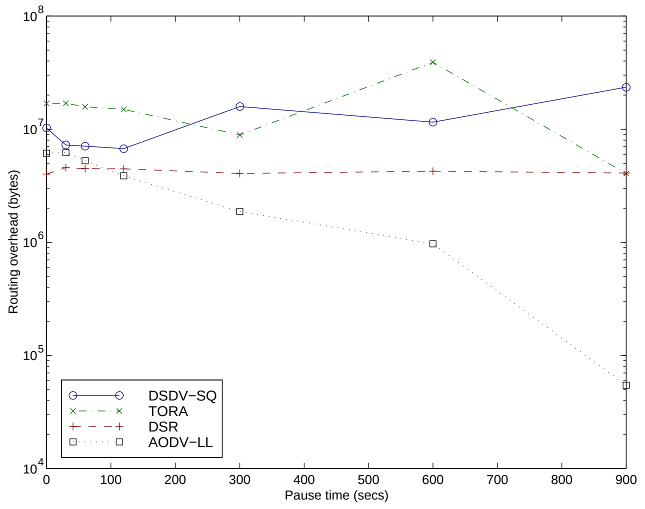


Figure 5.10 Comparison of the number of routing packets sent as a function of pause time. Speed is 1 m/s.



(a) Routing overhead in packets.



(b) Routing overhead in bytes.

Figure 5.11 Contrasting routing overhead in packets and in bytes. Both graphs use semi-log axes.

5.4. Additional Observations

5.4.1. Overhead in Source Routing Protocols

When comparing the number of routing overhead packets sent by each of the protocols, DSR clearly has the lowest overhead (Figure 5.6). The data for 20 sources is reproduced in Figure 5.11(a) on a semi-log axis for clarity. However, if routing overhead is measured in bytes and includes the bytes of the source route header that DSR places in each packet, DSR as implemented in this study becomes more expensive than AODV-LL except at the highest rates of mobility, although it still transmits fewer bytes of routing overhead than does DSDV-SQ or TORA. AODV-LL uses a Route Discovery mechanism based on DSR's, but it creates hop-by-hop routing state in each node along a path in order to eliminate the overhead of source routing from data packets. This reduction in overhead bytes is shown in Figure 5.11(b).

It is unclear whether this improvement in bytes of overhead is significant for real world protocol operation, because the majority of AODV-LL overhead bytes are carried in many small packets. The cost to acquire the medium to transmit a packet is significantly more expensive in terms of power and network utilization than the incremental cost of adding a few bytes to an existing packet, so the actual cost of the source route header in DSR is less than the number of bytes might indicate. A completely fair comparison based on overhead in bytes would also have to include the cost of physical layer framing and MAC protocol bytes, which we have deliberately factored out since the routing protocols could be run over many different MAC implementations, each of which would have a different overhead.

Ultimately, concerns about the overhead of source routes in DSR should be largely eliminated by the introduction of path-state [70] to the protocol. Path-state enables DSR to create hop-by-hop routing state in intermediate nodes when needed on-demand, thereby eliminating the need for most source-routes and vastly reducing the byte-overhead of the DSR protocol.

5.4.2. The Effect of Triggered Updates in DSDV

As noted in Section 5.1.1, DSDV can employ either of two strategies for determining when to send triggered updates. In the first strategy, DSDV-SQ, a node sends a triggered update each time it receives a new sequence number for some destination. As shown in Figure 5.12, DSDV-SQ delivers over 99% of its packets for all pause times when the maximum node speed is 1 m/s. In the 20 m/s case, DSDV-SQ's packet delivery ratio falls to 95% at a pause time of 300 seconds and further decreases at higher mobility rates as DSDV-SQ is unable to converge. Figure 5.13 shows that for both the 1 m/s and 20 m/s data, DSDV-SQ's routing overhead is approximately 45,000 packets for all pause times, as discussed in Section 5.3.3.

The second scheme for sending triggered updates, which we call simply DSDV, requires that they be sent only when a new metric is received for a destination. In this case, link breakages are not detected as quickly as in DSDV-SQ, generally resulting in more dropped packets.

For a movement speed of 1 m/s, DSDV delivers fewer packets than DSDV-SQ, with its packet delivery ratio decreasing to 95% at pause time 0. While DSDV's routing overhead is a factor of 4 smaller, the fact that its routing overhead is constant indicates that a movement speed of 1 m/s does not exercise the routing protocol fully. At 20 m/s, both DSDV-SQ and DSDV fail to converge, causing a large percentage of data packets to be dropped. However, the DSDV triggering scheme reduces the relative routing overhead by a factor of 4 at pause time 900 and by a factor of 2 at pause time 0.

5.4.3. Reliability Issues with Broadcast Packets

The broadcast nature of a shared wireless channel offers an apparently efficient way for a node to distribute information to all its neighbors with a single transmission. However, broadcast packets in shared channel wireless networks are inherently less likely to be successfully received than unicast packets, and routing protocols must be designed to tolerate this.

Many link-layer protocols for wireless networks take explicit steps to avoid the exposed terminal problem and the hidden terminal problem [94]. For example, IEEE 802.11 precedes each unicast packet with a Request-to-Send/Clear-to-Send(RTS/CTS) exchange that notifies surrounding nodes not to make transmissions that would interfere with the reception of the unicast packet. The RTS/CTS exchange protects unicast packets because the RTS is directed at a specific receiver, who transmits the CTS when it is ready to receive the packet. The transmission of the CTS also notifies any nodes around the receiver that it is about to receive a unicast packet, and so they should refrain from making any transmissions themselves for the duration of the incoming packet to avoid interfering with the packet's reception. The sender continues to retransmit the RTS until it successfully receives the CTS, indicating the wireless medium at the receiver now available for it to send its packet, or the sender gives up.

While an RTS/CTS exchange, or similar technique, can be used to protect unicast packets, no such technique exists to protect broadcast packets. Because broadcast packets are intended for any nearby node, and not directed to a particular receiver, the sender does not know which nodes to expect to receive a CTS from. Even if it did, multiple nodes attempting to send a CTS at approximately the same time would cause tremendous mutual interference at the node with a broadcast packet to send. Other techniques are possible, such as globally assigning each node a time slot during which it can transmit its broadcast packets, but tend to have very low efficiency. Since there is no known way to reasonably reserve the wireless medium at all the receivers before transmitting a broadcast packet on a shared wireless channel, broadcast packets will be received successfully only by those those nodes that, by chance, happen to have idle wireless medium around their receivers. Consequently, broadcast packets are inherently less reliable than unicast packets on such channels.

Upon sampling a number of our scenarios, I found that over any single hop, 99.8% of unicast data packets are received successfully, while only 92.6% of broadcast packets are received, based on counting the number of receivers within transmission range of the broadcasting node. The difference between the two numbers is due to collisions. Future work should examine how the difference varies with the average degree of the nodes, the size of the broadcast packets, and the relative proportion of broadcast packets.

The difference in reliability between broadcast and unicast packets does not exist in wired networks, and it represents a fundamental limitation of wireless networks that must be accounted for in the design of ad hoc network routing protocols. Wireless routing protocols that broadcast their control packets must be designed with the expectation that the control packets they send will be received less reliably than the data packets they direct the forwarding of. There appear to be only two alternatives to designing a robust routing protocol that tolerates the loss of control packets. The first is for each node to unicast control packets to each of its neighbors. This alternative, however, requires the routing protocol to keep track of who its neighbors are and increases the number of transmissions required to deliver each control packet. The second approach is to use a reliable broadcast mechanism. As shown in this chapter, however, reliable broadcast requires even more overhead than the first alternative (not surprisingly, as it is a superset of the functionality of the first alternative).

5.4.4. Interaction of ARP with On-Demand Protocols

When an on-demand routing protocol receives packets to a destination for which it does not have a route, the protocol typically buffers the packets in the routing layer until it can discover a route for the packets. Once the routing protocol finds a route, it sends the queued packets down to the link-layer for transmission. In the course of our early experiments, however, we observed a serious layer-integration problem that would effect any on-demand protocol running on top of an ARP implementation similar to that in BSD Unix [111].

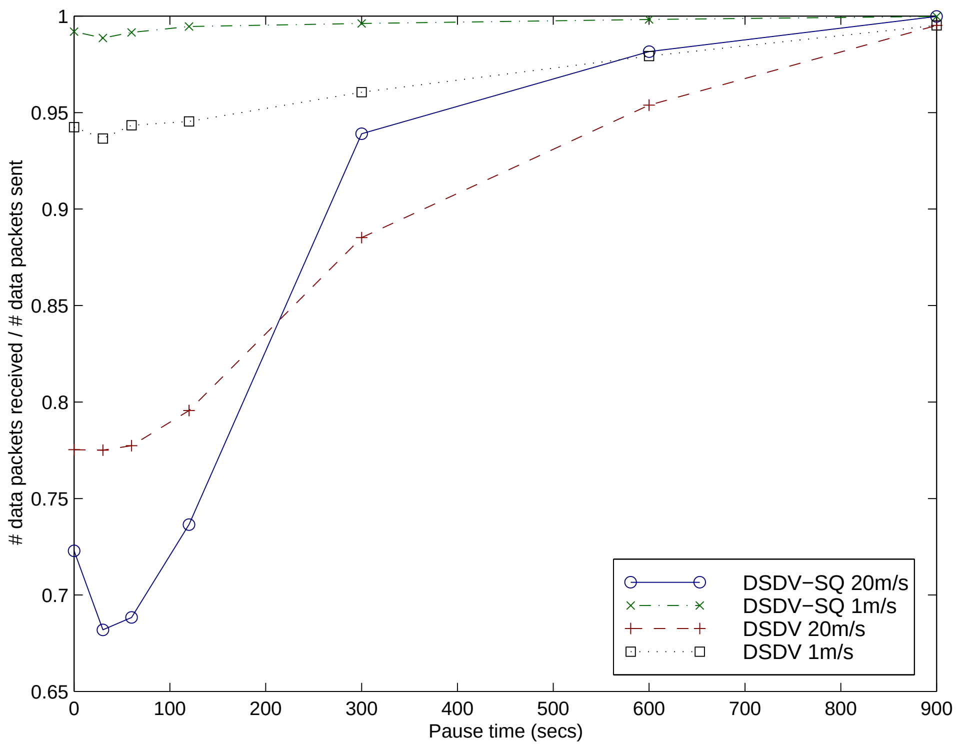


Figure 5.12 Fraction of originated data packets successfully delivered by DSDV-SQ and DSDV.

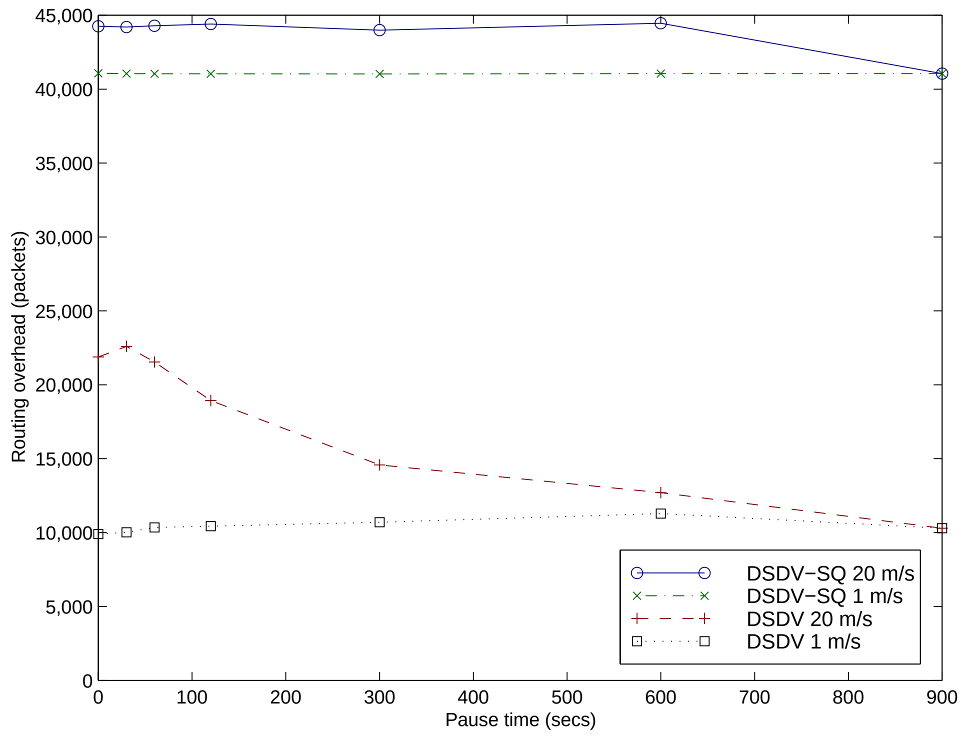


Figure 5.13 Routing overhead as a function of pause time for DSDV-SQ and DSDV.

Our ARP code, like the BSD code, buffers one packet per destination awaiting a link layer address. If a series of packets are passed by the routing layer to the ARP code with a next-hop destination whose link-layer address is unknown, all but the last of these packets will be dropped by ARP. In our simulation, we remedied this by pacing the rate at which packets are passed from the routing queue, though the implementation of ARP could be modified to buffer additional packets, or the routing protocol could be coded to check that the ARP layer has a link-layer address for the next-hop before passing it packets from the routing queue.

5.5. Chapter Summary

The simulation study described in this chapter has two main results. The first is the verification of the usefulness of our modifications to the *ns-2* network simulator to provide an accurate simulation of the MAC and physical-layer behavior of the IEEE 802.11 wireless LAN standard, including a realistic wireless transmission channel model. This new simulation environment provides a powerful tool for evaluating ad hoc networking protocols and other wireless protocols and applications.

The second result of this study is an exploration of several large areas of the design space for ad hoc network routing protocols. The set of protocols we studied (DSDV, TORA, DSR, and AODV) and the variants of these protocols we looked at cover a range of design choices, including periodic advertisements or on-demand route discovery, use of feedback from the MAC layer to indicate a failure to forward a packet to the next hop, and hop-by-hop routing or source routing.

We found each of the protocols performs well in some cases yet has certain drawbacks in others.

5.5.1. DSDV

DSDV performs quite predictably, delivering virtually all data packets when node mobility rate and movement speed are low, yet failing to converge as node mobility increases.

As noted in Section 5.1.1, we found that DSDV works better when it is used without link-layer feedback. For that reason, it can be argued that DSDV is treated unfairly in this study, because all the per-data-packet acknowledgments sent by the MAC protocol could be treated as routing overhead that should be charged against the other protocols. For an environment that has a rate of topology change on the order of the 1m/s data shown here (i.e., 1 link status change per second), DSDV would actually have a lower routing overhead than either TORA, DSR, or AODV while providing an equivalent packet delivery ratio. This effect occurs because the routing overhead would be dominated by the number of per-data-packet acknowledgments, which increases with the number of data packets sent. However, as described in Section 7.4, even when stationary, the nodes in a real ad hoc network see significant rates of topology change due to wireless propagation effects and multipath resulting from objects in the world moving around the nodes. These real-world effects would make the use of DSDV risky in a real ad hoc network.

5.5.2. TORA

Although the worst performer in our experiments in terms of routing packet overhead, TORA still delivered over 90% of the packets in scenarios with 10 or 20 sources. At 30 sources, the network was unable to handle all of the traffic generated by the routing protocol, and a significant fraction of data packets were dropped in these scenarios.

Packet loss in TORA resulted from two main sources. The first is short-lived routing loops caused by the delay between a link reversal at one node and the link-reversals at neighboring nodes. The second, and more serious one, is packet loss due to congestion caused by a positive feedback loop in the behavior of the IMEP reliable broadcast protocol used by TORA to distribute its routing updates. Given the increased probability of loss for broadcast packets in wireless networks (Section 5.4.3), our experience with TORA/IMEP argues that routing protocols must be designed to be tolerant of the loss of their broadcast packets, rather than attempting to eliminate the loss via a reliable broadcast algorithm.

5.5.3. DSR

The performance of DSR was very good at all mobility rates and movement speeds we studied. The analysis of routing overhead bytes, however, shows the high cost of including a source route in each data packet and motivates the need for adding path-state [70] to the protocol. Other conclusions about DSR's performance are distributed throughout this thesis and so are not repeated here.

5.5.4. AODV

AODV performs almost as well as DSR at all mobility rates and movement speeds and accomplishes its goal of eliminating source routing overhead, but at high rates of node mobility it requires the transmission of many routing overhead packets than DSR requires. The better performance of DSR with respect to AODV is due to DSR's ability to discover and use multiple routes and its use of Route Caches to contain Route Discoveries. The AODV protocol has undergone significant development since this study was performed. Some of the added features, such as an expanding ring search as suggested earlier by DSR [51], would improve AODV's performance on the scenarios in our study.

5.5.5. On-Demand Routing Mechanisms

Beyond the results for any particular protocol, two overall trends in this study are clearly visible. The first trend is that the less dependent on periodic behavior a protocol is, the better it appears to perform. For example, DSR has the best performance of the protocols we studied, and it is completely on-demand. AODV's performance significantly improved when we created AODV-LL by removing periodic mechanisms from AODV and making its Route Maintenance mechanisms operate on-demand. DSDV, the least on-demand protocol we studied, has the worst behavior at high rates of mobility.

The second trend is that protocols perform better the more they are able to reduce the number of nodes that must react to a topology change. For example, DSDV attempts to include on-demand behavior through the use of triggered updates. However, the fact that each topology change results in the creation of a routing update with a new sequence number that must be spread throughout the network results in a very high overhead, because each topology change requires action at each node. Similarly, the difference in performance between AODV-LL and DSR at high rates of mobility is due to the improved containment of Route Discoveries that DSR achieves.

Chapter 6

Emulation of Ad Hoc Networks

In general, evaluating a real system with just simulation is not practical. It is difficult to model the full complexity of a real application inside a simulator, and most real applications systems have been significantly performance tuned. Having to repeat this effort to create a simulation model is wasteful.

Network emulation allows application and protocol designers to combine components of real code or real hardware with simulated components, thereby creating a composite system. Such a composite system enables more flexible experiments than are possible with either a purely simulated or completely implemented system. In effect, emulation blurs the line between simulation and full-scale implementation. By doing so, it allows designers to maximize their understanding of their system by varying the trade-off between fidelity to real-world effects and experimental control. If designers create composite systems with more of the system implemented with real components, they will see more real-world effects in the system's behavior. If they create composite systems having more simulated components, they will be able to achieve better experimental control, since simulated components have better repeatability than real components.

The most important characteristic of a network emulation methodology is its *fidelity*, meaning how closely the inputs to the real components of the composite system match the inputs that would be experienced by the components in a real deployed implementation. When the emulation methodology causes real components to behave differently than they would in a real network, we will call these glitches *emulation artifacts*. The fewer emulation artifacts a system exhibits, the higher its fidelity.

In developing the network emulation systems described in this chapter, I attempted to achieve the following goals:

- Create the environment of an ad hoc network with sufficient fidelity that a performance study of network protocols or applications conducted in the emulation system will match the performance of that protocol or application in a full-scale implementation.

- Permit the use of an unmodified real application and a real user pattern, eliminating the need to create a simulation model of already implemented systems.

- Allow experiments to be conducted without the need to deploy and operate physical machines in the field.

Many types of network emulation are possible, and this chapter begins by presenting a novel framework for classifying them. The chapter then gives examples of two major methods for implementing network emulation: direct emulation and trace modulation. While direct emulation has been previously described in the literature [29], this is the first work to apply it to ad hoc networks and validate that it actually works as expected. Finally, the chapter describes how the direct emulation system was used by the developers of the Coda distributed file system [12, 101] to experiment with Coda in an ad hoc network.

6.1. A Framework for Classifying Emulation Systems

The different types of emulation I developed can be described in a single general framework by the concept of an *emulation cut.* If we imagine drawing each node in the network using the ISO 7-layer model, an emulation cut is a line across the network stack that separates the node into two pieces. Functions above the cut are fully and physically implemented, and those below the cut are simulated or represented by some type of model.

For example, a node that is completely simulated appears as in Figure 6.1. The emulation cut is above the application layer of the node, meaning that none of the node is physically implemented. A typical *network simulation* is depicted by Figure 6.2, where all the nodes in the network are completely simulated and therefore lie completely below the emulation cut.

6.1.1. Link Layer Emulation

Figure 6.3 shows a node where the functionality above the link layer is physically implemented, but the link layer and functions below it are simulated. When a network is built out of such nodes, as shown in Figure 6.4, we will call it an example of *link layer emulation.* In a link layer emulation, physical hardware is required for each node in the network. The advantage of this type of emulation, however, is that the hardware can be physically stationary and located inside a single lab. There is no longer a need to move the nodes around to vary the network topology, as would be required in a full-scale deployed ad hoc network. Instead, the link layer can be designed to model a changing topology by receiving all the packets sent by the other nodes in the network, and then either passing the packets up to higher layers or filtering them out, depending on what the current emulated topology is.

6.1.2. Doppelganger Emulation

Figure 6.5 depicts a network where nodes **A** and **B** are completely implemented to the network layer, but the other layers in those nodes and all layer in the other nodes are simulated. We will call this a *doppelganger emulation.* In a doppelganger emulation, a simulation of the network is run in real-time, with each node in the physical network represented by a corresponding unique node in the simulation. All of the behavior of the network from the network layer down is modeled by the action of the nodes inside the simulation. For node **A**, which has both simulated and real components, the node inside the simulation acts as a stand-in, or *doppelganger*, for the real components.

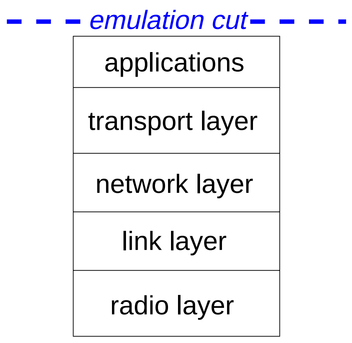


Figure 6.1 The representation of a node that is entirely simulated, as all layers fall below the emulation cut.

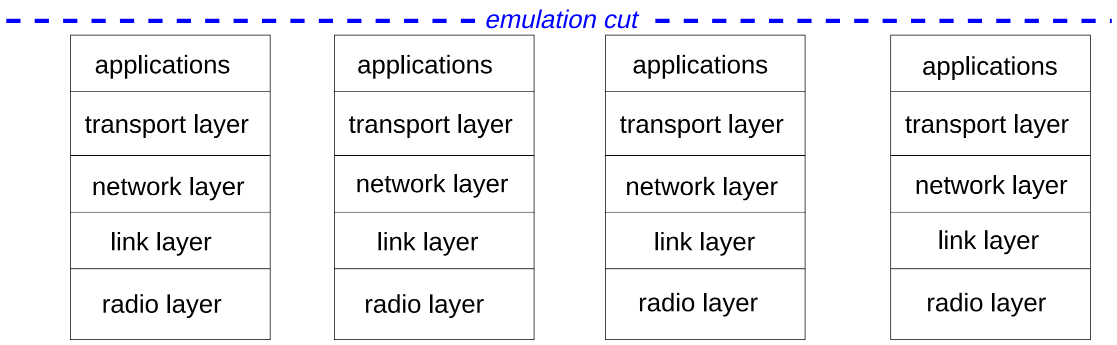


Figure 6.2 Representation of a network where all nodes are simulated.

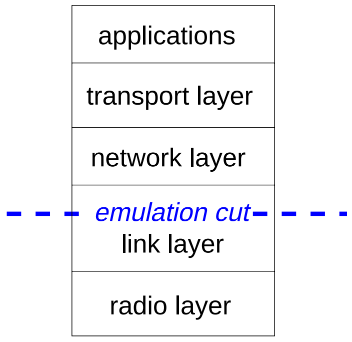


Figure 6.3 The representation of a node in a *link layer emulation*, where the layers above the emulation cut are fully implemented on physical hardware. The layers below the cut are emulated.

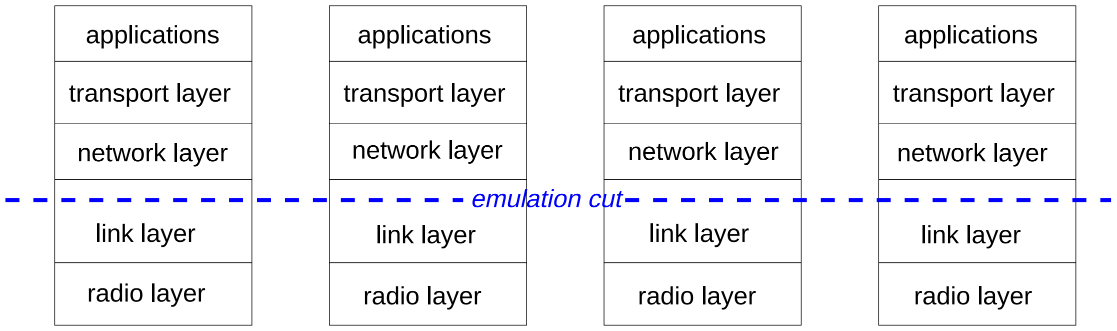


Figure 6.4 A depiction of a *link layer emulation* network.

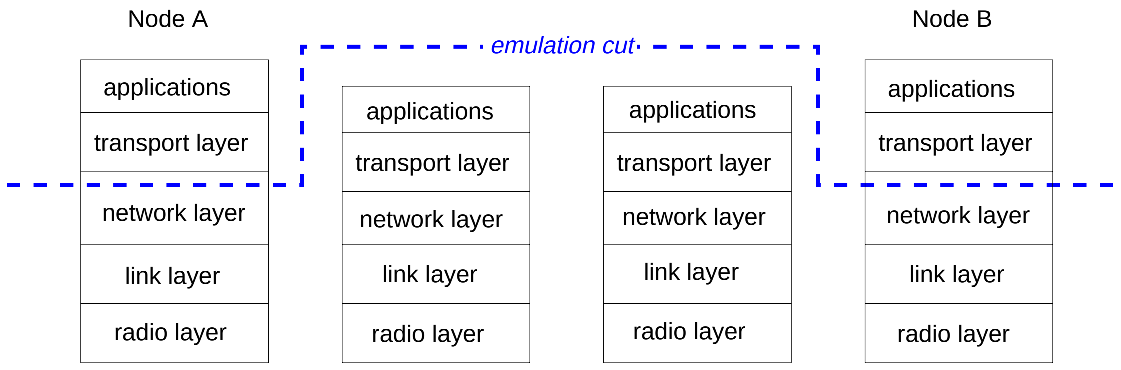


Figure 6.5 An example of a *doppelganger emulation* where the emulation cut passes through different nodes at different layers.

Doppelganger emulation has several advantages over other forms of emulation. First, the test network can include more nodes than can be implemented with the available physical hardware. Second, doppelganger emulation allows researchers to experiment with an application that has been fully implemented, without having to reimplement it as a simulation model or change its code in any way. Third, doppelganger emulation allows real users to actually interact with the application as it is tested, something typically not possible with pure simulation. This ability is particularly important if the application is embodied in a specialized device, such as a wearable computer, and the goal of the experiment is to evaluate the user's reaction to the device.

6.1.3. General Emulation Cuts and Virtual Node Emulation

The emulation cuts described above are those that I have found most useful in practice. In general, however, the emulation cut can be drawn across the network stacks at any layer. In order to support an emulation cut that varies in layer between nodes, as shown in Figure 6.6, it must be the case that either nodes **A** and **B** do not exchange layer M Protocol Data Units (PDUs), or that the simulation code on node **B** is complete enough to exchange PDUs with the real code implementing layer M on **A**, including all the details such as packet formats, data representations, and all protocol behavior.

Creating a simulation model complete enough to peer with a working implementation is roughly as much work as creating the implementation and so gives up many of the time-saving benefits of emulation. We will

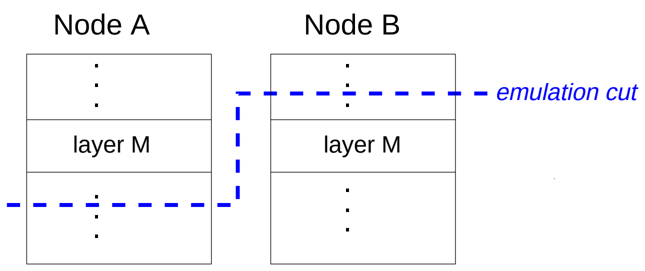


Figure 6.6 A generic emulation cut giving Node A a physical implementation of layer M and node B a simulated model of layer M.

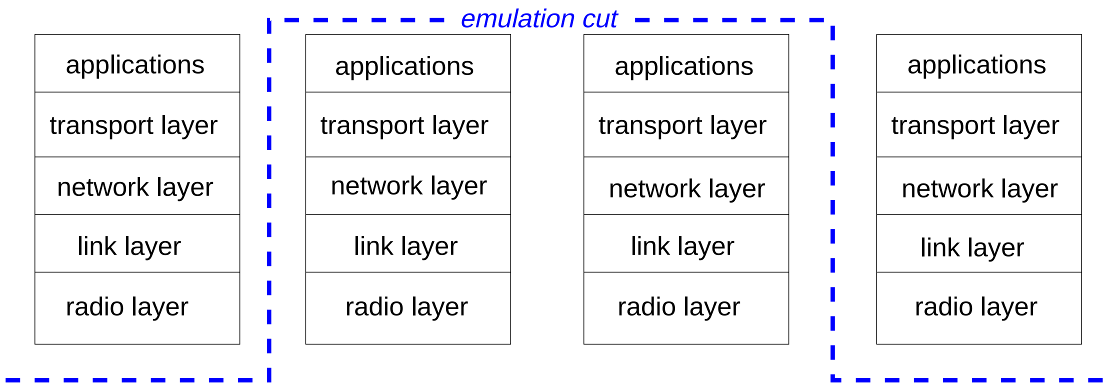


Figure 6.7 A depiction of a *virtual node emulation* where the two middle nodes act as nodes in a virtual network capable of peering with the other completely physically-implemented nodes at any layer.

refer to this form of emulation as *virtual node emulation*, and an example is shown in Figure 6.7. S. Keshav was developing a commercial product that embodied a form of virtual node emulation while working at the Ensim Corporation. Called the Etrapid product, it was going to contain a complete implementation of a TCP/IP network stack, and thereby enable the creation of virtual nodes that would exchange packets as peers with real network nodes. Unfortunately, to the best of my knowledge, this product was never released.

In addition to emulation cuts which go completely across the network stack at a particular layer, it also makes sense to think of *emulation slices* that place the boundary between simulated code and implemented code down the middle of network layer.[1] An example emulation slice would be a network layer where the packet handling mechanisms were implemented in real code, but the routing protocol software was simulated on a central machine and only the appropriate routing table entries were moved to the actual machines and installed in the implemented routing tables. Creating an emulation slice is very similar to a general emulation cut, in that the simulated modules on one side of the slice must be able to completely fulfill their interfaces to the implemented modules on the other side of slice, and, in effect, impersonate completely implemented modules.

6.1.4. Limitations of the Framework

The framework developed in this section assumes that the physically-implemented portion of a node always lies above the emulation cut, and the simulated portion lies below. All the emulation systems I experimented with follow this pattern, although it is possible to reverse it.

For example, a researcher might wish to experiment with protocols running across a specific physical link in some environment but be unsure how to accurately model wireless propagation in that environment. The researcher could conceivably build simulation models of the protocols and place physical radio transceivers in the specific environment. Whenever the simulated protocol model sent data, the simulation system would cause the physical transmitter to actually send a packet of the same size. If the physical test packet sent by the physical transmitter was received error-free by the physical receiver, then the simulated packet would be delivered to the intended destination protocol model.

Although an inverted emulation cut like that described above is possible, it is not very useful. Such a system depends critically on an assumption that the physical environment is time-invariant, which is

[1]The idea for an emulation slice was suggested by Martha Steenstrup.

frequently not true. Since the protocol models are unlikely to generate data packets at exactly the same times as a real implementation would, the test packets will be sent on the physical link at different times than a real implementation would send them. This means that the protocol models will observe a different pattern of transmission success and failure than the a real implementation would unless the physical environment is time-invariant.

6.2. Trace Modulation and Direct Emulation

We have experimented with two different means for effecting the emulation of multi-hop ad hoc networks: *trace modulation* and *direct emulation.* The goal of both emulation methods is to cause the packets sent between physically-implemented components to behave as if the components were part of nodes moving and interacting in a deployed ad hoc network. For ease of explanation, in this section we will assume that there are two nodes, **A** and **B**, with physically-implemented components and that these components are communicating.

6.2.1. Trace Modulation

Trace modulation is based on a communication-theoretic model of networks. As shown by the overview in Figure 6.8, a logical channel model is placed in the path of all Protocol Data Units (PDUs) flowing between layer M on node **A** and layer M on node **B**. The logical channel model is driven by a *trace file* that describes the properties that the logical channel should have, such as its bandwidth, delay, or packet drop rate. As in classical communication theory, the logical channel should be thought of as a very flexible abstraction. It can be used to model either a one-hop link between two nodes or an entire multi-hop network, depending solely on the trace file used to drive it.

The trace file specifies how the channel properties change as a function of time, which enables trace modulation to model the connectivity changes that occur in mobile ad hoc networks. For example, trace modulation can represent the change in connectivity caused when two nodes move out of range of each other, by setting the drop rate of the channel between **A** and **B** to 100 percent when **A** and **B** are no longer in range of each other. More sophisticated trace files can even specify the roll-off in the probability of successful packet reception as the nodes move apart, thereby increasing the fidelity of the emulation.

Trace modulation suffers from two major drawbacks. First, it is difficult to create a trace file that accurately characterizes what happens to the PDUs sent from **A** to **B** in the physical network that the emulation

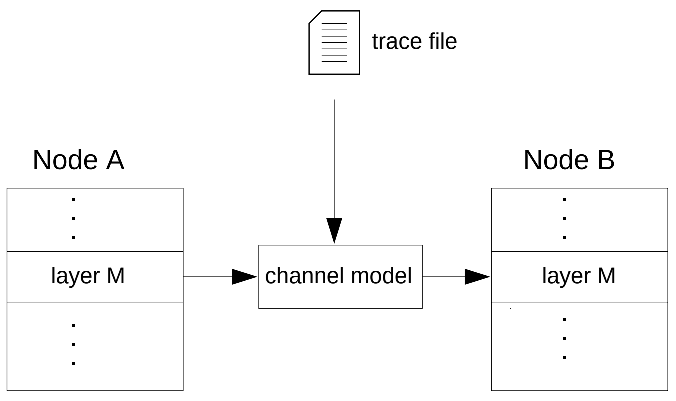


Figure 6.8 Logical operation of trace modulation.

is attempting to realize. Second, trace modulation depends on an assumption that all the complexities of a communication path between two nodes, which could stretch over an entire multi-hop network, can be well modeled by a logical channel model with relatively few parameters. To date, this assumption has only been validated in a few specific cases [77].

6.2.2. Direct Emulation

In contrast to trace modulation, which assumes we can model a logical communication path with relatively few parameters, direct emulation is predicated on the notion that we can have a detailed simulation of the ad hoc network run in real-time. Rather than abstract into the parameters of the trace file all the behavior of the network that connects nodes **A** and **B**, direct emulation uses a *detailed simulation model* of the network to *directly model* all the salient behavior of the network.

In order to implement direct emulation, we must arrange for the actions of the physical components of the composite network to affect the behavior of the modeled components inside the simulation, and vice-versa. Whenever a physical component causes an externally visible event which could affect the behavior of the simulated components of the network, we inject an event into the real-time simulation that represents the action taken by the physical component. Similarly, whenever a simulation event occurs that would be visible to a physically-implemented component of the network, the simulation generates an event on the physical component that is equivalent to what would happen to the component if all the simulated components were also physically implemented.

Figure 6.9 shows an example of the operation of direct emulation. Consider a packet sent from node **A** to node **B** in this network. As shown by the emulation cut drawn in Figure 6.9, the application and transport layer on **A** and **B** are physically implemented and the rest of the network is simulated. After a real application running on **A** originates a packet, it passes through the network stack until it reaches the level of the emulation cut shown at marker **①**. As it crosses the emulation cut (marker **②**), the data in the packet[2] is transported to the machine running the real-time simulation, and wrapped up into a simulation event. The simulation event must preserve enough information to enable the conversion of the event back into the original object, if required. The event is then handled by the simulation as normal, creating other events in the network stacks of the simulated nodes, until it eventually reaches the point shown by marker **③** in the

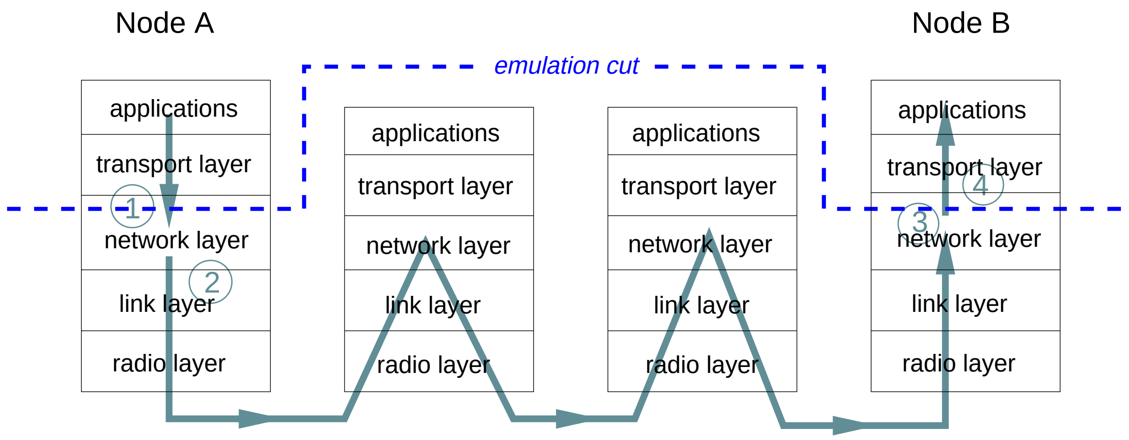


Figure 6.9 Logical operation of a direct emulation.

[2]More formally, it is not yet a packet at this point, but a Service Data Unit (SDU).

simulated network layer of node **B**. When the simulation detects the event crossing back over the emulation cut, it unwraps the data from the event. The emulation system then transports the data to the physical implementation of node **B**, where it is passed up into the real transport layer as shown at marker $\textcircled{4}$.

With direct emulation, the length of time it takes **B** to receive the data after **A** "sends" it — or whether the data is received at all — is determined by the operation of the simulation. If the simulation accurately models the real-time behavior of some ad hoc network, then the communications between **A** and **B** will see the same effects they would if they were actually participating in a real implementation of that ad hoc network. As with trace modulation, direct emulation also has two major drawbacks. The first is the need to minimize the time it takes to move data between physical nodes and the simulation machine. The second is the need to prevent the simulation from falling behind real-time operation. A failure to meet these needs will result in the introduction of emulation artifacts that can distort the performance of the system under test.

6.3. Implementing Link Layer Emulation Via Trace Modulation

In order to test the code our group developed for our DSR testbed (Chapter 7), I implemented a link layer emulation scheme as shown in Figure 6.10. Since the goal of these experiments was to test an actual DSR implementation, we already had enough laptops to assign one laptop to each node in the network. Combined with the fact that the code we wished to test is located in the network layer, this led me to link layer emulation implemented via trace modulation as the most attractive solution.

To conduct the tests, I placed all of the physical laptops running the actual DSR code together in our lab, connecting them into a LAN using either Ethernet or the actual broadcast radio transceivers. Because the machines are connected by a broadcast media, each machine receives a copy of any packet sent by one of the others. This allows us to emulate any desired topology by simply preventing the network stack of

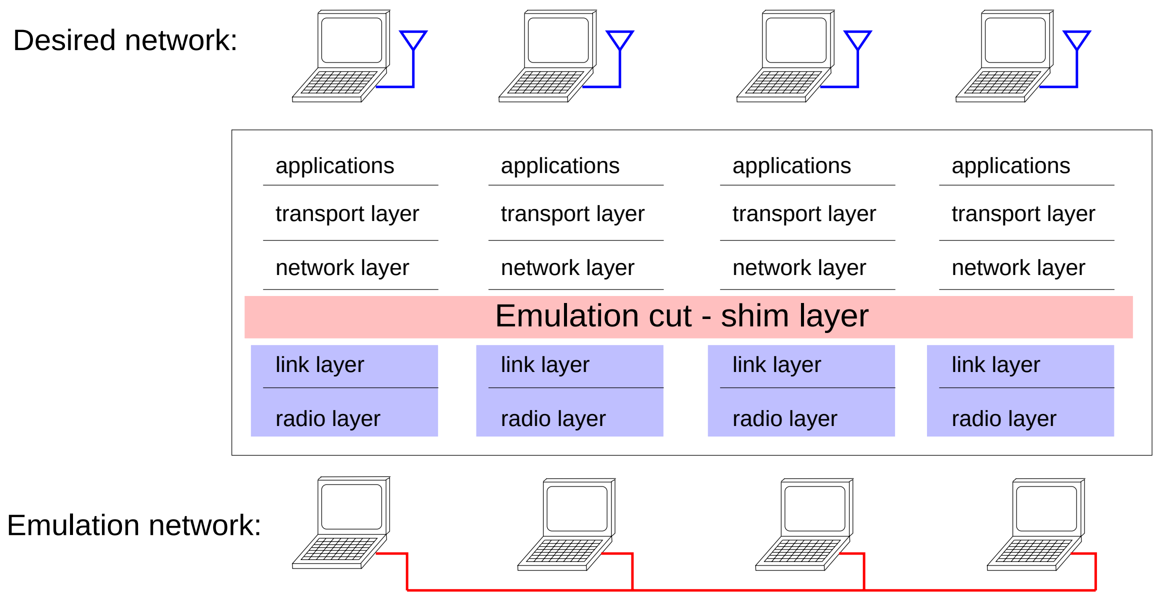


Figure 6.10 The placement of a packet killer as a shim layer at the level of the emulation cut creates a system for link layer emulation. Using this system, the desired network of independent mobile machines communicating via radio interfaces can be realized using stationary machines connected by an Ethernet in the lab.

each machine from processing the packets that it would not have been able to receive had the nodes actually been deployed in the field and separated by varying distances. To achieve this, we implemented a packet killer called the `macfilter`, which is placed between the physical network interface and the network layer input queue. The `macfilter` checks the MAC source address of each received packet against a list of prohibited addresses. Packets whose source addresses are found on the list are silently dropped. Emulating the motion of the nodes then reduces to the problem of loading each `macfilter` with the proper list of prohibited addresses at appropriate times.

The `macfilter` and the Ethernet together implement the logical channel model described earlier in Section 6.2.1. We control the `macfilter` packet killer with a *trace file* consisting of the MAC source address lists and the times at which these lists should be loaded into the `macfilter`. The desired node motions are first created using our `ad-hockey` graphical scenario creation tool to draw the paths the nodes should move along (Figure 6.11). The `ad-hockey` tool then generates the trace files using a trivial propagation model, where any nodes falling within a fixed radio range of each other can communicate, and any falling outside the range cannot. As the emulation runs, `ad-hockey` synchronizes itself to the rate at which the trace files are fed into the packet killers, and it displays the "positions" of the nodes as they "move." This synchronized display allows the human testers to correlate node positions with protocol behaviors. For testing ad hoc networks, where the topology changes are driven by movement, we found having a tool that allowed us to draw the "motions" of the nodes extremely useful.

We do not claim that performance measurements made on the emulated network realized by `macfilter` in any way approximate the measurements of a deployed network, and, in fact, we have verified that results obtained in the lab are not comparable to those collected in the field. Our packet killer and trace generator do not emulate the variability of the outdoor wireless environment, and all the nodes are in the same collision domain, so there are no hidden terminal problems.

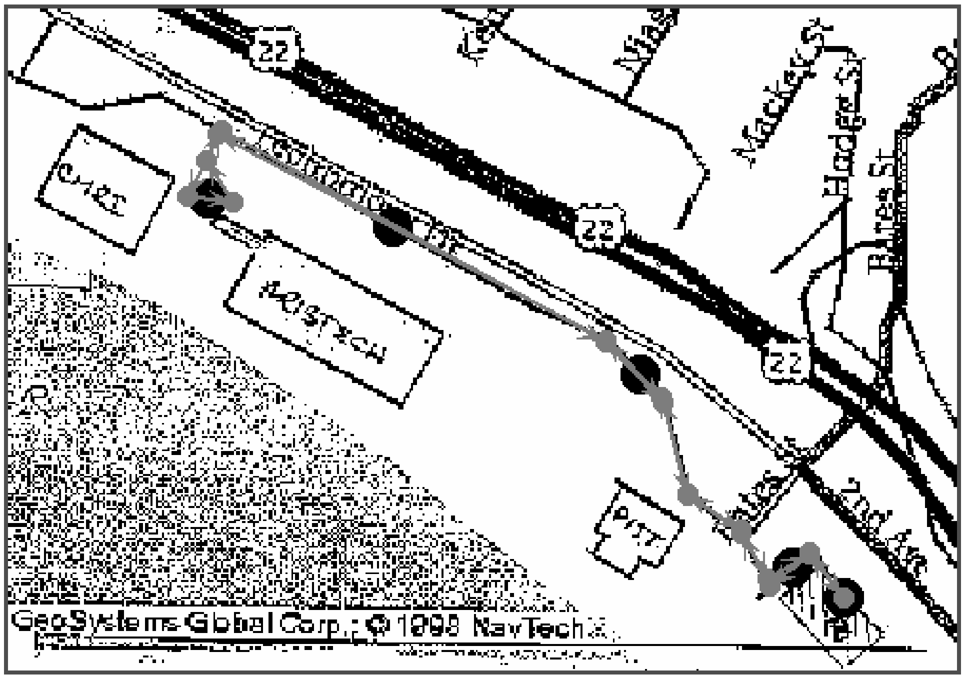


Figure 6.11 The map display of `ad-hockey`, showing the positions and trajectory of the nodes. The trajectory can be changed by click-and-drag on the grey knots.

Nevertheless, the `macfilter` was a critically important tool during the early stages of developing and debugging our ad hoc networking code, as we could exercise all aspects of the protocol from within our lab without having to spend countless hours running around campus to create physical topology change. Using the emulation system, we created scenarios that individually test all the protocol's reactions to topology change, including Route Discovery, Route Maintenance, and the retransmission timers. The `macfilter` also allowed us to perform regression testing on the implementation and find bugs that only appeared after tens of hours of running. Emulation was the only way to find such bugs, since in the physical testbed the nodes were car-mounted and required a human driver to move them and the drivers could not keep going that long.

BBN has developed a system similar to our `macfilter`, but with a more sophisticated packet killer [27]. In the BBN scheme, a centralized server periodically downloads into the packet killer in each node the probability of a successful packet reception from each other node.

Our emulation system of a packet killer controlled by a trace file is similar in concept to the Network Trace Replay system [77], but neither it nor any of the other emulation systems available at the time had the expressive power to emulate a multi-hop ad hoc network. The minimum requirement is that the packet killer be able to express the notion that a node **A** can receive packets from **B** while simultaneously and independently being unable to receive packets from **C**.

6.4. Implementing Doppelganger Emulation Via Direct Emulation

Our direct emulation system is implemented using the emulation code put into *ns-2* by Kevin Fall and the VINT project [29], with a few bug fixes and extensions. Figure 6.12 shows how we use these emulation primitives to create the doppelganger emulation shown in Figure 6.5.

A doppelganger is "a ghostly counterpart of a living person," [109] and this emulation technique works by setting up a simulation where each node in the network to be emulated is represented by a simulation model of the node. Some of these simulation nodes, however, are selected to be the doppelgangers for the physically implemented equipment in the emulation. The emulation system then makes it appear as if the real code running on the application and transport layers of the physical machines is present inside the simulation as the application and transport layers of the doppelganger nodes. Thus, the doppelgangers represent the physical nodes inside the simulation, and they serve as their "ghostly" counterparts.

To use doppelganger emulation, a researcher would first create a scenario that controls the environment inside *ns-2*. This scenario is essentially the same as those used to drive the simulator (Chapter 3): it specifies the number of nodes in the simulation, the node's movement pattern, and the traffic exchanged by the simulated application layer on each node. In addition to these normal features, for each physical node that is part of the emulation, the scenario also specifies the IP address of the node and the simulation node that serves as the doppelganger for that physical node. The location and movement of this doppelganger node inside the simulation determines the "location" and "movement" of the physical node.

The real application code runs on the real machines, shown as nodes **A** and **B** in the example in Figure 6.12. When a packet is generated, as shown at marker **1**, it passes down through the normal network stack of **A**. Each of nodes **A** and **B** has been configured with an IP routing table entry that lists the **Emulation-Server** machine as the node's default router, so any IP packets **A** or **B** attempt to send go to the **Emulation-Server**, and not directly between themselves.

When a packet sent by **A** arrives at the **Emulation-Server** at marker **2**, the **Emulation-Server** uses the Berkeley Packet Filter (BPF) [74] to capture the packet and wrap it into a simulation event. The simulation event representing the packet is then executed by the network layer of **A**'s doppelganger, shown at marker **3**. The event then passes through the simulation models of the network, link, and physical

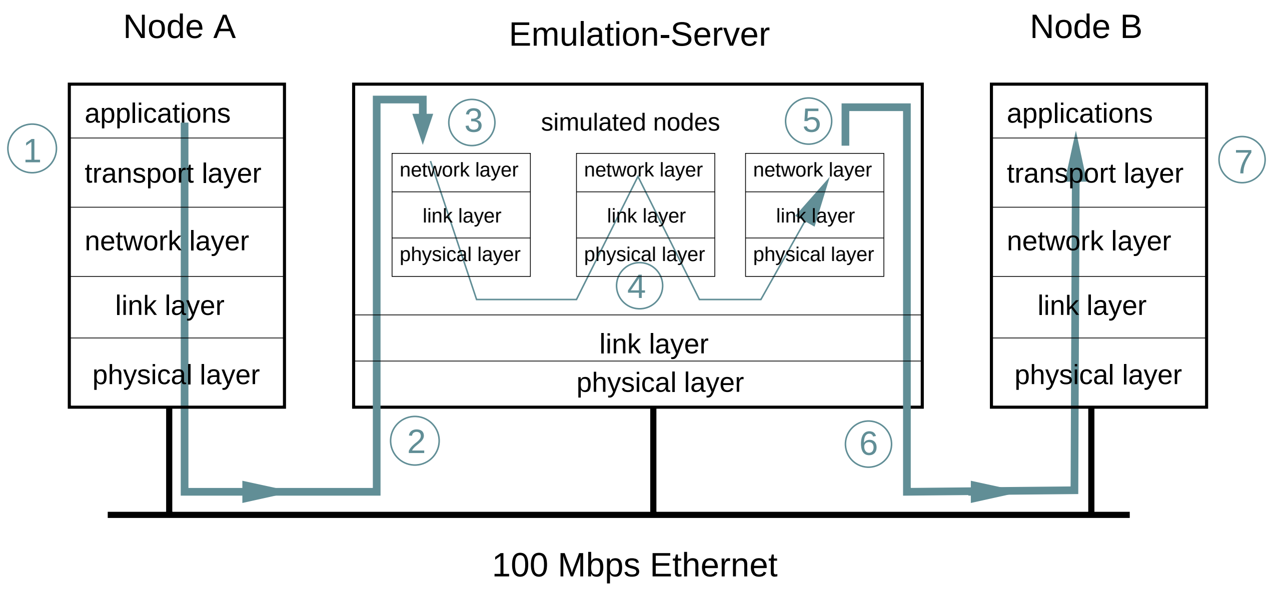


Figure 6.12 Physical implementation of doppelganger direct emulation.

layers, as shown in Figure 6.5. Since the simulation can model the movement of **A**'s doppelganger and the other nodes according to an arbitrary scenario, there is no need for the physical node **A** to move.

The simulation event representing the packet is destined to the application layer of **B**'s doppelganger, so it is not processed above the network layer by the simulation nodes that handle it as it moves through the simulated network, as shown by marker **4**. Although these simulation nodes do have application and transport layers that exchange packets with each other, in a doppelganger emulation those packets are internal to the simulation. This forms another strength of doppelganger emulation: at no point does a physically implemented layer directly peer with a simulation model of that layer. Since the the situation shown in Figure 6.6 does not arise, a researcher need not build complete application or transport models inside the simulation.

The simulation of the network is running in real time, so the processing and queueing delays dictated by the simulation models are converted into the actual passage of time on the wall clock. For example, if a link layer model simulates queueing a packet for 100 ms, the processing of the packet is actually delayed for 100 ms of wall clock time. When the simulation event carrying the real packet finally arrives at the network layer of the simulation node representing its destination, as shown by marker **5**, the packet is removed from the event and injected back on the physical Ethernet via a raw IP socket at marker **6**. When the packet arrives at the physical node **B**, it is processed as normal by the network stack and eventually handed to the application (marker **7**). If the simulation had determined that the simulated packet would be dropped instead of reaching marker **5**, the simulation event representing it is discarded, and the real packet it includes is also dropped.

A paper by Ke, Maltz, and Johnson [62] gives more information about our implementation of direct emulation and explains optimizations to the simulator that improve its performance. The keys to minimizing the emulation artifacts in direct emulation are ensuring that the simulation clock keeps up with the real-time wall clock, and minimizing the delays for the transfer of packets between markers **1** and **3** and markers **5** and **7**. If the simulation falls behind real-time, packets between **A** and **B** will be delayed by more than they should be. Likewise, in a real system, no time is lost while the packets are transferred from the machine that created them to an emulation-server and back. Any delay added by this process is an emulation artifact. Thankfully, when emulating wireless networks where the link bandwidth is typically 10 Mbps or less, 100 Mbps Ethernet operates fast enough that the time lost is proportionately small, and the emulation artifacts from this source are insignificant.

An advantage of doppelganger emulation is that *almost any* type of device can be placed into an ad hoc network environment. The only requirements on the device are that it communicate via IP packets and that its routing can be configured to direct outgoing packets to the emulation-server machine. There is no need to have access to the internals of the device's network stack, or the ability to modify its code. For example, consider a handheld device, such as a map viewer designed for allowing a soldier in a battlefield setting to access maps and situational awareness data. The device could be connected the **Emulation-Server** machine via a wired or wireless connection, and then doppelganger emulation could be used to evaluate how useful the device is when its maps and data must be fetched across an ad hoc network in a battlefield scenario. The device need not have even been designed with the intention of using it in an ad hoc network.

6.5. Validation of Direct Emulation

Before using emulation to conduct experiments on the performance of their systems, researchers must have confidence that the results they receive under emulation will match the results in the real world. The presence of too many emulation artifacts would reduce the fidelity of the emulation to the point it is not useful or is even misleading. In this section, I explain how we have validated the behavior of the direct emulation system against the behavior of the simulator.

We do not consider here the final step in the validation of emulation, namely verifying that the simulator itself accurately models the real world [50]. Verifying the simulator is an independent area of active research, and so, for now, we content ourselves to measure how well we can reproduce the simulation results and eliminate emulation artifacts. By transitivity, however, if we show that the emulation system matches the simulation, and it is shown that the simulator matches reality, we will have great faith that the emulation system matches reality. This is actually an example of another general strength of direct emulation: it immediately leverages all the independent work being done to improve and verify the simulator.

Our basic method for validating the direct emulation system involves the following steps. While we focus on validating doppelganger direct emulation, the same techniques can be used to validate the other forms of emulation described earlier.

1. Identify applications and transport layers for which there are both physical implementations and accurate simulation models. An accurate simulation model is one that is known to properly and correctly model the behavior of the application and transport layer, due to either analytic or experimental validation. Obtaining these simulation models is outside the scope of this thesis, although such models are available for a variety of traffic sources and transport layers (e.g., FTP or HTTP over TCP and CBR over UDP).

2. Create validation scenarios that are representative of the types of experimental scenarios to which the researcher wishes to apply direct emulation. In order to be representative, the validation scenarios should have a mix of nodes, traffic types and traffic intensities similar to the experimental scenarios.

3. For each validation scenario:

   (a) Run a simulation of the validation scenario using the *simulation model of the application* as the test workload source and sink.

(b) Run a direct emulation realization of the same scenario, but using the *physical implementation of the application* as the test workload source and sink.

(c) Compare the performance of the application in the simulation with its performance in the emulation. Since the scenarios and the applications are the same and the simulation models are assumed to be accurate, any differences between the two runs are due to emulation artifacts or inter-run variation in the workload offered by the application.

6.5.1. Qualitative Validation

As an example validation, Figure 6.13 qualitatively compares the performance of a TCP transfer of 30 MB between two nodes in a 10-node scenario. One trace shows the time-sequence graph of the TCP connection when the entire network is simulated, and the other trace shows the performance of the TCP connection when it is implemented by two physical nodes and the rest of the network is realized by direct emulation. The emulation-server was a 450 MHz Pentium II CPU, and, in both cases, two additional TCP transfers were made as background traffic between simulated nodes. As the nodes in the scenario move, the number of hops through which the packets must travel smoothly varies between 1 to 4, which causes a smooth variation in the bandwidth of the TCP connection. The closeness of the traces demonstrates direct emulation's general ability to mimic the environment of the ad hoc network scenario for the physical nodes.

6.5.2. Quantitative Validation

While experiments, such as that described in Figure 6.13, can be used to verify a particular run of direct emulation, it provides to the users of direct emulation little guidance on which scenarios can be safely used

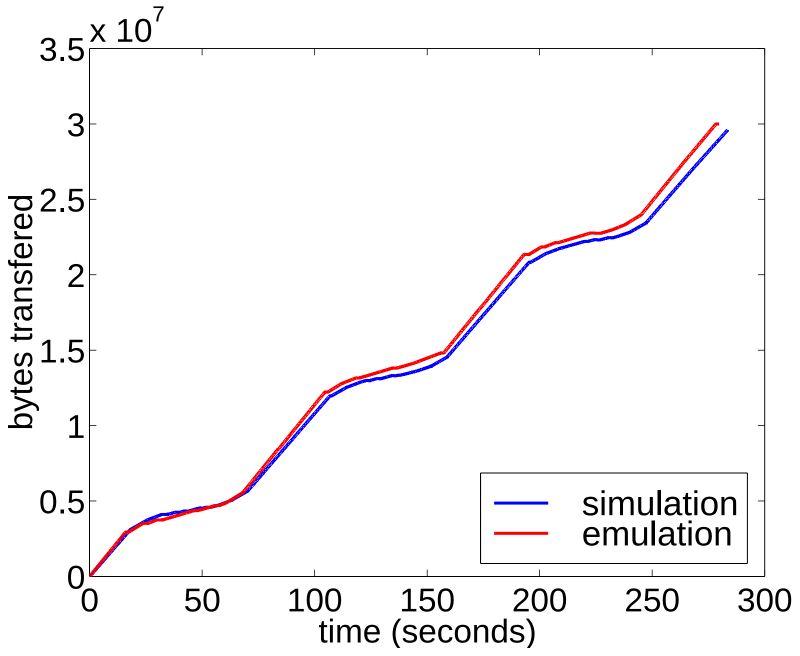


Figure 6.13 Time-sequence plot comparing a simulated TCP transfer with a physical TCP transfer using Direct Emulation to realize the simulated scenario (10 node scenario).

with the direct emulation system. What users need to know are the parameters of the class of scenarios that can be run on the emulation-server without causing significant emulation artifacts.

Based on observation of the emulation-server's performance, we hypothesize that it is most sensitive to two parameters of a scenario: the *number of nodes* in the scenario and the *number of events per second* that must be processed by the simulator. Since the number of events generated by any particular action is not easily estimated by a researcher not intimately familiar with the simulator, we chose to use the number of packets sent per second as a surrogate measure.

We created the template validation scenario shown in Figure 6.14 to enable us to independently vary the two parameters. The validation scenario consists of three groups of nodes, separated by enough distance that the radio transmissions in one group in no way effect the transmissions in the other groups. This is important, as it means that media contention has no effect on the bandwidth measurements we will describe next. Two of the groups consist of two nodes each, with the two nodes acting as end points for a single TCP connection each. The third group contains the all remaining nodes, which are arranged around a central node. This group provides the ability to vary the computational load placed on the emulation-server by varying the number of packets per second this central node sends.

When the validation scenario is run as a pure simulation, with the end points of the two TCP connections implemented by simulation models, both connections achieve a throughput of 1.44 Mbps — regardless of the number of nodes in the third group or the number of packets per second the nodes in that group send. Although the raw bandwidth of the wireless link in the scenario is 2 Mbps, the overhead of the IP and TCP headers and the 802.11 MAC protocol reduce the achievable throughput to 1.44 Mbps. That the achievable throughput is independent of the number of nodes and the rate the nodes send packets is expected, since in pure simulation there is no need for the simulator to maintain real-time performance.

When the same validation scenario is realized using direct emulation, with the two TCP connections implemented by FTP transfers between two pairs of physical machines, the TCP connections achieve a throughput that depends on the number of nodes in the scenario and the number of packets per second those

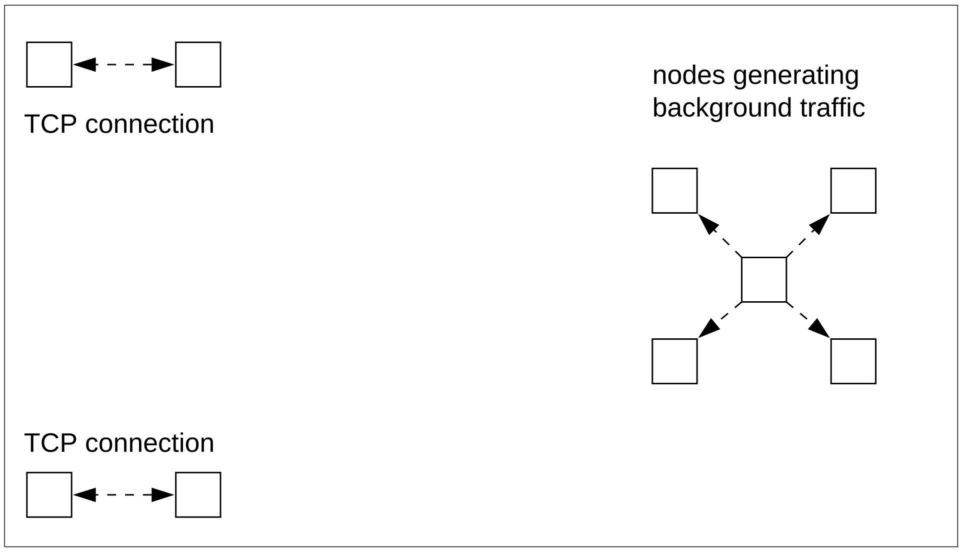
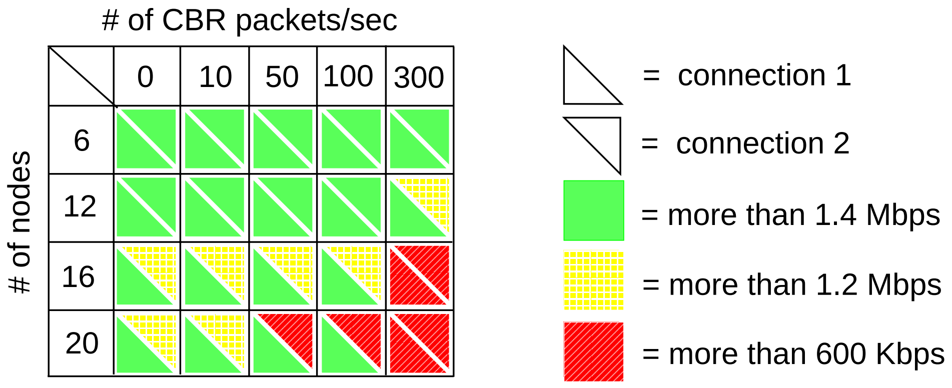


Figure 6.14 The template for scenarios used to validate direct emulation. Each group of nodes is located far enough away from the other groups that the transmission of a packet within one group does not interfere with packet transmission in any other group.



# of CBR packets/sec
# of nodes
= connection 1
= connection 2
= more than 1.4 Mbps
= more than 1.2 Mbps
= more than 600 Kbps

# of nodes	0	10	50	100	300
6	🟩🟩	🟩🟩	🟩🟩	🟩🟩	🟩🟩
12	🟩🟩	🟩🟩	🟩🟩	🟩🟩	🟩🟨
16	🟩🟨	🟩🟨	🟩🟨	🟩🟨	🟥🟥
20	🟩🟨	🟩🟨	🟩🟥	🟩🟥	🟥🟥

Figure 6.15 Table showing the average throughput obtained by two emulated connections as a function of scenario complexity. In the absence of any emulation artifacts, throughput should be 1.44 Mbps.

nodes send. We ran the experiment with scenarios having a total of either 6, 12, 16, or 20 nodes. We independently varied the level of background traffic in the scenario, with the central node in the third group of nodes sending either 0, 10, 50, 100, or 300 packets per second.

Figure 6.15 displays the results of these validation experiments. The results have the expected pattern: as the number of nodes and the level of background traffic increases, the quantity and magnitude of emulation artifacts increase. The emulation-server used in this experiment was a 450 MHz Pentium II CPU with a single 100 Mbps Ethernet card, and it appears to support 12-node scenarios with no significant artifacts. As the number of nodes increases beyond 12, the level of background traffic that the emulation-server can support decreases.

Researchers wishing to use direct emulation to analyze their own applications can rerun similar experiments to evaluate the hardware they use for their emulation-server. Using the results of this evaluation, they can create scenarios that will be unlikely to be affected by artifacts.

6.5.3. Analysis of Simulator Time-Lag during Emulation

In order to explain the performance of the direct emulation system, we analyzed how far behind real-time performance the simulation slipped during the emulation run. We define the *time-lag* of a simulation event to be the difference between the wall clock time at which the event was scheduled to occur at and the wall clock time at which the event was eventually executed. Figure 6.16 plots the time-lags as a histogram for the scenario with 16 nodes and 100 CBR packets/sec sent as background traffic. The figure is on a semi-log scale for clarity, and only events with a time-lag greater than 1 ms are shown. For this emulation, there were 1.64×10^7 events with time-lag greater than 1 ms. The rightmost bar collects all events with a time-lag greater than 256 ms.

Figure 6.17 shows the time-lag histograms for a series of scenarios with a mix of number of nodes and background traffic. The histograms follow the same pattern as Figure 6.15: moving to the right and down, the number and magnitude of time-lagged events increases.

The apparent correlation between the number and magnitude of time-lagged events with the presence of emulation artifacts suggests it would be fruitful future work to quantify the correlation. Should the correlation hold, it should be possible to run arbitrary scenarios on the emulation-server, and determine after

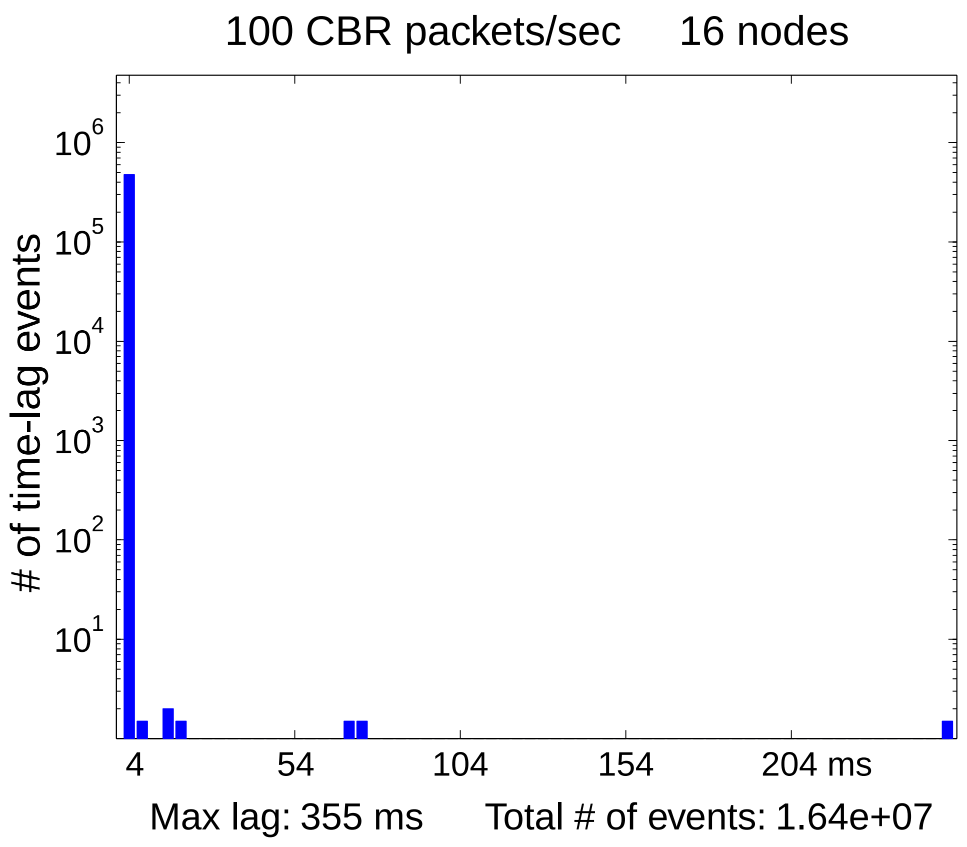


Figure 6.16 Histogram of time-lags from a direct emulation recreation of a scenario with 16-nodes and 100 packets per second sent as background traffic. The X-axis depicts the range of event time-lags, and the Y-axis shows the log of the number of events with a given time-lag.

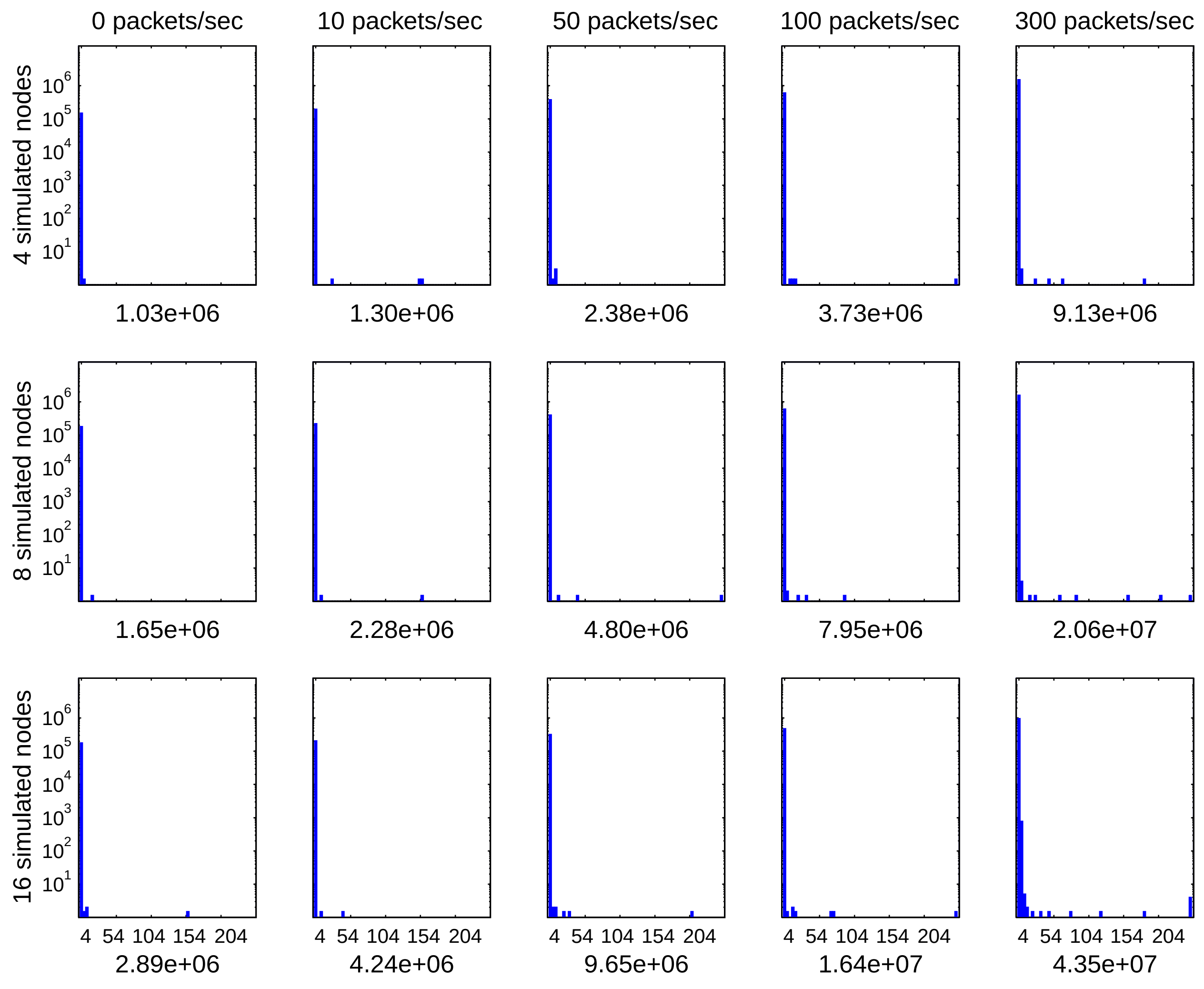


Figure 6.17 Time-lag histograms for some of the scenarios shown in Figure 6.15. The number under each plot is the number of events with a time-lag more than 1 ms.

a run completes whether the performance results collected during the run are valid or should be discarded as having been contaminated by emulation artifacts.

6.5.4. Sources of Emulation Artifacts

Beyond time-lag, another potential source of difference in the results between direct emulation and simulation is the TCP model used in the simulator. Our simulator (based on *ns-2*) used the TCP-Reno source with the Del-Ack sink, which generates behavior very similar to the canonical concept of TCP in bulk-throughput mode. However, there are certainly behavioral differences between the actual implementation of TCP on the physical machines and the simulator models, starting with the fact that the simulator model does not include the SYN-ACK or FIN-ACK exchanges that begin and end the TCP connection used for bulk-throughput.

Unfortunately, quantifying the results of these differences must wait for future work. However, when we varied the TCP model used in the simulator from TCP-Reno to TCP and the sink from Del-Ack to one that acknowledged every segment, we did *not* see significant changes in the comparison results. This suggests that, while differences between the simulation model and the actual implementation are responsible for some of the differences in validation experiments, it is not a first order effect.

6.6. Characterization of Coda Performance Over an Ad Hoc Network

As an example of how the ability to accurately emulate ad hoc networks can be useful for application designers, we worked with the members of the Coda project at Carnegie Mellon University to create a demonstration of how well the Coda distributed file system [12, 101] works in an ad hoc network environment [72]. In addition to allowing the Coda researchers to demonstrate how valuable Coda's features are in a network environment that is given to periods of partitioning and weak connectivity, the experiment gave us an opportunity to determine how useful emulation is as a tool.

6.6.1. Coda Overview

Coda is a client-server distributed file system based on the observation that mobile clients will experience many different degrees of connectivity with the server. At times, the client will be *strongly connected* to the server over a high-speed high-reliability network connection. At other times, the client will be *weakly connected* to the server, over slow, expensive, or lossy connections, such as modem lines. Finally, sometimes the client will be completely *disconnected* from the server, and it will need the ability to operate independently.

Unlike previous distributed file systems, Coda was designed to be aware of the changing connectivity of its clients. It maximizes the usefulness of the client by *hoarding* the files the user is most likely to need on the client machine, taking advantage of periods of connectivity between clients and servers to update the hoard. It supports a *weak consistency* model, so that changes can be made to files stored in the hoard during periods of disconnection, and *trickle integration* to allow these changes to be propagated back to the server whenever possible. Coda also supports replication between file servers, so that if one server is unreachable, data can be retrieved from any other server with a replica of the data.

Coda is particularly well suited as a test for the usability of emulation, in that it consists of both application code and a transport protocol. The system consists of more than 100,000 lines of code that have been implemented and tuned over 10 years, with some sections of it implemented in the kernel. Attempts have been made before to create simulation models of the system, but none have modeled the full complexity of the system, or the subtle interactions that occur in the actual implementation. Finally, the Coda researchers are the perfect intended users of direct emulation in that the Coda system is complicated enough that only the Coda researchers have the expertise to evaluate their system, but they otherwise lack the ability to realize the ad hoc network environments in which they wish to experiment with their system.

6.6.2. Coda Over an Ad Hoc Network

The goal of the Coda researchers in this demonstration was to use emulation to show how Coda's features would be useful in a natural disaster response, such as after an earthquake or fire. Figure 6.18 shows how four laptops were connected to the emulation-server. Each laptop was setup just as it would be in an actual deployed system. Two laptops were configured as file servers, and the other two laptops were configured as clients, such as would be carried by diaster response teams.

A total of 12 nodes participated in the disaster scenario designed by the coda team: 4 as doppelgangers for the 4 physical laptops, and 8 as simulated nodes representing other disaster response units. DSR was used as the routing protocol to connect the ad hoc network together, as the nodes were often multiple wireless hops away from each other. The scenario was intentionally designed to contain periods of disconnected operation, weak connectivity, and strong connectivity. As the simulation nodes moved through the scenario, humans using the laptops to complete assigned tasks experienced what the applications would "feel" like if all the nodes were physically implemented and moving in the real world.

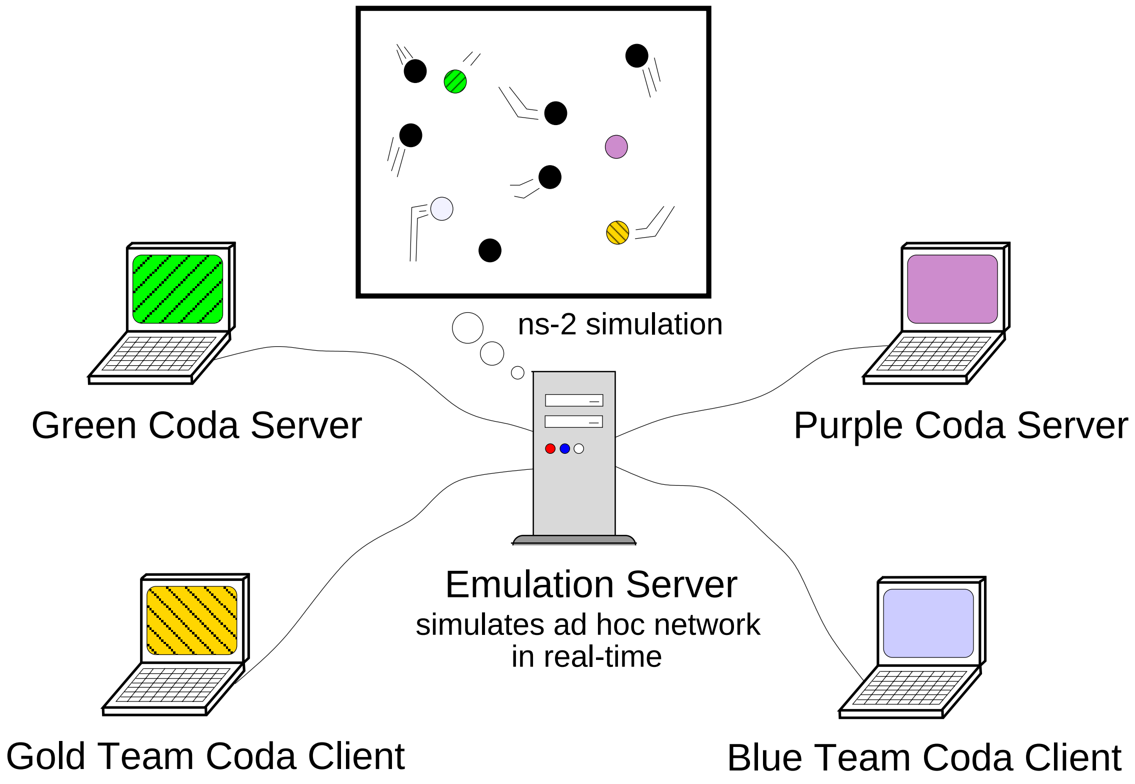


Figure 6.18 Configuration of the machines for experiments running Coda over a DSR ad hoc network.

6.6.3. Results of the Coda Experiment

We feel the greatest success of emulation was actually achieved while the Coda team was preparing their demonstration. In the process of running different scenarios through the emulation-server, the Coda transport protocols were stressed in ways typical of ad hoc networks, but which the Coda team had not attempted before. Studying the traces from these emulation runs led to the discoveries of weaknesses in the protocols that had not previously been exposed. The researchers report that the ability to repeatedly run exactly the same scenario was critical in tracking down the flaws and fixing them.

Ke, Maltz, and Johnson [62] have provided a more complete description of the issues found by the Coda team through their use of our direct emulation system. However, the simplest proof of the utility of the emulation system is shown by the performance improvement achieved in the Coda transport protocol.

Figure 6.19 compares a time-sequence plot of the Coda transport protocol as it was first run over an ad hoc network with a time-sequence plot of the transport protocol on the same emulation scenario after its weaknesses had been corrected. After correcting the weaknesses discovered through the use of direct emulation, the throughput of the transport protocol has roughly doubled. It is impossible to estimate how much time and effort would have been required to discover and repair these weaknesses if the only means for experimenting with an ad hoc network environment was to physically move nodes in the field.

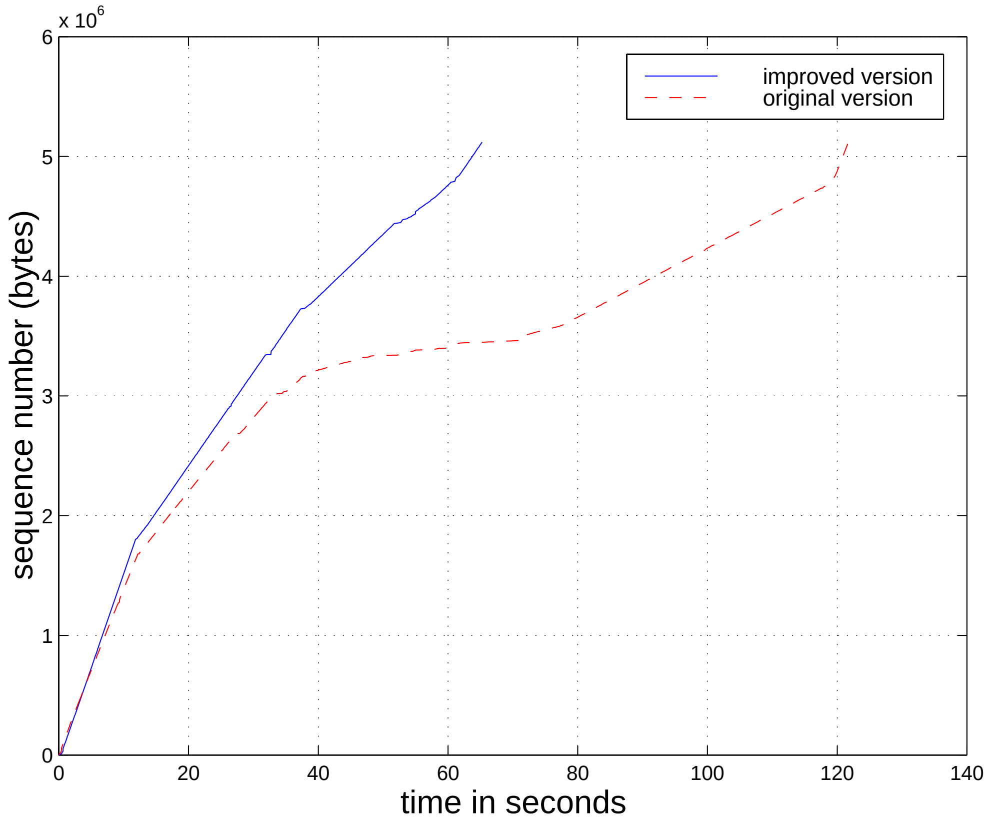


Figure 6.19 Comparison of Coda's transport protocol performance in an ad hoc network environment before and after use of the emulation system to test it and improve it (Figure taken from Ke *et al.* [62]).

6.7. Comparison of Direct Emulation and Trace Modulation

As described above, direct emulation requires that the simulator or simulation model must be able to maintain *on-line* real-time performance, or its slow-down will introduce emulation artifacts in the behavior of the system under test. Maintaining that real-time performance is difficult, since the simulation model of the network is a complex and detailed system. Network Trace Replay system (NTR) [77] is a trace modulation method of emulation that uses a *off-line* simulation to drive the realization of an emulated scenario. Although the current state of the NTR system does not allow it to represent a general multi-node, multi-hop ad hoc network, it does offer an opportunity to experiment with sophisticated trace modulation, which has potentially better scaling properties than direct emulation.

6.7.1. Adapting the NTR Trace Modulation System to Ad Hoc Networks

The Network Trace Replay system consists of distillation software, a special traffic workload, and a packet killer. The distillation software is an off-line analysis program designed to take a time-stamped packet trace of traffic flowing through a network and extract the characteristics of the logical communications channel between the source and destination of the packets. The special traffic workload has been designed to simplify the analysis the distillation software must perform, and it includes the exchange of a sequence of packet-pairs. The packet killer is used to modulate the logical channel between two physical nodes to match the conditions that existed between the source and destination when the packet trace was recorded.

In an earlier application of Network Trace Replay [77], a single mobile node was physically carried through a course while exchanging the special traffic workload with a server. The time-stamped trace of the packets exchanged between the mobile node and the server was then analyzed by the distillation software to create a trace file. The resulting trace file described the latency, loss-probability, and bit-rate of the logical channel between the mobile and the server. Since Network Trace Reply operates by condensing a network's behavior into a trace file, we classify it as another form of trace modulation (albeit significantly more sophisticated than that described in Section 6.3).

The current version of Network Trace Replay is limited to studying the traffic between two nodes. However, it can still be used to emulate a multi-hop ad hoc network with enough fidelity to permit meaningful performance measurement of the resulting system. First, we create a scenario file describing the motion and background traffic of the nodes in the network. We then identify the two simulation nodes that will represent the mobile node and the server, and we augment the scenario file with additional traffic sources that create the special NTR workload between the two nodes. We then run a simulation of the scenario. Finally, we post-process the resulting simulation output trace with the NTR distillation tools. This extracts into a trace file the salient characteristics of the logical channel between the server and mobile node.

Using the standard NTR packet killers to replay the trace file, we can give two physical nodes a logical communications channel that behaves in roughly the same fashion as it would if the nodes were actually moving about in the network described by the scenario file. The technique is advantageous because, unlike direct emulation, there is no longer any limitation on the complexity of the scenarios that can be emulated. As the scenario becomes more complicated, the simulation and post-processing steps simply become slower. These steps are performed *off-line*, however, and do not affect the performance of the packet killers, which are the only components of the system that must run on-line in real-time. Said another way, as much time as desired can be spent producing a trace file that describes the logical channel as accurately as possible, without introducing any emulation artifacts.

Trace modulation does suffer from a significant drawback, however. The technique is based on the assumption that a special workload, which is not the actual workload the mobile node and server would have applied to a real network, is capable of accurately exposing all the important characteristics of the communication channel between the nodes. For the assumption to be valid, it must be the case that the actual traffic would not perturb the state of the network away from that which existed when the NTR workload was simulated. Because the trace file is pre-generated, there is no way for trace modulation to react to the traffic actually offered by the nodes when replaying the trace.

As an example of how this drawback could be a problem in practice, the actual bandwidth required by the NTR workload is quite small and is unlikely to cause congestion in the network. If the actual workload is bandwidth-intensive, it could be the case that a real network would become congested and drop packets, although the trace file describes a communication channel with no packet loss. Direct emulation does not suffer from this problem since, as a result of directly simulating the additional packets, the simulated network would become congested and drop packets as appropriate.

Due to code limitations, the current NTR system is restricted to modulating the channel between only two nodes: it cannot independently modulate the channels between three or more nodes. However, even if the code were extended to handle the modulation of multiple channels, interesting future work would be required to determine if the NTR workload is able to characterize the network behavior that results when there is *interaction* between the traffic offered by two doppelgangers.

Although these are significant limitations, the area of trace modulation holds promise as an area for future research, given its ability to emulate large networks. In fact, as the network becomes larger and more complicated, trace modulation is likely to work even better. As the actual traffic offered by the physical nodes becomes a smaller and smaller fraction of the overall network traffic, it is even less likely to push the network in a different region of operation than the one measured by the NTR workload during simulation.

6.7.2. Comparison Experiment

As a first attempt to understand the scaling differences between direct emulation and trace modulation, we conducted the series of experiments outlined by Figure 6.20. We first created a scenario file that describes a network with 16 nodes, a light loading of TCP and CBR background traffic, and a 30 MB FTP transfer between two nodes in the network. This scenario file was then used to drive a conventional simulation, a trace modulation replay, and a direct emulation of the network. For the emulation methods, two physical machines acted as the end points of the 30 MB FTP transfer. Trace data was collected and used to draw the TCP time-sequence number plots shown in Figure 6.21. Since we are accepting the results of the simulator as a control, the design of the experiment is such that the resulting three time-sequence plots should be identical, with any differences due to emulation artifacts in the respective system.

As shown in the middle portion of Figure 6.20, simulating the 16 nodes and traffic mix present in the 470-second scenario took 300 wall clock seconds: the machine used in this example was a 300 MHz Pentium-II machine. Running the same movement scenario and background traffic, but with the NTR workload taking the place of the FTP transfer, took 130 wall clock seconds (left-hand portion of the figure). The result of this simulation was a 600 KB collection trace file, which when analyzed by the distiller reduced to a single 15 KB trace file. Although, in this case, the production of the trace file took even less time than the simulation, a strength of the trace modulation methodology is that this process must only be performed once and is performed off-line. The wall clock running time of the trace file generation process is therefore immaterial, and as much time as needed can be spent on the distillation process in order to get a more accurate trace file. Only the realization happens in real time, and the simple packet killer that is part of the trace modulation replay module can easily maintain real-time performance while following the commands in the trace file. Finally, as shown on the right portion of the figure, no set-up steps are required for direct emulation. The scenario file is fed directly into the emulation-server for realization of the ad hoc network environment described by the scenario.

The results of this comparison between direct emulation, trace modulation, and simulation appear in Figure 6.21. Both emulation technologies track the simulation performance fairly well, with the RMS error over the length of the run was 0.69 MB for direct emulation and 0.77 MB for trace modulation. During the

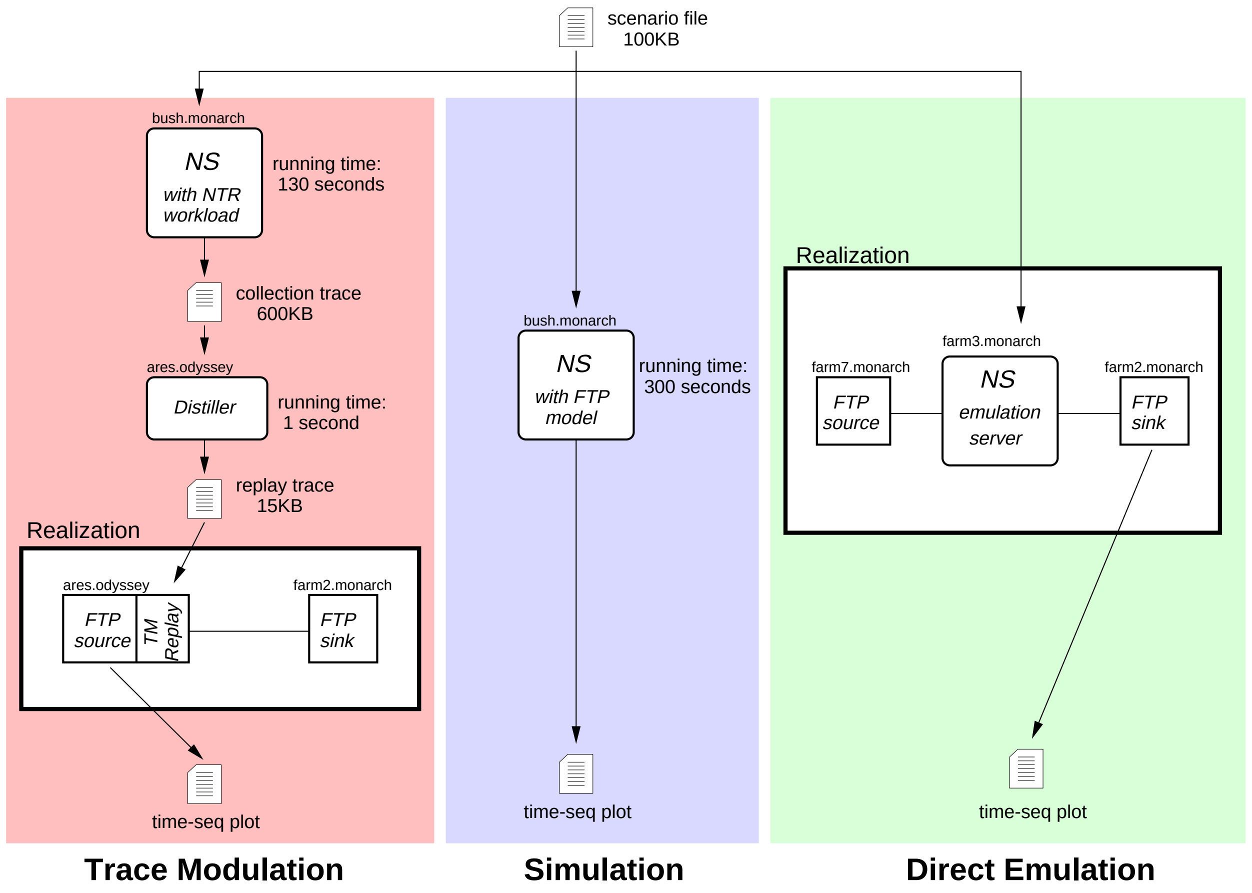


Figure 6.20 Outline of experimental method for comparing the performance of trace modulation with direct emulation.

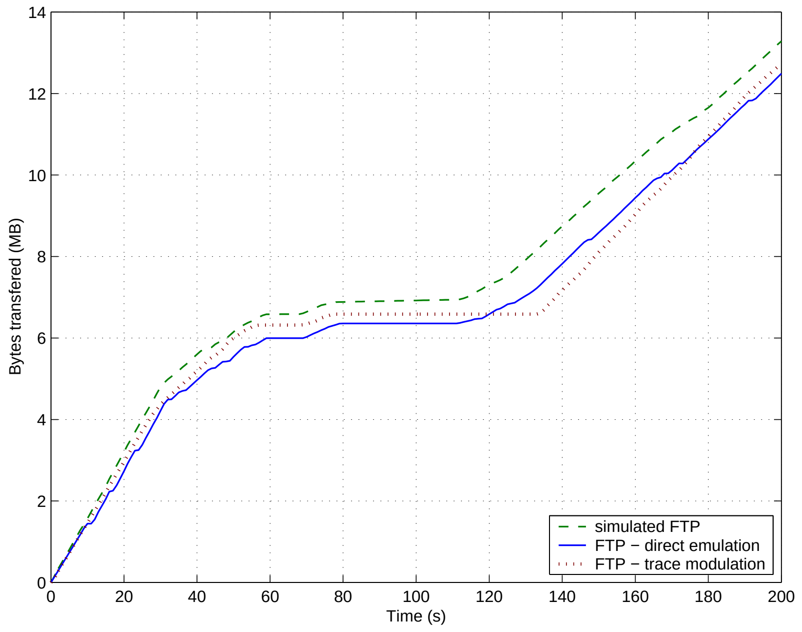


Figure 6.21 Time-sequence number plots for a TCP connection between two nodes in a 16-node ad-hoc network with TCP and CBR cross-traffic. Each line shows the progress achieved by the same connection under the same movement and traffic pattern, but compares the progress of the connection under simulation with the direct emulation and trace modulation realization of the same scenario.

period from 20 s to 60 s, the graph shows regions where the emulation-server could not keep up with the real-time requirement of carrying the TCP traffic while simulating the scenario, and the slope of the direct emulation curve shallows as a result to a slope less than that of the simulation curve. In contrast, the period from 120 s to 140 s points out a risk of trace modulation. Due to smoothing errors, the NTR distillation tool did not correctly locate the end of the network partition that lasted from 76 s to 110 s, and it therefore generated a trace file that erroneously prevents any progress until 135 s.

6.8. Future Directions for Validating Emulations

We believe that, at their core, the best methods for validating the behavior of emulation and simulation systems all involve comparing performance data measured from a real deployed network with performance data collected from an emulation or simulation of that same network. The critical components in such a methodology are the ability to create scenarios that fully exercise the protocols and the function used to compare the performance results [50].

We divide the types of comparison that can be made between sets of collected performance data into two different classes. Inside each class, we believe the comparisons should be made at different time scales for different applications. For example, the performance of a simulated network can be compared to a real network by considering the quantity and timing of all packets sent by nodes in the network, including routing protocol packets not visible to a network user — we call this an *internal-behavior comparison.* In contrast, end-users may care only about the performance they see for the traffic they offer to the network, which could be measured by an *external-behavior comparison* that focuses on end-to-end parameters such as latency, bandwidth, and packet loss rate. Qualitative external-behavior comparisons may also be useful, such as whether the words in an audio stream are of sufficient quality to be intelligible.

Similarly, the time scale over which comparisons are made may need to be varied depending on the use of the network. For some protocols and applications (e.g., a background file transfer) only the average throughput they receive over some long time period is important. For other protocols and applications (e.g., voice applications) the timing and drop probability of individual packets is important.

6.8.1. Determining the Fidelity of an Emulation

Figure 6.21, described in Section 6.7.2, is an example of the form of input on which the comparison function operates. In order to evaluate how closely the two types of emulation come to reproducing the simulation results, we must be able to quantitatively answer "how similar are these curves?" To a human, the curves all clearly have the same shape, however, the differences are interesting. The shape of the direct emulation curve tracks the the simulation curve more tightly than the trace modulation curve. In particular, when measured over intervals of 10 s, the slope of the simulation results and the direct emulation are substantially more similar than the simulation results and the trace modulation results.

For now we treat only external-behavior comparisons, though these techniques may be suitable for internal-behavior comparisons as well. We assume that some specific test workload of interest can be applied to both of the systems to be compared. While the workload is being applied, we assume the movement pattern of the nodes in both systems is identical. This means that the motions of the nodes cannot change based on the receipt or non-receipt of packets from the test workload. As an example, commands to change direction cannot be sent to the mobile node over the network, since the motion of the mobile node will be different depending on when they are received.

We propose the following algorithm for measuring the similarity between the performance of a simulation and a real-world experiment or between the performance of a simulation and an emulation.

1. Choose an external-behavior metric of interest on which to base the comparison.

2. Begin applying the test workload to the systems when the nodes are in identical positions.

3. Compute the average slope of the metric-versus-time curve over intervals of ΔT, with ΔT chosen to represent the time scale of interest.

4. Compute the RMS error of the slopes.

An advantage of this algorithm is that the RMS slope-error can be used over whatever time scale the modeler believes is of interest. Low-level modelers will have to demonstrate similarity with small ΔT, since they claim to be modeling the fate of each packet. Higher-level modelers only have to show similarity for larger ΔT. Likewise, an appropriate value of ΔT can be used for whatever application mix the real or simulated network will be carrying.

6.8.2. Using Trace Distillation as a Comparison Function

Another promising methodology for validating emulation systems derives from our work with trace modulation. To the extent that the NTR workload and distillation process is able to extract a characterization of salient network parameters from a run of the network, it can be used to characterize how well the emulation system matches the behavior of simulated or real networks.

In this methodology, the NTR workload is run on a real network and separately run on a simulated or emulated network with a scenario file designed to match the motions of the real network. The packet traces of the network runs are each run through the NTR distillation process, and the resulting trace files then compared. Differences indicate where the simulation or the emulation failed to model the behavior of the real world in a significant fashion.

6.8.3. How Much Fidelity is Enough?

Finally, it would be useful to provide simulator developers with guidance on how much effort should be put into writing extremely detailed simulations. At this time, it is not really known how sensitive applications are to the environments realized by simulation or emulation. If coarse simulations and emulations of an ad hoc network environment are sufficient to allow applications to behave the same way they do in real networks, then the effort to develop extremely detailed simulations and emulations can be avoided.

An answer to this question may come from adopting an information theory standpoint and asking, "how many bits of information are required to represent the topology of an ad hoc network as it changes over time?" For the n nodes in a network, imagine constructing the full mesh of n^2 logical channel models to connect each pair of nodes. If the behavior of the network is very simple and unchanging, then the trace files driving the logical channel models will all be quite short. The more complicated and time-varying the behavior of the network, however, the longer the trace files will have to be. In fact, the number of bits in the trace files can be used to estimate the entropy of the network, or said another way, the complexity of the topology changes.

To understand the relationship between entropy and trace file size, consider the following example. Given a trace of a real execution run, it is possible to create trace files that will cause a trace modulation system to exactly reproduce the performance of the real system on that particular workload, which will result in an RMS error of 0 when the performance of the real run is compared to the performance of the emulation run. Such trace files would be built by explicitly listing, individually, the fate of every packet sent, and so they would require a large number of bits. As the trace files are distilled down to extract the average performance of the network over longer periods of time, we expect the size of the trace files to shrink and the RMS error between actual and emulated performance to increase. By varying the amount we distill the traces of real executions and then plotting the RMS error versus the size of the trace files for a range of scenarios and a range of distillation levels, we can learn how sensitive the performance of the applications are to the fidelity of the emulation system.

Knowing the shape of the RMS-Error versus bits-in-trace-files graph will be useful to all simulation and emulation developers, as it shows the payoff they should expect for the effort they make to create detailed simulations or emulations of real systems. This information will be particularly important if the graph has break-points or knees. For example, if the graph has the general shape of Figure 6.22(a), developers would know that additional work to increase the detail of their simulations will be repaid with additional fidelity, since more detailed simulations are equivalent to larger trace files. If the graph appears as in Figure 6.22(b), however, only extremely detailed simulations have any value. Finally, if the graph appears as in Figure 6.22(c), then most of the achievable fidelity can be obtained with fairly simple models of the network, and work to create extremely detailed models is probably wasted.

6.9. Chapter Summary

This chapter has validated that direct emulation works, and that it works for interesting scenarios. The chapter explained how direct emulation can be used to test applications on top of any scenario that *ns-2* can support while maintaining real-time performance. The chapter also demonstrated a methodology for finding the parameters of the scenarios that can be used safely used with a particular emulation-server machine. For example, a 400 MHz Pentium II can easily handle useful scenarios with 16 nodes and a total of 100 packets being sent per second.

Emulating ad hoc networks has tremendous value, since it provides an easy way to experiment with protocols and applications without the hassle of a physical testbed. Ultimately, however, physical implementations are required in order to validate both emulation and simulation methods. This chapter points out several directions that could be pursued to conduct this validation, but the final closing of the loop awaits future work.

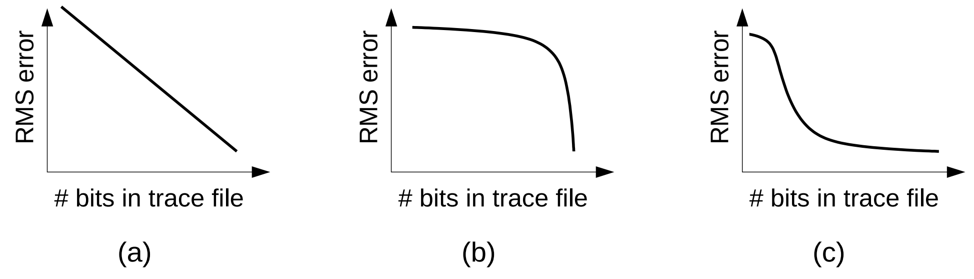


Figure 6.22 Examples of possible shapes for the graph of RMS Error versus the number of bits in the trace files representing an ad hoc network.

Chapter 7

Implementation of a DSR Ad Hoc Network

Ultimately, the final test for any protocol is to deploy it in one of the environments for which it was designed and measure it to determine if it operated successfully. During the 7 months from August 1998 to February 1999, a team of 12 people, largely under the direction of Josh Broch and myself, designed and implemented a full-scale, multi-hop ad hoc network testbed [71] to enable the performance evaluation of ad hoc networks in the field. From February through April, the testbed was used to demonstrate the potential of ad hoc networking to our group's sponsors and as a research tool to experiment with the carrying capacity and behavior of a fully-deployed network.

Routing in the testbed is performed by DSR, extended with additional features to allow the entire testbed to be seamlessly integrated with existing Internet infrastructure. The team implemented an extensive set of network monitoring tools, which track the precise location and protocol processing activities at each node and allow us to analyze the behavior of the network. A series of traffic generators were also developed to stress the network with a variety of different traffic loads.

This chapter describes the results of our initial experiments on the testbed, and discusses the considerable effect that real-world radio propagation had on the protocols in the network. Additional information on the lessons we learned, the tools we built, and the structure of the DSR implementation can be found in a technical report describing the testbed [71].

7.1. Testbed Overview

Our primary design goal for the testbed was to challenge the network protocols to the point where they were stressed, and so we subjected them to high rate of topology change and a heavy traffic load. With the vehicles, radios, and site used in our testbed, we forced the protocols to operate in an environment in which *all* links between nodes change status at least every 220 seconds. Ignoring the additional factor of packet loss due to wireless errors, on average, the network topology changed every 4 seconds.

7.1.1. Network Topology

Figure 7.1 shows a logical view of the ad hoc network testbed. The ad hoc network includes 5 moving car-mounted nodes, labeled **T1-T5**, and 2 stationary nodes, labeled **E1** and **E2**. Each of these nodes communicates using 900 MHz WaveLAN-I radios. These radios do not implement the IEEE 802.11 MAC protocol [43], since at the time the testbed was built, the WaveLAN-IEEE radios were not available. The ad hoc network is connected to a *field office* using a 2.4 GHz point-to-point wireless link over a distance of about 700 m. This point-to-point link does not interfere with the 900 MHz radio interfaces on the individual ad hoc network nodes.

At the field office is a router **R** that connects both the ad hoc network and an IP subnet at the field office back to the *central office* via a wide-area network (i.e., the Internet). The visualizer node **V** is used to monitor the status of the ad hoc network, and the GPS reference station (**RS**), located on the roof of the field office, is responsible for sending differential real-time kinematic (RTK) GPS corrections to nodes in the ad hoc network.

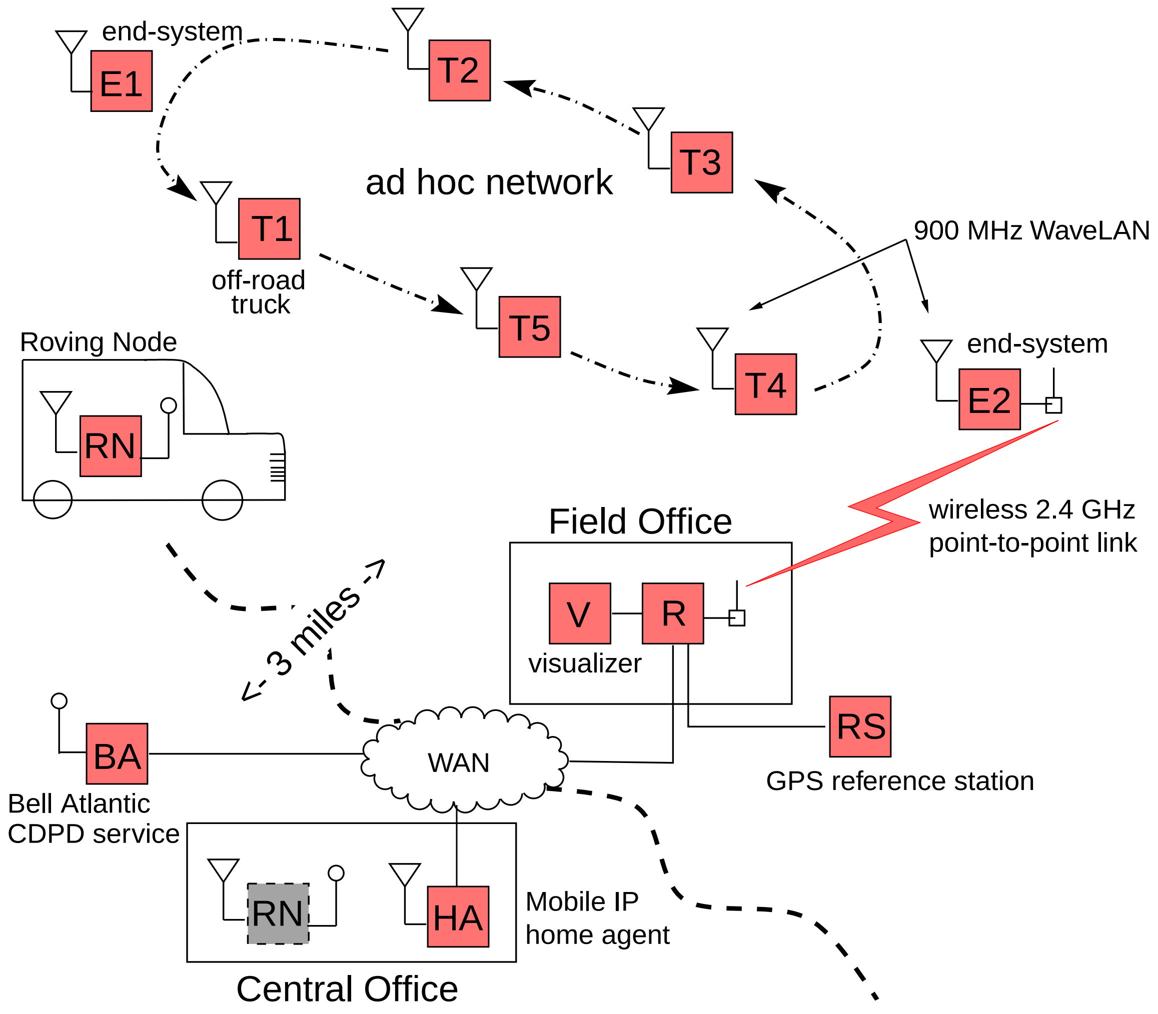


Figure 7.1 Logical overview of the testbed network.

The central office is home to a *roving node* (**RN**) that drives between the central office and the ad hoc network, participating in three networks: its home wireless LAN, the Bell Atlantic Cellular Digital Packet Data (CDPD) service, and the ad hoc network. Node **HA** provides Mobile IP home agent services [82] for the roving node, enabling it to leave the central office and still maintain routing connectivity with all of the other nodes in the Internet.

During a typical experiment, which we call a *run*, the drivers of each of the cars carrying an ad hoc network node follow the course shown in Figure 7.2 at speeds varying from 25 to 40 Km/hr (15 to 25 miles per hour). Each run lasts for between 30 and 120 minutes. The road we use is open to general vehicle traffic and has several Stop signs, so the velocity of each node varies in a complex fashion, just as it would in any real network. Likewise, the nodes are constrained to move along the paved surfaces of the site. This prevents us from testing the arbitrary topologies used in some theoretical simulations on abstract flat planes, but enables us to evaluate the performance we can expect in a real application.

During each run, the network was subjected to the composite workload shown in Table 7.1, consisting of synthetic voice calls, bulk data transfer, location-dependent transfers, and real-time data. The workload includes: each node making one voice call to every other node once per hour; each node transferring a data file to every other node once per hour; each moving node (**T1-T5**) making a location-dependent transfer to **E1** when located within 150 m of **E1**; and multicast differential RTK GPS corrections. Finally, the workload also includes real-time situational awareness data sent by the Position and Communication Tracking daemon (PCTd) on each node to the visualizer machine located at the Field Office. Sent once per second, these packets contain the current location of the node, as read from the node's GPS unit, and status information on the node, such as the number of packets it has forwarded, dropped, queued, originated or retransmitted.

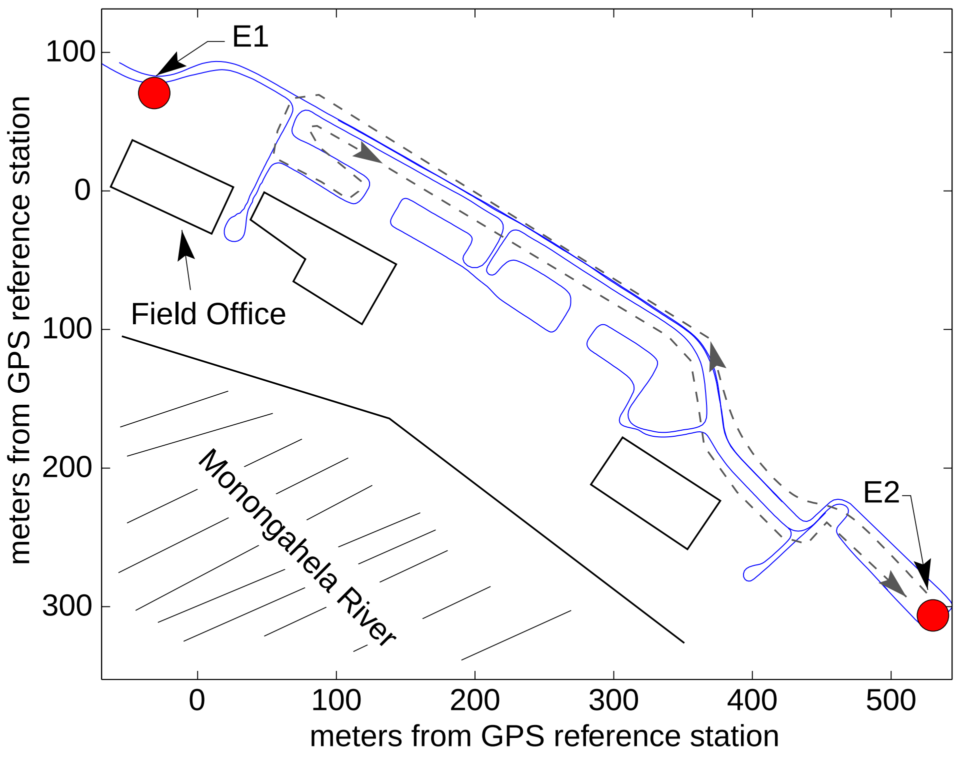

E1
E2
Field Office
Monongahela River
100
0
100
200
300
0
100
200
300
400
500
meters from GPS reference station
meters from GPS reference station

Figure 7.2 Map of the testbed site showing the endpoint locations and the typical course driven by the nodes.

Table 7.1 Load offered to the network by nodes in the testbed.

Application	Rate	Protocol	Size
Voice	6/hour/node	UDP	Average of 180 Kbytes
Data	5/hour/node	TCP	30, 60, or 90 Kbytes
Location-dependent	When near **E1**	TCP	Average of 150 Kbytes
GPS	1 pkt/sec multicast	UDP	150 bytes
PCTd	1 pkt/sec/node unicast	UDP	228 bytes

The visualizer machine continuously displays on a map of the site the last known location of each node. The visualizer can also graph the protocol status information, and it logs all the data it receives, thereby allowing a detailed replay of the run after the fact.

7.1.2. Network Configuration

All communication among the ad hoc network nodes, **T1-T5**, **E1**, and **E2**, is routed by the Dynamic Source Routing (DSR) protocol. DSR is a network layer routing protocol in that operates at the IP layer of the network stack (OSI layer 3) and permits interoperation between different physical network interfaces. However, our DSR implementation conceptually operates as a virtual link layer just under the normal IP layer. This allows DSR to route packets using IP addresses as flat identifiers when the other nodes in the ad hoc network are not organized as hierarchical subnets.

Nodes **T1-T5**, **E1**, and **E2** are assigned IP addresses from a single subnet, with **E2** acting as a gateway between the Internet and the ad hoc network subnet. **E2** was manually configured to use the DSR protocol for communication on one network interface (the 900 MHz WaveLAN link), and to use normal IP routing over the other interface (the 2.4 GHz point-to-point link to its default router **R**). Packets from nodes in the Internet destined to addresses in the ad hoc subnet are routed by normal means to **E2**, which has a statically configured route directing them out the network interface to the ad hoc network. Once forwarded into the ad hoc network by **E2**, DSR takes care of routing the packets to their final destination, which often requires multiple hops inside the ad hoc network. As explained in Section 7.2.1, nodes in the ad hoc subnet (i.e., **T1-T5** and **E1**) did not have to be configured to use **E2** as a default router: when nodes in a DSR ad hoc network send packets to nodes not in the ad hoc network, the DSR protocol itself automatically routes the packets to the nearest gateway (**E2** in this case), where they are forwarded into the Internet. The gateway node, **E2**, also provides Mobile IP foreign agent services to any Mobile IP nodes that visit the ad hoc network.

The roving node **RN** has available several methods for connecting to the Internet, and uses Mobile IP [82] to choose the best method as it drives around the city. **RN** is normally within range of the WaveLAN network at the central office, and its WaveLAN network interface carries an IP address belonging to the central office subnet. When **RN** is roving away from the central office, it uses Mobile IP to register a *care-of address* with its *home agent* on the central office subnet. While **RN** has a care-of address registered with the home agent, the home agent intercepts packets destined to **RN**, and tunnels each to the care-of address using encapsulation.

When **RN** cannot use its primary WaveLAN interface because it is not in range of any other WaveLAN radios, it uses its CDPD modem to connect to the CDPD service from Bell Atlantic (now known as Verizon), and registers its CDPD IP address with its home agent. Once **RN** realizes it is in range of a DSR network, it can use the DSR protocol to communicate directly with the other nodes in the ad hoc network. To enable packets from nodes outside the DSR network to reach **RN**, it registers itself with its home agent via the foreign agent at **E2**, just as in normal Mobile IP. When **E2** receives a tunneled packet, it checks to see if the packet is destined to a node registered as visiting the ad hoc network. If so, **E2** routes the packet to the visiting node using DSR.

7.2. DSR Extensions for Internet Connectivity

While mechanisms to scale DSR to networks of hundreds of mobile nodes are outside the scope of this thesis, we have begun exploring several ideas in that design space [17]. We have already, however, extended the mechanisms of Route Discovery and Route Maintenance to support communication between nodes inside the ad hoc network and those outside in the greater Internet.

7.2.1. Integration with the Internet

In order to connect a DSR ad hoc network to the Internet, we first require that each node choose a single IP address, called its *home address*, by which it is known to all other nodes. By having a single home address, nodes in the ad hoc network maintain a constant identity as they communicates with nodes inside and outside the network. This notion of a home address is identical to that defined by Mobile IP [82]. As in Mobile IP, each node is configured with its home address and uses this address as the IP source address for all of the packets that it sends.

Figure 7.3 illustrates node **T2** inside the ad hoc network discovering a route to a node **D** outside the network. To demonstrate the generality of this technique, nodes **T1-5** and **E1** have not even been configured to know that they are part of a common subnet with **E2** as a default router. The routing tables in **T1-5** and **E1** simply provide a default route out the DSR virtual interface. When **T2** originates a packet destined

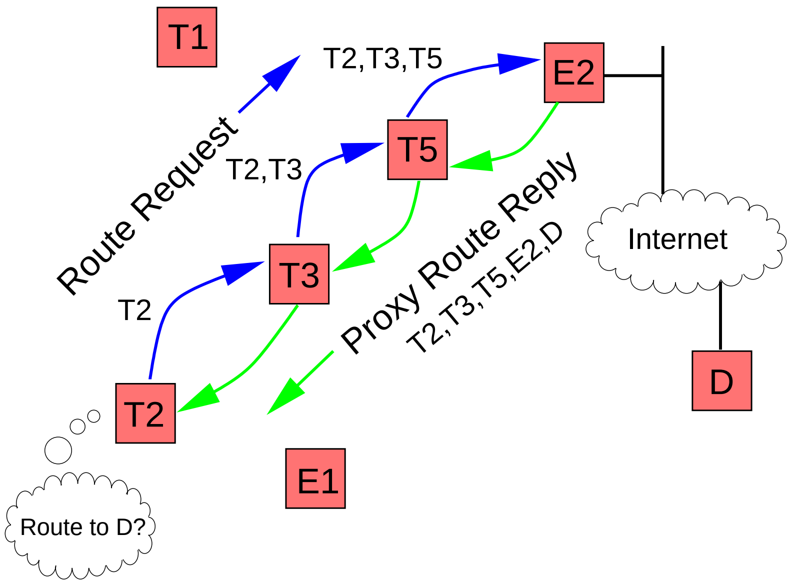


Figure 7.3 Route Request for a node not in the ad hoc network being answered by the Foreign Agent

to **D**, the packet is given to the DSR code for delivery. Assuming **T2** does not already have a cached route to **D**, it begins a Route Discovery for **D**.

As the ROUTE REQUEST from **T2** targeting **D** propagates, it is eventually received by the gateway node **E2**, which consults its routing table. If it believes **D** is reachable outside the ad hoc network, it sends a *proxy* ROUTE REPLY listing itself as the second-to-last node in the route, and marking the REPLY such that **T2** will recognize it as a proxy reply. If the target node **D** actually is inside the ad hoc network, then node **T2** will receive a ROUTE REPLY from from both **E2** and **D**. Since **T2** can distinguish which replies are proxy replies, it can prefer the direct route when sending packets to **D**.

7.2.2. Integration with Mobile IP

Since node **RN** in Figure 7.1 must be able to participate in different IP subnets depending on its current location, it uses Mobile IP to connect to the Internet. Figure 7.4 shows an example where **RN** is homed in a subnet not belonging to the ad hoc network, but it has wandered into range of the ad hoc network. As described in Section 7.1.2, node **E2** provides Mobile IP foreign agent services, in addition to being configured as a gateway between the ad hoc network and the Internet.

As part of normal Mobile IP operation, **RN** periodically checks to verify that it is currently using the best means available to maintain connection with the Internet. We configured **RN** to operate in LAN mode as its top preference, to connect to the Internet via a DSR ad hoc network as second choice, and to connect via CDPD when no other options are available. When **RN** receives DSR packets, such as ROUTE REQUESTs, ROUTE REPLYs or data packets with DSR source routes on them, it knows it is within range of a DSR network and enables that connectivity option in its Mobile IP code.

If node **RN** decides its best connectivity would be via the ad hoc network, it transmits a Mobile IP AGENT SOLICITATION piggybacked on a ROUTE REQUEST targeting the IP limited broadcast address (255.255.255.255). This allows the SOLICITATION to propagate over multiple hops through the ad hoc network, though gateways will not propagate it between subnets. When the foreign agent at **E2** receives

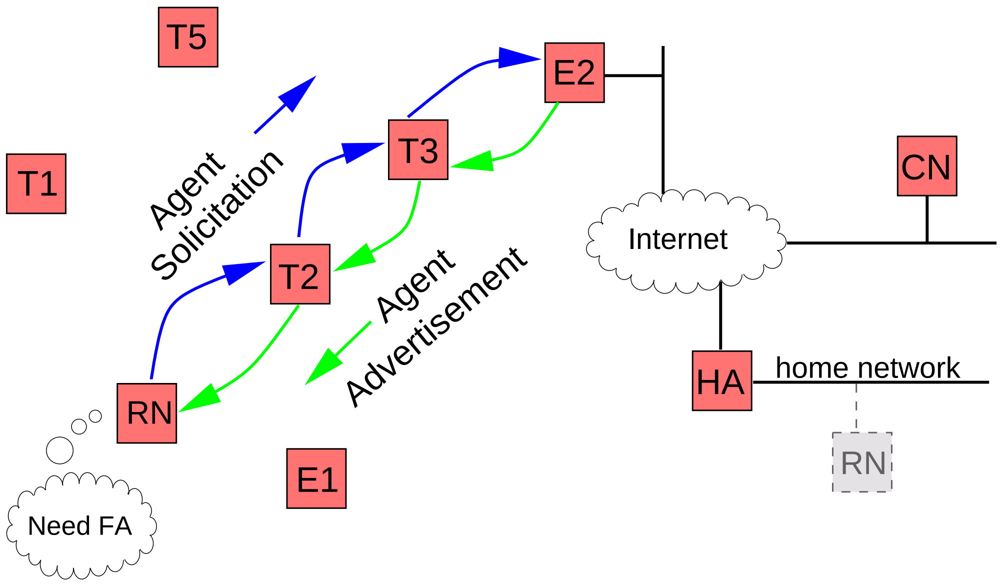


Figure 7.4 The roaming node **RN** registering with a foreign agent located on **E2** in the ad hoc network.

the SOLICITATION, it will reply with an AGENT ADVERTISEMENT, allowing **RN** to register itself with this foreign agent and with its home agent as a Mobile IP mobile node visiting the ad hoc network. Once the registration is complete, the mobile node's home agent will use Mobile IP to tunnel packets destined for mobile node **RN** to the foreign agent at **E2**, and **E2** will deliver the packets locally to the mobile node using DSR.

7.3. Layer 3 Mechanisms for Acknowledgments and Retransmission

Since the WaveLAN-I radios do not provide link-layer reliability, we implemented a hop-by-hop retransmission and acknowledgment scheme within the DSR layer that provides the feedback necessary to drive DSR's Route Maintenance mechanism. One interesting aspect of our ARQ scheme was the use of passive acknowledgments [58], which significantly reduces the number of acknowledgment packets transmitted when compared to acknowledgment schemes that acknowledge every packet (e.g., IEEE 802.11 [43]).

7.3.1. Implementation Overview

Our implementation utilizes passive acknowledgments whenever possible, meaning that if node **A** originating or forwarding a packet hears the next hop node **B** forward the packet, **A** accepts this as evidence that the packet was successfully received by **B**. If **A** fails to receive a passive acknowledgment for a particular packet that it has transmitted to **B**, then **A** retransmits the packet, but sets a bit in the packet's header to request an explicit acknowledgment from **B**. Node **A** also requests an explicit acknowledgment from **B** if **B** is the packet's final destination, since in this case, **A** will not have the opportunity to receive a passive acknowledgment from **B**. To avoid the inefficiencies of a stop-and-wait ARQ scheme, node **A** uses a buffer to hold packets it has transmitted that are pending acknowledgment plus an identifier based on the IP ID field [89] to match acknowledgments with buffered packets.

This acknowledgment procedure allows **A** to receive acknowledgments from **B** even in the case in which the wireless link from **A** to **B** is unidirectional, since explicit acknowledgments can take an indirect route from **B** to **A**. During an average run, 90 percent of the acknowledgments used a direct one-hop route, and 10 percent of the acknowledgments were sent over routes with multiple hops. While this strongly suggests the presence of unidirectional links in the network, it does not support a conclusion that 10 percent of the packets travel over a unidirectional link. Once a multiple-hop route for acknowledgments is discovered, it may continue to be used for some period of time even after the direct route begins working again.

When performing retransmissions at the DSR layer, we also found it necessary to perform duplicate detection, so that when an acknowledgment is lost, a retransmitted packet is not needlessly forwarded through the network multiple times. The duplicate detection algorithm used in our implementation specified that a node should drop a received packet if an identical copy of the packet was found in a buffer awaiting either transmission or retransmission. We found that this simple form of duplicate suppression was sufficient, and that maintaining a separate history of recently seen packets was not necessary.

7.3.2. Heuristics for Selecting Timeout Values

Early in the design of our retransmission mechanism, we found that contention for the wireless medium produced enough variance in the Round Trip Time (RTT) between neighboring nodes that using a fixed value for the retransmission timer was not practical, and that *adaptive* retransmission timers were required.

Our initial implementation of an adaptive retransmission timer employed the scheme used by TCP [104, p. 301], where a smoothed RTT estimator (`srtt`) and a smoothed mean deviation (`rttvar`) are maintained independently for each next hop to which a node is communicating. The retransmission timeout (RTO) is then computed as:

$$RTO = srtt + 4 \times rttvar$$

Unfortunately, the variance in RTT prevented this implementation from performing adequately. Frequently, the RTO would not adapt quickly enough to congestion in the network, causing packets to be retransmitted unnecessarily and creating even more congestion. It also suffered from the fact that the RTO to each next hop was computed independently, while the need to defer transmissions due to congestion is common across all neighbors accessed via the same network interface.

We found that several simple methods of reacting to increasing congestion did not work. For example, we tried an algorithm that feeds into the smoothing function for RTT estimation an RTT sample of twice the current smoothed RTT estimate whenever there is a retransmission timeout. This algorithm causes the value of the RTT estimator, and hence the retransmission timer, to tend to diverge and remain pegged at its maximum value, even after congestion has subsided.

We developed a successful retransmission timer algorithm by including a heuristic estimate of the level of local congestion, so that the retransmission timer could react quickly to changes. [1] One of the simplest ways for a node to measure congestion in the network is to look at the length of *its own* network interface transmit queue. Specifically, if more than 5 packets are found in the interface transmit queue — meaning that congestion is starting to occur — we increase the value of the retransmission timer 20 ms for each packet in the queue. Assume that there are N packets in the network interface queue. For $N \leq 5$, the retransmission timeout is computed as before:

$$RTO = srtt + 4 \times rttvar$$

However, for $N > 5$, the retransmission timeout is computed as:

$$RTO = srtt + 4 \times rttvar + (N \times 20 \text{ ms})$$

This heuristic allows the retransmission timer to increase quickly during periods of congestion and then return just as quickly to its computed value once the congestion dissipates. In 4710 measurements over several runs, approximately 75% of the packets transmitted use the minimum retransmission timer value of 50 ms. However, for the other 25% of the packets, the retransmission timer adjusted itself to values between 60 ms and 920 ms. The wide range indicates that an adaptive retransmission scheme is required for good performance if acknowledgments are implemented above the link layer.

7.4. Wireless Propagation

When our testbed network operated without the layer 3 acknowledgment and retransmission scheme, the average packet loss rate over a single hop was measured as 11 percent. With the ARQ scheme described in Section 7.3, the average loss rate over a single hop dropped to 5 percent. The losses are highly correlated with position, but also demonstrate the highly variable nature of wireless propagation due to scattering, multi-path, and shadowing effects in the real world.

In a result that surprised us, we also found that the probability of a successful packet reception rolls off very slowly with distance. At twice the nominal transmission range, the probability of successfully receiving a packet is still 40 percent. Section 7.6 describes some of the *problems* caused when propagation is *better* than expected.

7.4.1. Small and Large Scale Fading

As an example the highly variable nature of wireless propagation, Figure 7.5 shows the signal level at which packets were received when sent by node **T4** directly over one hop to **T3**. This data was measured during a full run of the testbed, so all the nodes (**T1-T5**) were in motion and exchanging data when it was recorded.

[1]This heuristic was created and implemented by Josh Broch.

Dropped packets are marked with a '+' and shown at -90 dBm; the actual value of the received signal strength is unknown because the network interface hardware does not report signal strength for packets that not successfully received. From 1390 s to 1404 s the figure shows most packets being successfully delivered, but with significant numbers of multi-path fades of 20 dB or greater. Scattered among these are packet losses, most of which the link-layer retransmission algorithm is able to correct. However, given the impracticality of predicting such losses, link-layer retransmission is the only defense against unrecoverable packet loss. That link-layer retransmission can come from the link-layer directly, as in IEEE 802.11 DCF, or from lower levels of the routing layer, as in our implementation of DSR. The variability in propagation creates a significant level of inherent packet loss with which higher layers must be prepared to cope.

During the period from 1405 s to 1410 s, however, Figure 7.5 shows what can be best described as a routing protocol error. Although other paths were available, the protocol continued to send packets directly from **T4** to **T3** when nearly half those packets were being lost. As discussed further in Section 7.6, today's routing protocols must be extended in at least one of two directions. They must predict the continued decrease in average signal level that occurred from 1400 s to 1406 s and switch to a new route, or they must maintain state to record that a particular link is becoming lossy and avoid it until such time as it functions well again.

7.4.2. Correlation of Location with Packet Loss

As part of conducting an initial survey of the site, we found it particularly helpful to obtain a rough characterization of the site's propagation environment. We had two cars drive the course at about 30 Km/hr (20 miles per hour), one following the other at a separation of 90 meters, with the trailing car transmitting

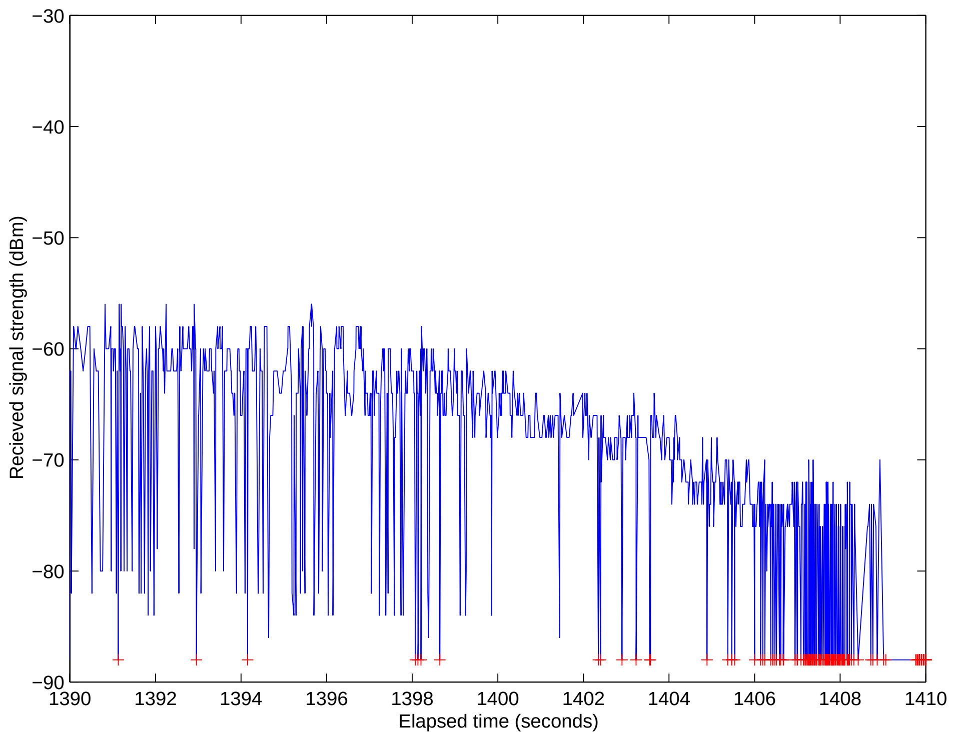


Figure 7.5 Received signal strength of packets sent directly between two nodes during a full run of the testbed.

1024-byte packets to the lead car 10 times per second. The cars made three laps of the course, a total driving time of about 660 seconds.

Figure 7.6 shows the position of the cars during the intervals when more than 1% of the total packet loss occurred (the 90 meter separation is significantly less than the nominal WaveLAN-I range of 250 m). Essentially all of the loss bursts occurred while the nodes were on the straightest part of the course with clear line-of-sight to each other. There is no elevation change along that portion of the course, and the parking lots along the south side of the road were empty during the test. While we have been unable to completely determine the cause of these losses, our current hypothesis is that the radios are suffering from multipath reflection off cars unrelated to the testbed traveling on a road approximately 20 feet north of the course.

7.5. Jitter in Inter-packet Spacing

Isochronous communications, such as interactive voice and video, are extremely sensitive to packet jitter. In order to evaluate how well isochronous communication works across our testbed network, we evaluated the jitter experienced by the synthetic audio traffic — part of the composite workload described in Table 7.1. Each audio connection consisted of alternating simplex packet streams between the communicating parties, and each packet stream consisted of 250-byte UDP packets sent 8 times per second, giving an average bit rate of 16 Kb/s. This traffic pattern was chosen to model the Push-To-Talk mobile radios commonly used on mine and construction sites.

During an average run, a total of 98,000 voice packets are originated, which represents 3.4 hours of voice communication. Of these 98,000 packets, 3.8% are lost in the network. The testbed does not contain any special handling rules for the voice packets, so the packets are retransmitted according to the same mechanism described in Section 7.3. The jitter, defined as the variation in inter-packet spacing introduced by the network, ranged from -9.4 s to 6.5 s. The extreme values of jitter are rare, and typically occur

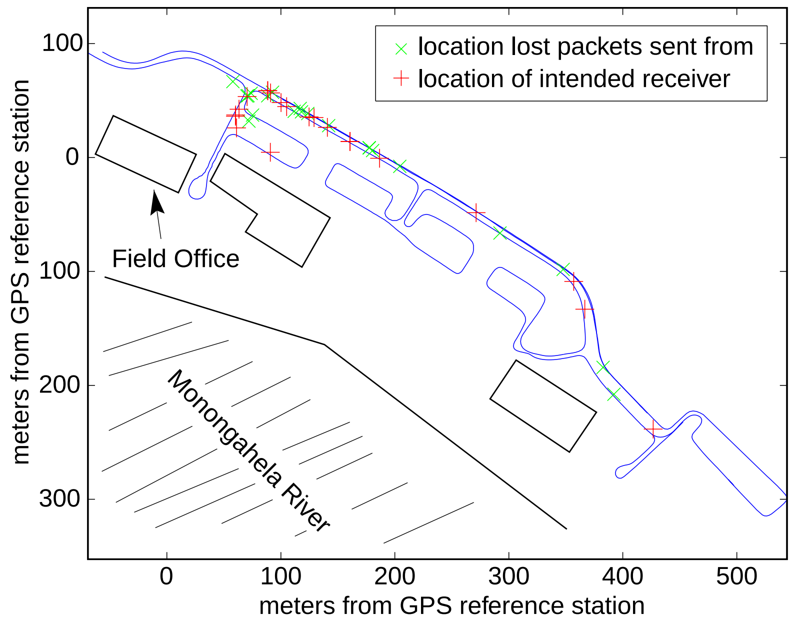


Figure 7.6 The location of the sending node and receiving node when more than 1% of packet loss occurred. Nodes maintained 10 s separation (90 m).

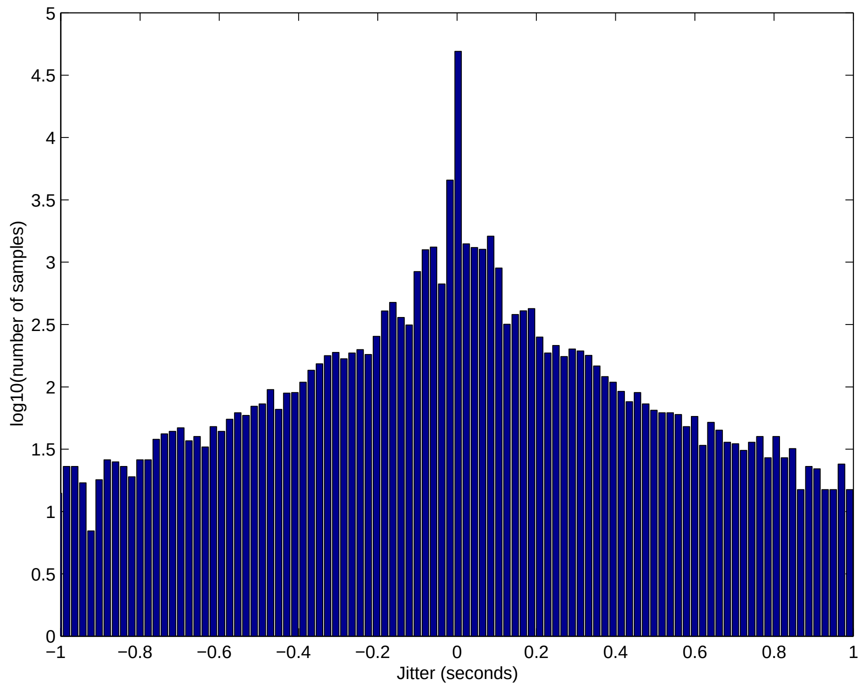


Figure 7.7 Distribution of jitter. Y-axis is on a log scale for clarity.

when the network is temporarily partitioned. During a partition, the voice sources continue to send data, but the packets are buffered inside the network. The result is a burst of back-to-back packet arrivals at the destination when the partition heals. As an area for future research, more sophisticated packet handling algorithms might detect these delayed, but time-sensitive, packets and drop them inside the network to conserve resources [70].

When the most extreme 2% of jitter samples are removed as outliers, the range of jitter drops to between -1.04 s and 1.02 s. The mean jitter is 0.001 s, and the standard deviation 0.143 s. Figure 7.7 shows the distribution of jitter samples using a histogram. The y-axis is on a log scale for clarity; each bar is 20 ms wide, and there are 10 times more packets with 0.0 s of jitter than packets with either $\pm$ 20 ms of jitter. 90% of the voice packets experience a jitter between -0.2 s and 0.2 s, so a playback buffer of 400 ms should be sufficient for voice communication.

7.6. The Need for Hysteresis in Route Selection

As mentioned above, the packet loss rate seen between any two nodes in the network is highly variable, depending not only on the positions of the nodes involved, but also on the movement of other objects around the nodes. In working with TCP connections carried over the testbed network, we found this variability had a disastrous effect on the bandwidth delivered by these TCP connections. Other researchers have addressed related problems in layer 4 and at the boundary between layer 3 and layer 4 [5, 6, 40, 66], so in this section we will concentrate on a layer 3 issue caused when radio propagation is transiently *better* than expected.

7.6.1. Evaluating the Need for Hysteresis in Route Selection

To isolate the layer 3 issue, we conducted an experiment using our DSR implementation, with three nodes arranged in a line. The experiment consisted of the node at one end of the line transferring a large amount of data to the node at the other end of the line via a TCP connection. The nodes were mounted on cars and positioned by driving the first and third cars in opposite directions away from the middle car as far apart as possible, while still allowing both of the end nodes to successfully flood ping the middle node with 1024-byte packets. The flood ping tests were carried out serially, meaning that only one node was flood pinging at a time, and a successful flood ping test was defined as sending ping packets as fast as possible with more than 99% of the ping ICMP ECHO REPLY packets being received over a 10 s sample. Once positioned, the nodes remained stationary for the remainder of the test. For the purpose of discussion, let node **A** be the TCP source, **B** the middle node, and **C** the TCP sink.

This is a particularly challenging scenario, not only because electromagnetic propagation is highly variable, but because the specific setup of this test deliberately introduces the hidden terminal problem. A number of times during runs of this test, we saw the DSR retransmission timer expire, creating ROUTE ERRORs and subsequent ROUTE REQUESTs as the nodes attempt to restore connectivity.

As described in Section 2.5.1, Route Discovery in the current implementation of DSR operates by first sending a non-propagating ROUTE REQUEST that the target node will answer with a ROUTE REPLY if the target can directly receive the originator's REQUEST. If the originator does not receive a ROUTE REPLY within 30 ms, it sends a propagating ROUTE REQUEST that floods through the network in a controlled fashion to discover multi-hop routes to the target. In the testbed implementation, nodes do not generate replies based on information in their Route Caches.

In order to obtain baseline performance metrics, we used our `macfilter` tool [71], which allows us to create a synthetic propagation environment among nodes in a laboratory setting. The `macfilter` tool was described in Section 6.3. We first made 5 independent TCP transfers of 1 MB each from node **A** to node **C**. Over these 5 transfers, TCP averaged 0.50 Mb/s (61 KB/s) with a standard deviation of 0.079 Mb/s. When the same transfers were performed in the field with real radio propagation, however, the average data rate was 0.12 Mb/s (14.65 KB/s) with a standard deviation of 0.025 Mb/s — only 25% of the throughput measured in the lab. In fact, some of the 1 MB data transfers, which were set up to last for a maximum of 50 seconds, timed out before the entire megabyte could be transfered. In these cases, we report the average data rate for the 50-second duration of the connections.

The TCP sequence number plot from a typical two-hop connection in the testbed implementation is depicted in Figure 7.8. Sequence numbers marked with a small dot were transmitted using the two-hop route **A**$\rightarrow$**B**$\rightarrow$**C**, whereas sequence numbers marked '$\times$' were transmitted using the one-hop route **A**$\rightarrow$**C**. The dashed vertical lines in the figure indicate when the TCP source **A** performed a Route Discovery consisting only of a non-propagating ROUTE REQUEST, and the solid vertical lines indicate when a Route Discovery consisting of both a non-propagating and a propagating ROUTE REQUEST occurred (by the rules of Route Discovery, if the non-propagating REQUEST returns a REPLY, the propagating REQUEST is not sent).

The TCP connection in Figure 7.8 made very good progress for the first 9 seconds of the connection, using almost exclusively a two-hop route. However, during the time interval from 9 s to 22 s, the connection makes almost no progress, sending about 30 KB in this 13 s interval. After processing a ROUTE ERROR at 9 s, the TCP source (node **A**) initiates a Route Discovery. The non-propagating ROUTE REQUEST is answered directly by node **C**, causing **A** not to send a subsequent propagating ROUTE REQUEST and thus to use a single-hop route to node **C**. The poor quality of this single-hop link leads to repeated ROUTE ERRORs and Route Discovery attempts. Finally, at 18 s, node **A**'s non-propagating ROUTE REQUEST fails to return any REPLYs and so **A** transmits a propagating REQUEST. This results in the discovery of both the single-hop route and the two-hop route through intermediate node **B**. By this time, TCP has backed off and does not offer the next packet to the network until 22 s. Node **A** attempts to use the one-hop route that it discovered,

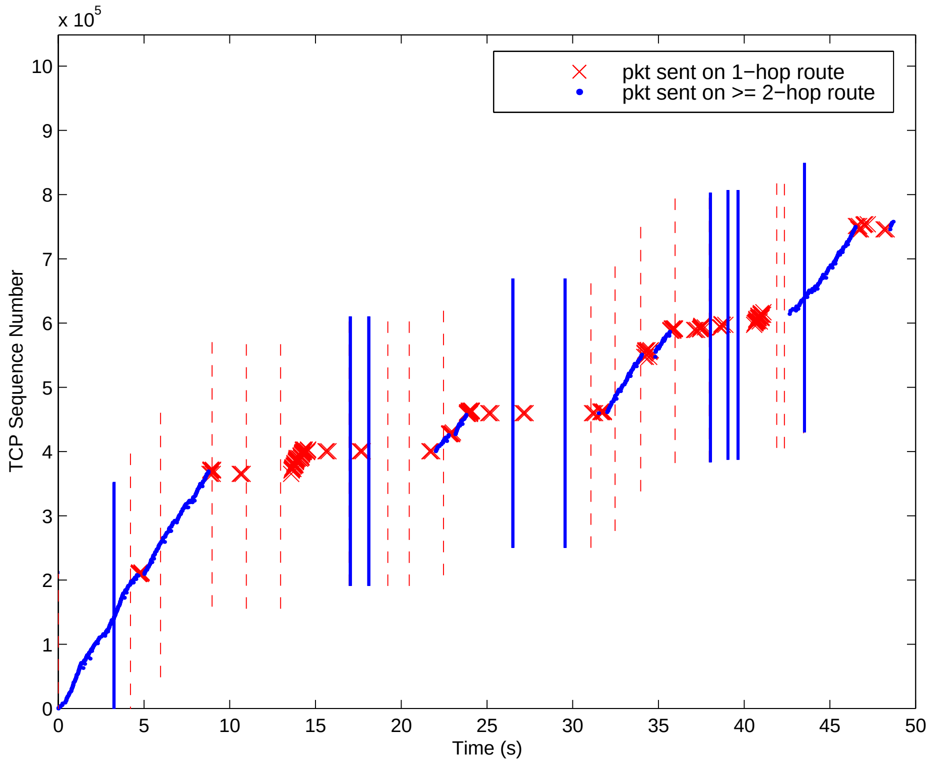


Figure 7.8 A TCP sequence number plot for a 1 MB transfer over a two-hop route. The vertical lines indicate the times at which the TCP source initiated Route Discovery; the dashed lines indicate the times at which only a non-propagating ROUTE REQUEST was transmitted and the solid lines indicate both a non-propagating and a propagating ROUTE REQUEST.

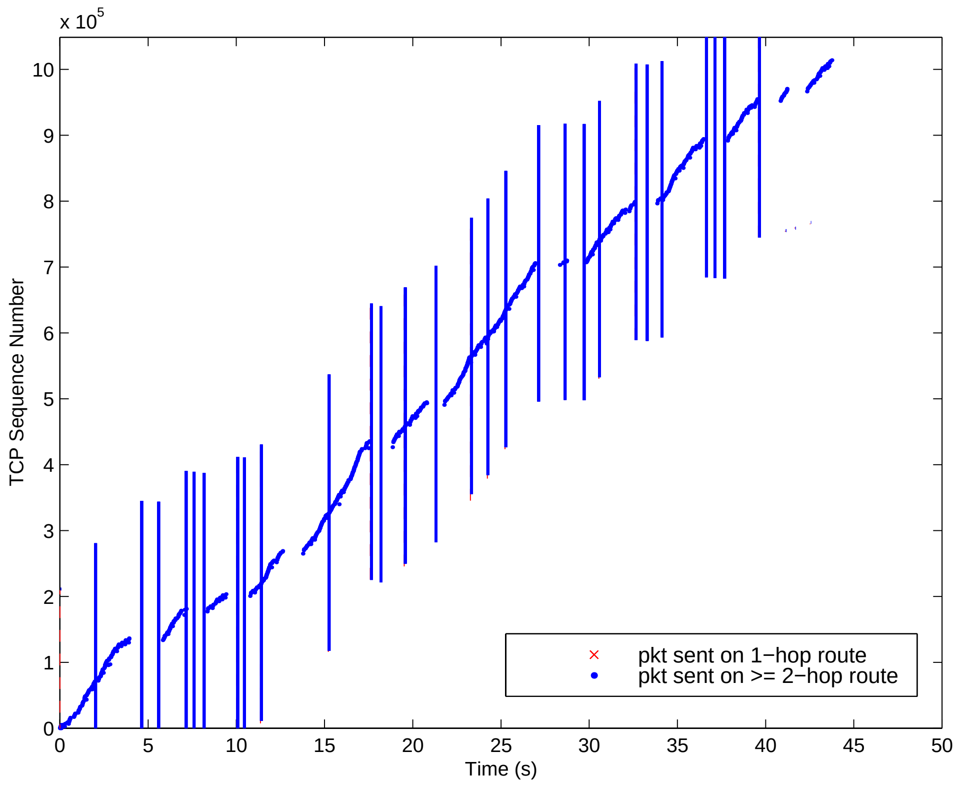


Figure 7.9 A TCP sequence number plot for a 1 MB transfer over a two-hop route when the `macfilter` utility was used on both the source and destination nodes to prevent the use of single-hop routes. The vertical lines indicate the times at which the TCP source initiated Route Discovery (a non-propagating REQUEST followed by a propagating REQUEST).

finds that it does not work well, removes the one-hop route from its Route Cache, and begins using the two-hop route. At this time, the connection again starts making progress. The same scenario of repeated attempts to use a one-hop route occurs again during the intervals 25 s to 32 s and 35 s to 43 s.

This scenario illustrates an important challenge for ad hoc network routing protocols and argues strongly that all routing protocols need some ability to remember which recently used routes have been tried and found not to work. Even traditional distance vector-style protocols are subject to this problem, as they attempt to minimize a single metric — usually hop count.

For example, if **A**, **B**, and **C** in the three-node scenario discussed above were all using a distance vector routing protocol, **A** would sometimes hear advertisements from **C**. Since the direct route to **C** is more optimal in terms of hop count than the route through the intermediate node **B**, **A** would attempt to send all of its packets directly to **C** until that direct route timed out. In other words, without some type of local feedback or other hysteresis, **A** will often try to send its packets directly to **C**, effectively black-holing most of these packets since that link is so unreliable. Protocols such as Signal Stability based Adaptive routing (SSA) [26] may behave much better in this scenario.

7.6.2. The Potential Gain of Better Route Hysteresis

To evaluate the potential gain of having a mechanism that would prevent the repeated use of the poor direct route from **A** to **C**, we emulated perfect routing information by using our `macfilter` to eliminate the discovery of 1-hop routes. Figure 7.9 shows the TCP sequence number plot for a 1 MB transfer on our testbed implementation using this "perfect routing." The flat plateaus are no longer present, and the throughput is 30% higher. The remaining Route Discoveries are triggered when packets are repeatedly lost due to variation in wireless propagation, and techniques such as notifications to the TCP module could be used to prevent the TCP stalls that follow these Route Discoveries.

7.6.3. Achieving Better Route Hysteresis

There are two basic approaches that can be used to achieve better route hysteresis. The first approach is for the routing protocol to attempt to determine the root physical cause of the packet losses it sees, in order to decide whether the losses represent a condition it should take action to avoid. The second approach is more statistical in nature. Rather than determining the root cause of packet loss, the protocol acts in whatever fashion appears to statistically reduce the likelihood of packet losses.

Each approach has advantages and disadvantages. Using information such as received signal strength and physical position would appear to make it easy to determine the root physical cause of a packet loss, but determining the root cause of packet losses is not possible in many situations and provides fragile and rapidly changing solutions fraught with potential error in most situations. The statistical solutions have the potential to be more robust, but I have not yet produced one I am happy with. A combination of the two approaches has the best potential, but developing and evaluating these solutions is unfortunately left to future work.

Improving Route Hysteresis by Using External Information

One source of the route hysteresis problem described in the sections above is that the routing protocol does not have sufficient information to determine the root cause of a link failure. If the routing protocol was able to determine whether a link failure was due to congestion, a temporary variation in signal propagation, or a hard failure caused by complete separation of the two nodes, it would be able to better predict which links will work in the future and be able to discover and use routes made of stable links.

Determining the root cause of a link failure, as indicated by receipt of a ROUTE ERROR message, is a difficult task. On it's own, the routing protocol is capable solely of sending packets and receiving packets, and it determines which of the packets it sent were not received by the lack of an acknowledgment. In order to determine the root physical cause for the lack of an acknowledgment, the routing protocol must be provided with additional external information. Examples of external information include the position of the nodes as determined by GPS, topographic maps of terrain and buildings, and the signal strength at which a packet was received.

As a simple example of the use of external information, Hu has altered Route Maintenance in DSR to send a special type of “soft” ROUTE ERROR whenever a packet is received with a signal strength less than some configured level [42]. The soft ROUTE ERROR serves as notice to the packet’s originator that the route it is using is becoming more error-prone, and enables it to perform a Route Discovery to find alternate routes before the previous route has even broken. Hu reports that this technique significantly improved the human-perceived quality of multimedia data streams being carried across the network.

In networks where each node knows its position (e.g., due to the use of GPS as in our network), communicating nodes could use the location information propagated by the other nodes to model the position of their correspondents. For example, after the DSR process running on **A** received several ROUTE ERRORs for the link from **A** to **C** as described in Section 7.6.1 above, it could insert into its cache a marker that would prevent it from using the link **A**$\rightarrow$**C**, and thereby force it to use the better route **A**$\rightarrow$**B**$\rightarrow$**C**. **A** would retain the negative information in its cache until it finds that either node **A** or **C** has changed position in some reasonably significant way.

An even more sophisticated approach could combine the signal strength at which the node received ROUTE REPLYs, the position of the nodes, and the mobility pattern of the nodes to estimate the probability of successful communication over a particular route. Punnoose and colleagues have experimented with such an approach and had some success with it [92].

Improving Route Hysteresis Without External Information

As an alternative to using external information to deduce whether a link will work well or not, the routing protocol can attempt to determine the trend in behavior of a link over time and thereby predict whether it will have stable performance in the future. DSR already contains the data structures necessary to implement such a mechanism. The Route Cache can be used to record the previous behavior of each link, and the Get operation on the Route Cache can be written to prefer routes with only well behaved links [41].

One potential solution [51] is for DSR to cache a *negative link* for each link for which it receives a ROUTE ERROR. The Get operation on the Route Cache would simply not return routes containing such negative links, and attempts to insert a route into the cache that traverse the negative link would be prohibited. The negative link would be deleted from the cache after a period based on the estimated rate of link fluctuation, but would prevent DSR from repeatedly attempting to use the poor quality link. The drawbacks of this solution are the difficulty of designing a strategy to pick a reasonable timeout value and the possibility that the only way to reach some destinations may be over an unstable link (this solution would render such destinations unreachable).

Another potential solution is for DSR to simply not remove links from its Route Cache when it receives a ROUTE ERROR for the link. Rather, DSR would increment a counter called `link_errors` on the link. When retrieving a route to a destination from the Cache, the Get operation would sum the `link_errors` along each candidate route and prefer the route with the fewest errors. Among several available routes to a destination, the most stable one would be chosen.

The drawback of the `link_errors` scheme is the difficulty of inventing a rule for fairly updating the `link_errors` counters when new links are added to the Cache. For example, if there are two routes to a destination in the Cache that have been in use for a long period of time, both routes may have accumulated a large number of `link_errors`. When a new route to the destination comprised of new links is added to the cache, those links will have recorded no `link_errors` yet and the new route will be preferred over the older routes. If the new route is of poor quality, it would be suboptimal to allow the new route to drop as many packets as the old routes had before the new route's `link_error` count catches up to the old routes' and the old routes are chosen again. What is needed is a rule for retaining the relative ordering of the old routes, but leveling the playing field so the new route starts out level with the best of the old routes. The logic is non-trivial given that each link may be part of routes to many different destinations and we would like to share the information about the number of link errors the link has experienced across all those destinations.

7.7. Chapter Summary

Our testbed successfully demonstrates that DSR, and the on-demand mechanisms it embodies, can be successfully implemented in real networks carrying meaningful traffic across multiple hops. The testbed features 2 stationary nodes, 5 car-mounted nodes that drive around the testbed site, and 1 car-mounted roving node that enters and leaves the site. DSR not only routes packets between the nodes in the ad hoc network, but it seamlessly integrates the ad hoc network into the Internet via a gateway. DSR was also extended to integrate with Mobile IP, allowing nodes to roam transparently between the ad hoc network and normal IP subnets.

The testbed is novel among non-military testbeds in the completeness of its implementation and the relatively large number of nodes that comprise it. Furthermore, among all ad hoc network testbeds, the rate of topology change is significantly greater than has been previously studied.

In analysis of data from runs of the testbed, we measured the jitter introduced by the network and found that, even under a full load and without any special QoS handling, the network can support a Push-To-Talk voice system. We also analyzed how the novel heuristic for the DSR-layer packet acknowledgment and retransmission scheme allows nodes to rapidly adapt to changing levels of network congestion.

Finally, we demonstrated that the selection of routes that are used in ad hoc networks must include some form of hysteresis to prevent the use of routes that exist only transiently. In particular, this chapter highlighted the problems caused when packets occasionally travel further than the normal wireless transmission range, a phenomenon that might appear neutral or beneficial at first glance. Our measurements of a full-scale network also point out the need for further refinement of our simulation models of wireless propagation. While this is not a new insight for Electrical Engineering researchers in radio communications, the dramatic effects on routing protocols have not been previously addressed. Our experiences with the testbed have led us to reexamine the wireless propagation model in the simulator and extend it with additional models of packet loss.

Chapter 8

Related Work

Most of the previous work related to this thesis is described where appropriate in the preceding discussion. This chapter ties up the remaining loose ends.

8.1. Routing Protocol Design

Ramanathan and Steenstrup [93] and Royer and Toh [98] both provide an excellent survey of routing protocols designed for use in ad hoc networks. Three other routing protocols with very different designs than DSR, namely DSDV, TORA, and AODV, were briefly described in Section 5.1.

The earliest protocol that I know of with a structure similar to DSR was created as part of the DARPA PRNET project [58]. This protocol [59] used source routes for controlling the forwarding of packets across an ad hoc network, and could use a flooding mechanism like basic Route Discovery to find these source routes. It does not appear to include the notion of Route Caches, Route Maintenance, or any of the optimizations to Route Discovery described in this thesis. The original inspiration for DSR came from considering how ARP [88] could be extended to handle networks where some nodes may beyond the sender's wireless transmission range, reachable only via a series of multiple hops. When all nodes are directly reachable, ARP is the only routing protocol required. The resulting multi-hop form of the protocol is also similar to IEEE 802.3 Source Routing Bridges [87].

Brayer [15] proposed a protocol that is based on source routes for use in satellite networks. However, the protocol differs significantly from DSR in two ways. First, the protocol uses the exchange of source routes to build hop-by-hop routing tables. Second, its Route Discovery method is based on random forwarding. Unlike in DSR, where packets are routed according to a source route choosen by the packet's originator, in Brayer's protocol nodes who do not know how to reach a destination simply transmit the packet to a randomly choosen neighbor. The assumptions are that the packet will evenutually find its destination and that it will build up useful hop-by-hop routing state in each of the nodes it passes through while traversing the network, reducing the need for random forwarding in the future. The protocol was evaluated on networks with restricted topologies, and it is unclear how it would perform in the more general topologies used in the experiments reported in this thesis.

DSR has already had a significant impact on the development of routing protocols for ad hoc networks. Many of the other ad hoc network routing protocol proposed after DSR either incorporate DSR itself, are based on the DSR framework of Route Discovery and Route Maintenance, or include DSR's component mechanisms.

The Zone Routing Protocol (ZRP) [38, 37, 39] is a framework for routing in ad hoc networks that uses a periodic Intra-zone Routing Protocol for creating and routing inside clusters. It uses an on-demand Inter-zone Routing Protocol for routing between clusters, and the designers of ZRP suggest DSR as a candidate. Several of the optimizations proposed for improving the efficiency of the ZRP Inter-zone Routing Protocol also appear similar to DSR optimizations to Route Discovery.

The Cluster Based Routing Protocol (CBRP) [47] uses a clustering protocol to elect and maintain cluster heads, but then uses DSR for routing packets between clusters. Signal Stability based Adaptive routing

(SSA) [26] and Associativity Based Routing (ABR) [106, pp. 303–306] each use a scheme like DSR for routing, but chooses routes based on the length of time the routes' component links have been receiving packets with strong signal levels.

AODV [85, 86] is a routing protocol for ad hoc networks that combines mechanisms from both DSR and DSDV [83]. However, the two protocols are fundamentally different, as AODV is based on hop-by-hop routing, rather than on source routing like DSR. AODV constructs routes on-demand using a Route Discovery mechanism like DSR's, but its hop-by-hop nature requires it to use hard-state sequence numbers (adapted from DSDV) in order to prevent routing loops. Hop-by-hop routing in AODV eliminates the need for a source route in each data packet, which reduces the byte overhead of the protocol, but at the cost of dramatically increasing packet count overhead due to missed opportunities for optimizations. The path-state extension [70, 16] to DSR will reduce source route overhead while maintaining the properties of DSR, such as route shortening and support of asymmetric routes, that AODV does not contain and that rely on the existence of source routes in some form.

Researchers both inside and outside the Monarch Project are now working to add additional features to the basic DSR framework. Castaneda and Das have developed a query-localization optimization that operates as part of Route Discovery to reduce the overhead of finding a new route to a destination when a route that had been in use breaks [18]. Ko and Vaidya extended DSR's Route Discovery mechanism using Global Positioning System (GPS) information to direct the ROUTE REQUEST towards the area mostly likely to contain the targeted node [63]. Gerla has developed cluster algorithms that improve the scalability of Route Discovery [35]. Hu has developed a more sophisticated Route Cache data structure that enables DSR to operate as an on-demand, loosely consistent Link State protocol [41]. I have developed extensions to DSR that enable the explicit management of network resources, such as battery power and the use of bandwidth, in order to provide better-than-best-effort Quality of Service [70, 16]. Examples of such services providing delay bounds, guaranteed buffer space at intermediate nodes in the network, and notifications of the bandwidth available to traffic flowing to destinations in the network.

8.2. Routing Protocol Simulation and Evaluation

Before the Monarch Project developed our extensions to the *ns-2* simulator, most academic research work was being done in simulators that grossly failed to model the ad hoc environment. The failures of these simulators to accurately model the environment misled protocol designers, who received erroneous feedback in their attempts to refine their protocols.

As an example of earlier simulators, one simulator modeled the network as a "densely-connected honeycomb" with constant node density. There is no notion of node mobility. Each node is connected to a fixed set of neighbors by separate links that cycle between an active and an inactive state independent of all other links. The links are error-free, and no dynamics below the network layer are modeled. Radio propagation, medium access, collisions, and physical node mobility are completely ignored; it is even possible for a node to correctly receive two simultaneous transmissions. In the simulation, link transitions cause interrupts that give the protocols immediate feedback whenever a link goes up or down. However, in reality, a node can only detect that a link has broken if it is trying to use the link or if it fails to receive expected periodic beacons over it, and a node can only recognize the existence of a new neighbor when it receives a packet from that neighbor.

Another earlier simulator was used by Park and Corson [80] to analyze TORA, comparing it to an "idealized" link state routing protocol. Their results show TORA delivering over 90% of its packets in all cases, but these results are different from ours (Chapter 5) because of the many simplifications they made in simulating the environment. In order to avoid congestion, their simulation used a packet transmission rate of only 4, 1.5, or 0.6 packets per *minute* per node. Simulations were run for 2 hours, the mean time between failure for links was varied from 32 minutes to 1 minute, and the average network connectivity was artificially held constant at either 90%, 70%, or 50%. The protocol on which TORA is, in part, based [21] was also evaluated using an earlier simulator, but the simulator used in that study had the same problems as the simulator used by Park and Corson, and the metrics used are incomparable to ours.

I previously simulated DSR [51] using the same Random Waypoint mobility model as in this thesis, but with a simulator that lacked both a realistic model of radio propagation and a realistic MAC layer such as IEEE 802.11. With these pieces missing, the problem faced by the routing protocol is greatly simplified and leads to questionable results. For example, in the early work, propagation delay, capture effects, MAC-layer collisions, and the effects of congestion due to large packet sizes are unaccounted for. Furthermore, broadcast and unicast packets were delivered with the same probability, and, as noted in Section 5.4.3, this is not a realistic assumption.

The range of topologies DSR was subjected too was also much more limited, since that study did not consider networks with more than 24 nodes, and the maximum network diameter was 4 hops. Movement speeds ranged from 0.3 to 0.7 m/s, with the nodes moving about in a 9m $\times$ 9m space using short-range infrared wireless transmitters with a 3-meter range.

Using our current *ns-2* simulator, Johansson et. al [48] compared the performance of DSR with AODV, but in scenarios they created to resemble specific real-world applications. They found that DSR showed consistently good performance. In the few settings and metrics where DSR performed worse than other protocols, they attribute the performance to DSR's increased byte overhead causing congestion effects. As noted earlier, DSR's byte overhead results mainly from an explicit source route on each packet, and I have designed extensions to the protocol that eliminate this overhead.

Contemporaneously with the development of our *ns-2* simulator, Bargrodia and colleagues developed another simulator package, PARSEC [4], that supports modeling of wireless ad hoc networks. PARSEC grew out of an earlier simulation language called Maise [7], and it is based on a variant of the C language that has been extended to support the implementation of event handlers for a discrete-event simulation. Unlike *ns-2*, PARSEC is not open-source.

Our extensions to *ns-2* have now been incorporated into the mainline *ns-2* distribution. Many research groups are actively using our extensions to support their own research on ad hoc networks, including groups at INRIA [1], The University of Illinois at Urbana-Champaign [75, 103], Columbia University [8], the Swedish Institute of Computer Science [31], the University of California at Santa Cruz [97], Ericsson [8, 48], the University of Maryland at College Park [46], UCLA [67], Microsoft Research [108], the University of Pennsylvania, the University of Texas at Dallas, Texas A&M, Nokia, Telcordia, and many others.

8.3. Metrics for Analyzing Protocols

The metrics used in this thesis are not the only possible metrics for use in analyzing routing protocols for ad hoc networks. The set of metrics used in this thesis were chosen by selecting those that concisely captured salient characteristics of the protocols. I considered many other metrics, and other researchers have used still others.

In the first evaluation of DSR, I used two metrics. Instead of measuring routing overhead, I calculated

$$\frac{\text{number of transmissions}}{\text{optimal number of transmissions}}.$$

Conceptually, the metric measures the number of transmissions required for each transmission that did useful work in getting a packet closer to its destination. The "number of transmissions" counts the total number of packets transmitted during the simulation (data and routing), while "optimal number of transmissions" calculates the minimum number of transmissions that should have been needed to deliver each of the data packets to their destinations if perfect routing information were available. Thus, the expression measures the *work efficiency* of the routing protocol, with value of 1.0 indicating zero overhead. The metric represents a summary of the work efficiency of the entire system, however, including all layers of the network stack. For example, packets dropped at intermediate nodes in the network count only against the number of transmissions, and so penalize the work efficiency when the link-layer fails to deliver a packet along a perfectly valid route provided by the network layer. The metric can be useful as a summary of the work efficiency of the entire system, but it lumps together all the factors that contribute to the overhead of the system and prevents factoring out the cost of the routing protocol alone, which the metrics used in this thesis permit.

Also in these earlier simulations, I characterized the path optimality of the routing protocol by reporting the *ratio* of the average route length used to the optimal route length that could have been used if the routing protocol had perfect information. However, I did not report the actual route lengths used, and since both numbers are averages, this ratio tended to blur the ability to see the dynamics of the protocol. As a result, in this thesis I adopt the clearer convention and present path optimality data as the *difference* between the optimal and actual path lengths used. I separately show the distribution of shortest possible paths, which both helps characterize the scenarios and provides context for the path optimality numbers.

In a paper describing an early routing protocol based on a link reversal algorithm [21], as described in Section 8.2, Corson and Ephremides introduce the metric of routing power, which was defined as the "average message throughput divided by the average message delay." This metric is not comparable to metrics used in this thesis and conveys less information than would separately presenting the average throughput and average delay.

Corson also evaluated TORA on the metrics of "bandwidth utilization", "mean message packet delay", and "message packet throughput." Bandwidth utilization includes both routing and data packets, and so it contains the information that I have broken out into routing overhead and path optimality. The message packet throughput metric is identical to my packet delivery ratio.

Mean packet delay measures the latency of a packet from the time it enters the network until it arrives at its destination. For any individual packet, the metric's value depends on the latency of protocol mechanisms (such as Route Discovery), the level of network congestion, and in some sense the routing overhead, since the more routing packets there are, the longer data packets will have to queue waiting for transmission. In comparing protocols, I did not measure delay, since the mean packet delay is mostly a function of properties of the routing protocol for which I was not trying to control. Individual data packets will either experience the delay of a Route Discovery or they will not, so it is better to report the delay caused by a Route Discovery and the frequency of Route Discovery, as I have done, than to divide up the latency of a Route Discovery over the many data packets sent using routes discovered by that Route Discovery and report the mean packet delay. When the latency of Route Discovery is subtracted out, mean packet delay in the ad hoc networks I have studied is primarily a function of queuing delay and media acquisition time. While a QoS sensitive routing protocol might change its behavior or routing decisions based on QoS factors, such as the queue lengths or media acquisition times at nodes in the network, at the time this thesis was written none of the protocols I studied attempted to control these QoS factors.

8.4. Direct Emulation of Ad Hoc Networks

Kevin Fall created the initial emulation support inside the *ns-2* simulator and has used it for experimenting with TCP performance via the modulation of a simulated network link [29]. My direct emulation system extends Fall's work by using the simulator to emulate the entire network, including routing layer, link layer, and physical layer effects. I have also constructed the tools needed to easily develop scenarios for the emulation system. The `ad-hockey` tool permits both the creation of scenarios and the visualization of a scenario as the emulation system recreates it. Working with Ke, I have extended the simulator with additional optimizations and modifications to improve the real-time performance of the simulator, such as adding copy-by-reference for the live packet objects and the use of the Intel Pentium's nanosecond resolution clock. Finally, I have developed a methodology for validating that the emulation system is not introducing significant artifacts, and extended *ns-2* with the additional logging necessary to carry out this validation. Previous reports of emulation work have not attempted a validation of the emulation system, while we provide both a quantitative and qualitative description of the emulation artifacts.

Researchers at BBN have created a link-layer emulation of ad hoc networks based on trace modulation that is more sophisticated than my independently created `macfilter` system (Section 6.3). They use one logical machine for each node in the network, and on a central server they precalculate the path loss between all pairs of locations in the emulated terrain using a terrain-based propagation model. The central server is configured with the nodes' movement patterns, and it periodically pushes out to each node the node's current path loss to all other nodes. The nodes then use these path loss figures to calculate the probability that they will receive a packet sent by another node [27].

My direct emulation system differs from the BBN emulation system and from the `macfilter` scheme in two ways. First, direct emulation avoids the need to have one logical node per node in the network, which can dramatically reduce the equipment requirement. Second, direct emulation can model the interaction of individual packets and detect individual collisions, where the other methods can model only the average packet loss behavior. This ability to model individual packet fates is extremely useful in debugging protocols. For example, a pathologic behavior where many nodes all transmit a packet at the same time (e.g., in response to the receipt of a broadcast packet) will show up clearly in direct emulation and not in a trace modulation method.

Another implementation of network emulation, called dummyNet, was developed by Rizzo [96]. The dummyNet system is a form of trace modulation, where the logical channel is implemented by code in a special virtual network interface. The delay, bandwidth, and packet loss rate of all packets routed through the network interface can be controlled, though it appears that the equivalent of our trace files are intended to be constructed manually. A system similar to dummyNet was used in the evaluation of TCP Vegas by Ahn and Danzig [2].

8.5. Testbeds

Very few testbeds for multi-hop ad noc networks have been constructed, and almost no documentation of experiments on these testbeds exists in the public domain. According to our knowledge of the unclassified literature on mobile network testbeds, we believe that our testbed has produced the highest rate of topological change of any testbed thus far built. Unfortunately, aside from the testbed described in this thesis, the most advanced tests have been conducted by the military, and the results of these tests are not available without a security clearance.

Both the SURAN [9] and PRNET projects [59] conducted field experiments of their systems, but the tests in the open literature describe scenarios with few nodes in which only a few (i.e., two or three) links break and form during a run. During a February 1998 test of the NTDR system [100] at Fort Huachuca, approximately 60 nodes participated in the network, with about a dozen of them moving. The network successfully carried video traffic from cameras mounted on the moving nodes, but I do not know of any more detailed performance results [28]. The Task Force XXI Advanced Warfighting Experiment sought to build and deploy a Tactical Internet [33], but personal communication indicates the experiment was not very successful, with Packet Delivery Ratios around 30% in some cases.

There are now many ongoing efforts to build more testbeds for ad hoc networking. Royer and Perkins are working on an implementation of the AODV protocol, but it was not complete as of November 2000 [99].

Toh and his students have started on the implementation of a testbed, but there is no published information available about it.

Chapter 9

Conclusions

The ability for nodes to form ad hoc networks in the absence of communication infrastructure is a critical area of current research. There are existing communication needs which ad hoc networks can meet, such as military and commercial applications, and the development of ad hoc network technology will enable new classes of applications. With the potential for low cost deployment and high availability, coupled with the dropping costs of wireless transceivers, ad hoc networks are becoming economically and technologically feasible right now.

9.1. Thesis Summary

The major conclusions of the work presented in this thesis fall into three general areas.

- First, careful simulation analysis of the on-demand mechanisms that make up the DSR protocol show that the latency of Route Discovery is reasonable, that the cost of Route Discovery can be effectively controlled by localization, and that Route Maintenance allows Route Caches to be used while minimizing the loss of packets due to stale cache data. Chapter 4 raised four major risks that come with the use of on-demand mechanisms in a routing protocol, and then used data measured from simulations of DSR to show that these risks can be mitigated.

  When the performance of DSR, in terms of packet loss rate and routing protocol overhead, is compared with that of three other routing protocols designed for use in multi-hop ad hoc networks, DSR significantly outperforms the other protocols across a wide range of scenarios. As explained in Chapter 5, two overall trends in this study were clearly visible. The first trend is that the more on-demand behavior a protocol uses, the better it performs. The second trend is that protocols perform better the more they are able to reduce the number of nodes that must react to a topology change.

- Second, emulating ad hoc networks has tremendous value, since it provides an easy way to experiment with protocols and applications without the hassle of a physical testbed. The experiments described in Chapter 6 show how emulation enables the testing of protocol implementations, and the value such testing provides. Emulation of ad hoc networks is not only possible, it is possible for interesting scenarios. For example, a 400 MHz Pentium II can easily handle scenarios with 16 nodes and a total of 100 packets being sent per second — modern processors can presumably handle substantially larger scenarios.

- Finally, the ultimate test for any protocol is to deploy it in one of the environments for which it was designed and measure it to determine if it operated successfully. DSR has successfully been implemented in a real 8-node network carrying meaningful traffic across multiple hops. Chapter 7 described the network and several lessons learned from working with it. Of greatest importance are the problems caused when packets occasionally travel further than normal wireless transmission range, a phenomenon that might appear neutral or beneficial at first glance. However, the phenomenon means that routing protocols need some form of hysteresis in the selection of routes to prevent the use of routes that exist only transiently.

9.2. Thesis Contributions

This thesis has provided a proof-by-demonstration that a practical, very dynamic, ad hoc network can be built and has described tools that make the area of ad hoc network research more accessible to others. In more detail, the thesis makes three main contributions in the areas of protocol design, research methodology, and tools to support future research.

In the area of protocol design, I have quantitatively shown the value of on-demand mechanisms in routing protocols for ad hoc networks. I have also examined in detail how the performance of several on-demand mechanisms interact to create the DSR protocol. Through the comparison of four different routing protocols for ad hoc networks, I identified behaviors in each protocol that had not previously been understood, and provided an explanation for the behavior. The characterization of the on-demand mechanisms and routing protocol behavior should help guide future designers in developing DSR or other protocols. The final contribution in this area is the DSR protocol itself, a complete protocol which has already demonstrated its usefulness in a realistic setting as part of our ad hoc network testbed.

In the area of research methodology, I have developed a series of metrics for analyzing routing protocols and the Random Waypoint movement model which have become widely adopted among researchers in the area. I have also developed novel metrics, such as discovery containment, that help explain the behavior and cost of Route Discovery. A final contribution in this area is the methodology for directly emulating ad hoc networks, which enables the users of ad hoc networks to experiment easily with the behavior of their applications in an ad hoc network without the cost of actually deploying a complete network.

In the area of tool generation, I have participated in the design, development, and testing of numerous tools that others have already used to further research in the area. The simulator, the emulator, the testbed, and their associated tools and scripts are among the concrete contributions of this thesis.

9.3. Future Work

Numerous areas for future research are described throughout this thesis, with the improvement of direct emulation and the development of larger testbeds among the most exciting opportunities.

Increased refinement of the routing protocols for ad hoc networks will always be possible. With the tools developed as part of this thesis, hopefully these refinements can be directed towards the areas that will most dramatically improve the performance of the protocols in real-world deployments. For DSR in particular, significant improvements in its performance have been realized through research on the Route Cache data structure [41], and there are potentially other savings available.

Another area for research in DSR and other protocols is the development of general control packet aggregation schemes. The number of control packets transmitted by DSR could be reduced if nodes had rules for buffering control packets, such as ROUTE REPLYs, for short periods to allow multiple control packets to be aggregated together. Such techniques would likely involve the use of timer-based mechanisms. As with the LRU cache policies described in Section 4.5.3, if well-tuned the mechanisms could improve the performance of DSR, but such tuning is difficult and must be performed for each particular environment and would violate the pure on-demand nature of DSR that has been the basis of this research.

This thesis has not directly addressed Quality of Service issues for ad hoc networks, and this area has been receiving extensive attention recently, as described in Section 8.1.

Even without this future work, multi-hop ad hoc networks are ready for deployment in actual applications. Ultimately, it will be the analysis of these deployed systems and people's usage of them that leads to the next set of innovations in ad hoc networks.

Bibliography

[1] Imad Aad and Claude Castelluccia. Differentiation Mechanisms for IEEE 802.11. In *Proceedings of IEEE INFOCOM 2001*, pages 209–218, Anchorage, Alaska, April 2001.

[2] J.S. Ahn, Peter B. Danzig, Z. Liu, and L. Yan. TCP Vegas: Emulation and Experiment. In *Proceedings of the SIGCOMM '95 Conference: Communications Architectures & Protocols*, pages 185–195, August 1995.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. Internet Request For Comments RFC 2581, April 1999.

[4] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zengand J. Martin, and H.Y. Song. PARSEC: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, 31(10), October 1998.

[5] Bikram S. Bakshi, P. Krishna, N. H. Vaidya, and D. K. Pradham. Improving the Performance of TCP over Wireless Neworks. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, pages 365–373, May 1997.

[6] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance Over Wireless Networks. In *Proceedings of the First Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'95)*, pages 2–11, November 1995.

[7] R. L. Bargodia and W.-T. Liao. Maise: A Language for the Design of Efficient Discrete-Event Simulation. *IEEE Transactions on Software Engineering*, 20(4), April 1994.

[8] Michael Barry, Andrew T. Campbell, and Andras Veres. Distributed Control Algorithms for Service Differentiation in Wireless Packet Networks. In *Proceedings of IEEE INFOCOM 2001*, pages 582–590, Anchorage, Alaska, April 2001.

[9] David A. Beyer. Accomplishments of the DARPA Survivable Adaptive Networks SURAN Program. In *Proceedings of MILCOM'90*, 1990.

[10] Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang. MACAW: A Media Access Protocol for Wireless LAN's. In *Proceedings of the SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, pages 212–225, August 1994.

[11] Rajendra V. Boppana and Satyadeva P. Konduru. An Adaptive Distance Vector Routing Algorithm for Mobile, Ad Hoc Networks. In *Proceedings of IEEE INFOCOM 2001*, pages 1753–1762, Anchorage, Alaska, April 2001.

[12] Peter J. Braam. The Coda Distributed File System. *Linux Journal*, (50), June 1998.

[13] R. Braden, editor. Requirements for Internet Hosts – Communication Layers. RFC 1122, October 1989.

[14] Scott Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[15] Kenneth Brayer. Mobile Earth Stations via Low-Orbit Satellite Network. *Proceedings of the IEEE*, 72(11):1627–36, November 1984.

[16] Josh Broch, David B. Johnson, and David A. Maltz. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks. Internet-Draft, draft-ietf-manet-dsr-03.txt, October 1999. Work in progress.

[17] Josh Broch, David A. Maltz, and David B. Johnson. Supporting Hierarchy and Heterogeneous Interfaces in Multi-Hop Wireless Ad Hoc Networks. In *Proceedings of the Workshop on Mobile Computing* held in conjunction with *The International Symposium on Parallel Architectures, Algorithms, and Networks,* pages 370–375, Perth, Australia, June 1999.

[18] Robert Castaneda and Samir R. Das. Query localization techniques for on-demand routing protocols in ad hoc networks. In *The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 186–194, Seattle, WA, August 1999.

[19] TAPR The Tucson Amateur Packet Radio Corp. http://www.tapr.org/, 1995. Last updated: 2001.

[20] M. S. Corson, S. Papademetriou, P. Papadopoulos, V. Park, and A. Qayyum. An Internet MANET Encapsulation Protocol (IMEP) Specification. Internet-Draft, draft-ietf-manet-imep-spec-01.txt, August 1998. Work in progress.

[21] M. Scott Corson and Anthony Ephremides. A Distributed Routing Algorithm for Mobile Wireless Networks. *Wireless Networks*, 1(1):61–81, February 1995.

[22] M. Scott Corson and Joe Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501, January 1999.

[23] Susan B. Davidson. Optimism and Consistency in Partitioned Distributed Database Systems. *ACM Transactions on Database Systems*, 9(3):456–481, September 1984.

[24] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[25] Stephen E. Deering and Robert M. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998.

[26] Rohit Dube, Cynthia D. Rais, Kuang-Yeh Wang, and Satish K. Tripathi. Signal Stability based Adaptive Routing (SSA) for Ad Hoc Mobile Networks. *IEEE Personal Communications*, pages 36–45, February 1997.

[27] Chip Elliot. Personal Communication. Telephone discussion of BBN testbed and emulation experience, August 1999.

[28] Chip Elliot. Discussion of NTDR Field Tests. Private Email Correspondence, April 2001.

[29] Kevin Fall. Network Emulation in the VINT/ns Simulator. In *Proceedings of the Fourth IEEE Symposium on Computers and Communications (ISCC'99),* July 1999.

[30] Kevin Fall and Kannan Varadhan, editors. *ns* Notes and Documentation. The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC, January 1999. Available from http://www-mash.cs.berkeley.edu/ns/.

[31] Laura M. Feeney and Martin Nilsson. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proceedings of IEEE INFOCOM 2001*, Anchorage, Alaska, April 2001.

[32] Daniel M. Frank. Transmission of IP datagrams Over NET/ROM Networks. In *ARRL Amateur Radio 7th Computer Networking Conference*, pages 65–70, October 1988.

[33] James A. Freebersyser and Barry Leiner. A DoD Perspective on Mobile Ad Hoc Networks. In *Ad Hoc Networking*, edited by Charles E. Perkins, chapter 2, pages 29–51. Addison-Wesley, 2001.

[34] E.M. Gafni and D.P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 29(1):11–18, January 1981.

[35] Mario Gerla, Taek Jin Kwon, and Guangyu Pei. On Demand Routing in Large Ad Hoc Wireless Networks with Passive Clustering. In *Proceedings of the IEEE Wireless Communications and Networking Conference*, Chicago, IL, September 2000.

[36] Ventura County ARES AREA 6 Training Group. *Packet Radio Training Course Reference Manual*. Ventura County, CA. Available from http://www.rain.org/ jkrigbam/packet.htm.

[37] Zygmunt J. Haas and Marc R. Pearlman. A New Routing Protocol for the Reconfigurable Wireless Networks. In *Proceedings of the 6th IEEE International Conference on Universal Personal Communications*, pages 562–566, October 1997.

[38] Zygmunt J. Haas and Marc R. Pearlman. The Performance of Query Control Schemes for the Zone Routing Protocol. In *Proceedings of SIGCOMM'98*, pages 167–177, September 1998.

[39] Zygmunt J. Haas and Marc R. Pearlman. The Zone Routing Protocol (ZRP) for Ad Hoc Networks. Internet-Draft, draft-ietf-manet-zone-zrp-01.txt, August 1998. Work in progress.

[40] Gavin Holland and Nitin Vaidya. Analysis of TCP Performance over Mobile Ad Hoc Networks. In *The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 219–230, 1999.

[41] Yih-Chun Hu and David B. Johnson. Caching Strategies in On-Demand Routing Protocols for Wireless Ad Hoc Networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'00)*, pages 231–242, Boston, MA USA, August 2000.

[42] Yih-Chun Hu and David B. Johnson. Demonstration of Preliminary QoS Features of DSR. Demonstrated at DARPA GloMo Technology Expo, July 2000.

[43] IEEE Computer Society LAN MAN Standards Committee. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std 802.11-1999. The Institute of Electrical and Electronics Engineers, New York, New York, 1999.

[44] Internet Engineering Task Force (IETF). http://www.ietf.org/.

[45] IPX Router Specification. Novell Part Number 107-000029-001, Document Version 1.30, March 1996.

[46] Lusheng Ji and M. Scott Corson. Differential Destination Multicast—A MANET Multicast Routing Protocol for Small Groups. In *Proceedings of IEEE INFOCOM 2001*, pages 1192–1201, Anchorage, Alaska, April 2001.

[47] Mingliang Jiang, Jinyang Li, and Y.C. Tay. Cluster Based Routing Protocol (CBRP). Internet-Draft, draft-ietf-manet-cbrp-spec-01.txt, August 1999. Work in progress.

[48] Per Johansson, Tony Larsson, Nicklas Hedman, Bartosz Mielczarek, and Mikael Degermark. Scenario-Based Performance Analysis of Routing Protocols for Mobile Ad-Hoc Networks. In *The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 195–206. ACM, August 1999.

[49] David B. Johnson. Routing in Ad Hoc Networks of Mobile Hosts. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 158–163, December 1994.

[50] David B. Johnson. Validation of Wireless and Mobile Network Models and Simulation. In *Proceedings of the DARPA/NIST Workshop on Validation of Large-Scale Network Models and Simulation*, Fairfax, VA, May 1999. Available from http://www.monarch.cs.cmu.edu/papers.html.

[51] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, edited by Tomasz Imielinski and Hank Korth, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[52] David B. Johnson and David A. Maltz. Protocols for Adaptive Wireless and Mobile Networking. *IEEE Personal Communications*, 3(1):34–42, February 1996.

[53] David B. Johnson and David A. Maltz. Protocols for Adaptive Wireless and Mobile Networking. *IEEE Personal Communications*, 3(1):34–42, February 1996.

[54] David B. Johnson, David A. Maltz, and Josh Broch. DSR The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks. In *Ad Hoc Networking*, edited by Charles E. Perkins, chapter 5, pages 139–172. Addison-Wesley, 2001.

[55] David B. Johnson, David A. Maltz, Yih-Chun Hu, and Jorjeta Jetcheva. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks. Internet-Draft, draft-ietf-manet-dsr-04.txt, November 2000. Work in progress.

[56] Greg Jones. Introduction to Packet Radio. In *Packet Radio: What? Why? How? / Articles and Information on General Packet Radio Topics*, number Publication #95-1. Tucson Amateur Packet Radio (TAPR) Corp., 1995. Read as http://www.tapr.org/tapr/html/pktf.html.

[57] Greg Jones. Why Packet Radio? In *Packet Radio: What? Why? How? / Articles and Information on General Packet Radio Topics*, number Publication #95-1. Tucson Amateur Packet Radio (TAPR) Corp., 1995. Read as http://www.tapr.org/tapr/html/pktf.html.

[58] John Jubin and Janet D. Tornow. The DARPA Packet Radio Network Protocols. *Proceedings of the IEEE*, 75(1):21–32, January 1987.

[59] Robert E. Kahn, Steven A. Gronemeyer, Jerry Burchfiel, and Ronald Kunzelman. Advances in Packet Radio Technology. *Proceedings of the IEEE*, 66(11):1468–1496, November 1978.

[60] Phil Karn. MACA — A New Channel Access Method for Packet Radio. In *Proceedings of the 9th Computer Networking Conference*, pages 134–140, September 1990.

[61] Philip R. Karn, Harold E. Price, and Robert J. Diersing. Packet Radio in the Amateur Service. *IEEE Journal on Selected Areas in Communications*, SAC-3(3):431–439, May 1985.

[62] Qifa Ke, David A. Maltz, and David B. Johnson. Emulation of Multi-hop Wirless Ad Hoc Networks. In *Proceedings of the 7th International Workshop on Mobile Multimedia Communications (MoMuC 2000)*, October 2000.

[63] Young-Bae Ko and Nitin H. Vaidya. Location-Aided Routing (LAR) in Mobile Ad Hoc Networks. In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pages 66–75, Dallas, TX, USA, October 1998.

[64] Gregory S. Lauer. Packet-Radio Routing. In *Routing in Communications Networks*, edited by Martha E. Steenstrup, chapter 11, pages 351–396. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[65] Barry M. Leiner, Robert J. Ruth, and Ambatipudi R. Sastry. Goals and Challenges of the DARPA GloMo Program. *IEEE Personal Communications*, 3(6):34–43, December 1996.

[66] Jian Liu and Suresh Singh. ATCP: TCP for Mobile Ad Hoc Networks. Personal Communication, June 1999.

[67] Haiyun Luo, Paul Medvedev, Jerry Cheng, and Songwu Lu. A Self-Coordinating Approach to Distributed Fair Queueing in Ad Hoc Wireless Networks. In *Proceedings of IEEE INFOCOM 2001*, pages 1370–1379, Anchorage, Alaska, April 2001.

[68] David A. Maltz. *The CMU Monarch Project's ad-hockey Visualization Tool For ns-2 Scenarios and Trace Files*. The CMU Monarch Project, monarch@monarch.cs.cmu.edu, snapshot release 1.1.0 edition, November 1998.

[69] David A. Maltz. *The CMU Monarch Project's Wireless and Mobility Extensions to ns*. The CMU Monarch Project, monarch@monarch.cs.cmu.edu, snapshot release 1.1.1 edition, August 1999.

[70] David A. Maltz. Resource Management in Multi-hop Ad Hoc Networks. Technical Report CMU CS TR00-150, School of Computer Science, Carnegie Mellon University, November 1999. Available from http://www.monarch.cs.cmu.edu/papers.html.

[71] David A. Maltz, Josh Broch, and David B. Johnson. Experiences Designing and Building a Multi-Hop Wireless Ad Hoc Network Testbed. Technical Report 99-116, School of Computer Science, Carnegie Mellon University, March 1999. Available from http://www.monarch.cs.cmu.edu/papers.html.

[72] David A. Maltz, Jan Harkes, Robert V. Baron, M. Satyanarayanan, David B. Johnson, Brian D. Noble, and Dushyanth Narayanan. Evaluating Coda & DSR using Network Emulation. Video of a demonstration given in August, 1999. Available from the Computer Science Department, Carnegie Mellon University, November 1999.

[73] A Short History of the Marconi Trans-Atlantic Receiving Station in Louisbourg. http://fortress.uccb.ns.ca/marconi/marconi2.htm, August 1999.

[74] S. McCanne. The Berkeley Packet Filter man page. BPF distribution available at ftp://ftp.ee.lbl.gov, May 1991.

[75] Jeffrey P. Monks, Vaduvur Bharghavan, and Wen-Mei W. Hwu. A Power Controlled Multiple Access Protocol for Wireless Packet Networks. In *Proceedings of IEEE INFOCOM 2001*, pages 219–228, Anchorage, Alaska, April 2001.

[76] National Science Foundation. Research Priorities in Wireless and Mobile Communications and Networking: Report of a Workshop Held March 24–26, 1997, Airlie House, Virginia. Available at http://www.cise.nsf.gov/anir/ww.html.

[77] Brian Noble, M. Satyanarayanan, Giao Nguyen, and Randy Katz. Trace-Based Mobile Network Emulation. In *Proceedings of SIGCOMM '97*, pages 51–61, September 1997.

[78] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, Eric J. Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 276–287, St. Malo, France, October 1997.

[79] Vincent D. Park and M. Scott Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proceedings of INFOCOM'97*, pages 1405–1413, April 1997.

[80] Vincent D. Park and M. Scott Corson. A Performance Comparison of TORA and Ideal Link State Routing. In *Proceedings of IEEE Symposium on Computers and Communication '98*, June 1998.

[81] Vincent D. Park and M. Scott Corson. Temporally-Ordered Routing Algorithm (TORA) Version 1: Functional Specification. Internet-Draft, draft-ietf-manet-tora-spec-01.txt, August 1998. Work in progress.

[82] Charles Perkins, editor. IP Mobility Support. Internet Request For Comments RFC 2002, October 1996.

[83] Charles E. Perkins and Pravin Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proceedings of the SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, August 1994. A revised version of the paper is available from http://www.cs.umd.edu/projects/mcml/pub.html.

[84] Charles E. Perkins and Elizabeth M. Royer. Ad Hoc On Demand Distance Vector (AODV) Routing. Internet-Draft, draft-ietf-manet-aodv-02.txt, November 1998. Work in progress.

[85] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc On-Demand Distance Vector Routing. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, pages 90–100, New Orleans, LA, February 1999.

[86] Charles E. Perkins, Elizabeth M. Royer, and Samir Das. Ad Hoc On Demand Distance Vector (AODV) Routing. Internet-Draft, draft-ietf-manet-aodv-07.txt, November 2000. Work in progress.

[87] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading, Massachusetts, 1992.

[88] David C. Plummer. An Ethernet Address Resolution Protocol: Or Converting Network Protocol Address to 48.bit Ethernet Addresses for Transmission on Ethernet Hardware. RFC 826, November 1982.

[89] J. Postel, editor. Internet Protocol. RFC 791, September 1981.

[90] J. Postel, editor. Transmission Control Protocol. RFC 793, September 1981.

[91] The CMU Monarch Project. http://www.monarch.cs.cmu.edu/. Computer Science Department, Carnegie Mellon University.

[92] Ratish J. Punnoose, Pavel V. Nikitin, Josh Broch, and Daniel D. Stancil. Optimizing Wireless Network Protocols Using Real-Time Predictive Propagation Modeling. In *Radio and Wireless Conference (RAWCON)*, Denver, CO, August 1999.

[93] S. Ramanathan and Martha Steenstrup. A Survey of Routing Techniques for Mobile Communications Networks. *ACM/Baltzer Mobile Networks and Applications*, 1(2):89–103, 1996.

[94] Theodore S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall, New Jersey, 1996.

[95] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700, October 1994.

[96] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1), January 1997.

[97] Soumya Roy and J. J. Garcia-Luna-Aceves. Using Minimal Source Trees for On-Demand Routing in Ad Hoc Networks. In *Proceedings of IEEE INFOCOM 2001*, pages 1172–1181, Anchorage, Alaska, April 2001.

[98] Elizabeth Royer and C-K. Toh. A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks. *IEEE Personal Communications*, April 1999. Read from http://users.ece.gatech.edu:80/ cktoh/ad.html.

[99] Elizabeth M. Royer and Charles E. Perkins. An Implemention Study of the AODV Routing Protocol. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC) 2000*, September 2000. To appear.

[100] R. Ruppe, S. Griswald, P. Walsh, and R. Martin. Near Term Digital Radio (NTDR) System. In *Proceedings of the 1997 IEEE Military Communications Conference (MILCOM'97)*, Monterey, CA, November 1997.

[101] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[102] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 135–146, dec 1997.

[103] Prasun Sinha, Raghupathy Sivakumar, and Vaduvur Bharghavan. Enhancing Ad Hoc Routing with Dynamic Virtual Infrastructures. In *Proceedings of IEEE INFOCOM 2001*, pages 1763–1772, Anchorage, Alaska, April 2001.

[104] W. Richard Stevens. *TCP/IP IIlustrated, Volume 1: The Protocols*. Addison-Welsley, 1994.

[105] Andrew S. Tannenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.

[106] C-K Toh and Vasos Vassiliou. The Effects of Beaconing on the Battery Life of Ad Hoc Mobile Computers. In *Ad Hoc Networking*, edited by Charles E. Perkins, chapter 9, pages 299–321. Addison-Wesley, 2001.

[107] Bruce Tuch. Development of WaveLAN, an ISM Band Wireless LAN. *AT&T Technical Journal*, 72(4):27–33, July/August 1993.

[108] Roger Wattenhofer, Li Li, Paramvir Bahl, and Yi-Min Wang. Distributed Topology Control for Power Efficient Operation in Multihop Wireless Ad Hoc Networks. In *Proceedings of IEEE INFOCOM 2001*, pages 1388–1397, Anchorage, Alaska, April 2001.

[109] *Webster's Ninth New Collegiate Dictionary*. Merriam-Webster, 1987.

[110] B. Welsh, N. Rehn, B. Vincent, J. Weinstein, and S. Wood. Multicasting with the Near Term Digital Radio (NTDR) in the Tactical Internet. In *Proceedings of the 1998 IEEE Military Communications Conference (MILCOM'98)*, Boston, MA, October 1998.

[111] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, Massachusetts, 1995.

Appendix A

Specification of the Dynamic Source Routing Protocol

The Internet Engineering Task Force (IETF) [44] is the body responsible for developing standards for the Internet. The IETF is divided into Working Groups, each Working Group being chartered by the IETF to address a specific issue. The Mobile Ad Hoc Network (MANET) working group was chartered in 1997 to propose for standardization protocols for routing in mobile ad hoc networks. The working documents of the IETF are called Internet Drafts, and, through historical accident, the final standards documents are called Request for Comments (RFCs).

This appendix contains the text of an Internet Draft submitted to the MANET working group on October 22, 1999 that defines the Dynamic Source Routing Protocol as it existed at that time. This text is the third revision of the document, and it is the best available description of the version of DSR used in this thesis. Since, at that time, DSR was being constantly evolved as a result of our experiments, the document captures only a snapshot of the protocol features that were deemed ready for public dissemination.

IETF MANET Working Group
INTERNET-DRAFT

Josh Broch
David B. Johnson
David A. Maltz
Carnegie Mellon University
22 October 1999

The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks

<draft-ietf-manet-dsr-03.txt>

Abstract

Dynamic Source Routing (DSR) is a routing protocol designed specifically for use in mobile ad hoc networks. The protocol allows nodes to dynamically discover a source route across multiple network hops to any destination in the ad hoc network. When using source routing, each packet to be routed carries in its header the complete, ordered list of nodes through which the packet must pass. A key advantage of source routing is that intermediate hops do not need to maintain routing information in order to route the packets they receive, since the packets themselves already contain all of the necessary routing information. This, coupled with the dynamic, on-demand nature of DSR's Route Discovery, completely eliminates the need for periodic router advertisements and link status packets, significantly reducing the overhead of DSR, especially during periods when the network topology is stable and these packets serve only as keep-alives.

A.1. Introduction

This document describes Dynamic Source Routing (DSR) [49, 51], a protocol developed by the Monarch Project [53, 91] at Carnegie Mellon University for routing packets in a mobile ad hoc network [22].

Source routing is a routing technique in which the sender of a packet determines the complete sequence of nodes through which to forward the packet; the sender explicitly lists this route in the packet's header, identifying each forwarding "hop" by the address of the next node to which to transmit the packet on its way to the destination node.

DSR offers a number of potential advantages over other routing protocols for mobile ad hoc networks. First, DSR uses no periodic routing messages of any kind (e.g., no router advertisements and no link-level neighbor status messages), thereby significantly reducing network bandwidth overhead, conserving battery power, reducing the probability of packet collision, and avoiding the propagation of potentially large routing updates throughout the ad hoc network. Our Dynamic Source Routing protocol is able to adapt quickly to changes such as node movement, yet requires no routing protocol overhead during periods in which no such changes occur.

In addition, DSR has been designed to compute correct routes in the presence of asymmetric (uni-directional) links. In wireless networks, links may at times operate asymmetrically due to sources of interference, differing radio or antenna capabilities, or the intentional use of asymmetric communication technology such as satellites. Due to the existence of asymmetric links, traditional link-state or distance vector protocols may compute routes that do not work. DSR, however, will always find a correct route even in the presence of asymmetric links.

A.2. Changes

Changes from version 02 to version 03 (10/1999)

- Added description of path-state and flow-state maintenance (Section A.10). These extensions remove the need for every data packet to carry a source route, thereby decreasing the byte-overhead of DSR. They also provide a framework for supporting QoS inside DSR networks.

A.3. Assumptions

We assume that all nodes wishing to communicate with other nodes within the ad hoc network are willing to participate fully in the protocols of the network. In particular, each node participating in the network should also be willing to forward packets for other nodes in the network.

We refer to the minimum number of hops necessary for a packet to reach from any node located at one extreme edge of the network to another node located at the opposite extreme, as the *diameter* of the network. We assume that the diameter of an ad hoc network will be small (e.g., perhaps 5 or 10 hops), but may often be greater than 1.

Packets may be lost or corrupted in transmission on the wireless network. A node receiving a corrupted packet can detect the error and discard the packet.

We assume that nodes can enable *promiscuous* receive mode on their wireless network interface hardware, causing the hardware to deliver every received packet to the network driver software without filtering based on link-layer destination address. Although we do not require this facility, it is for example common in current LAN hardware for broadcast media including wireless, and some of our optimizations take advantage of its availability. Use of promiscuous mode does increase the software overhead on the CPU, but we believe that wireless network speeds are more the inherent limiting factor to performance in current and future systems. We also believe that portions of the protocol are also suitable for implementation directly within a programmable network interface unit to avoid this overhead on the CPU.

A.4. Terminology

A.4.1. General Terms

link A communication facility or medium over which nodes can communicate at the link layer, such as an Ethernet (simple or bridged). A link is the layer immediately below IP.

interface A node's attachment to a link.

prefix A bit string that consists of some number of initial bits of an address.

interface index An 7-bit quantity which uniquely identifies an interface among a given node's interfaces. Each node can assign interface indices to its interfaces using any scheme it wishes.

The index IF_INDEX_MA is reserved for use by Mobile IP [82] mobility agents (home or foreign agents) to indicate that they believe they can reach a destination via a connected internet infrastructure. The index IF_INDEX_ROUTER is reserved for use by routers not acting as Mobile IP mobility agents to indicate that they believe they can reach the destination via a connected internet infrastructure.

The distinction between the index for mobility agents and the index for routers, allows mobility agents to advertise their existence "for free". A node that processes a routing header listing the interface index IF_INDEX_MA, can then send a unicast Agent Solicitation to the corresponding address in the routing header to obtain complete information about the mobility services being provided.

link-layer address A link-layer identifier for an interface, such as IEEE 802 addresses on Ethernet links.

packet An IP header plus payload.

piggybacking Including two or more conceptually different types of data in the same packet so that all data elements move through the network together.

home address An IP address that is assigned for an extended period of time to a mobile node. It remains unchanged regardless of where the node is attached to the Internet [82]. If a node has more than one home address, it SHOULD select and use a single home address when participating in the ad hoc network.

source route A source route from a node **S** to some node **D** is an ordered list of home addresses and interface indexes that contains all the information that would be needed to forward a packet through the ad hoc network. For each node that will transmit the packet, the source route provides the index of the interface over which the packet should be transmitted, and the address of the node which is intended to receive the packet.

DSR Routing Headers as described in Section A.7.3 use a more compact encoding of the source route and do not explicitly list address **S** in the Routing Header‘, since it is carried as the IP Source Address of the packet.

A source route is described as “broken” when the specific path it describes through the network is not actually viable.

Route Discovery The method in DSR by which a node **S** dynamically obtains a source route to some node **D** that will be used by **S** to route packets through the network to **D**. Performing a Route Discovery involves sending one or more Route Request packets.

Route Maintenance The process in DSR of monitoring the status of a source route while in use, so that any link-failures along the source route can be detected and the broken link removed from use.

A.4.2. Specification Language

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [14].

A.5. Protocol Overview

A.5.1. Route Discovery and Route Maintenance

A source routing protocol must solve two challenges, which DSR terms *Route Discovery* and *Route Maintenance*. Route Discovery is the mechanism whereby a node **S** wishing to send a packet to a destination **D** obtains a source route to **D**.

Route Maintenance is the mechanism whereby **S** is able to detect, while using a source route to **D**, if the network topology has changed such that it can no longer use its route to **D** because a link along the route no longer works. When Route Maintenance indicates a source route is broken, **S** can attempt to use any other route it happens to know to **D**, or can invoke Route Discovery again to find a new route.

To perform Route Discovery, the source node **S** link-layer broadcasts a Route Request packet. Here, node **S** is termed the initiator of the Route Discovery, and the node to which **S** is attempting to discover a source route, say **D**, is termed the target of the Discovery.

Each node that hears the Route Request packet forwards a copy of the Request, if appropriate, by adding its own address to a source route being recorded in the Request packet and then rebroadcasting the Route Request.

The forwarding of Route Requests is constructed so that copies of the Request propagate hop-by-hop outward from the node initiating the Route Discovery, until either the target of the Request is found or until another node is found that can supply a route to the target.

The basic mechanism of forwarding Route Requests forwards the Request if the node (1) is not the target of the Request, (2) is not already listed in the recorded source route in this copy of the Request, and (3) has not recently seen another Route Request packet belonging to this same Route Discovery. A node can determine if it has recently seen such a Route Request, since each Route Request packet contains a unique identifier for this Route Discovery, generated by the initiator of the Discovery. Each node maintains an LRU cache of the unique identifier from each recently received Route Request. By not propagating any copies of a Request after the first, the overhead of forwarding additional copies that reach this node along different paths is avoided.

In addition, the Time-to-Live field in the IP header of the packet carrying the Route Request MAY be used to limit the scope over which the Request will propagate, using the normal behavior of Time-to-Live defined by IP [89, 13]. Additional optimizations on the handling and forwarding of Route Requests are also used to further reduce the Route Discovery overhead.

When the target of the Request (e.g., node **D**) receives the Route Request, the recorded source route in the Request identifies the sequence of hops over which this copy of the Request reached **D**. Node **D** copies this recorded source route into a Route Reply packet and sends this Route Reply back to the initiator of the Route Request (e.g., node **S**).

All source routes learned by a node are kept in a *Route Cache*, which is used to further reduce the cost of Route Discovery. When a node wishes to send a packet, it examines its own Route Cache and performs Route Discovery only if no suitable source route is found in its Cache.

Further, when some intermediate node **B** receives a Route Request from **S** for some target node **D**, **B** not equal **D**, **B** searches its own Route Cache for a route to **D**. If **B** finds such a route, it might not have to propagate the Route Request, but instead return a Route Reply to node **S** based on the concatenation of the recorded source route from **S** to **B** in the Route Request and the cached route from **B** to **D**. The details of replying from a Route Cache in this way are discussed in Section A.9.1.

As a node overhears routes being used by others, either on data packets or on control packets used by Route Discovery or Route Maintenance, the node MAY insert those routes into its Route Cache, leveraging the Route Discovery operations of the other nodes in the network. Such route information MAY be learned either by promiscuously snooping on packets or when forwarding packets.

A.5.2. Packet Forwarding

To represent a source route within a packet's header, DSR uses a Routing Header similar to the Routing Header format specified for IPv6, adapted to the needs of DSR and to the use of DSR in IPv4 (or in IPv6 in the future). The DSR Routing Header uses a unique Routing Type field value to distinguish it from the existing Type 0 Routing Header defined within IPv6 [25].

To forward a packet, a receiving node **N** simply processes the Routing Header as specified in Section A.8.3 and transmits the packet to the next hop. If a forwarding error occurs along the link to the next hop in the route, this node **N** sends a Route Error back to the originator **S** of this packet informing **S** that this link is "broken". If node **N**'s Route Cache contains a different route to the destination of the original packet, then the packet is salvaged using the new source route (Section A.8.5). Otherwise, the packet is dropped.

Each node overhearing or forwarding a Route Error packet also removes from its Route Cache the link indicated to be broken, thereby cleaning the stale cache data from the network.

A.5.3. Multicast Routing

At this time DSR does not support true multicasting. However, it does support the controlled flooding of a data packet to all nodes in the network that are within some number of hops of the originator. While this mechanism does not support pruning of the broadcast tree to conserve network resources, it can be used to distribute information to nodes in the network.

When an application on a DSR node sends a packet to a multicast address, DSR piggybacks the data from the packet inside a Route Request packet targeted at the multicast address. The normal Route Request distribution scheme described in Sections A.5.1 and A.8.4 will result in this packet being efficiently distributed to all nodes in the network within the specified TTL of the originator. The receiving nodes can then do destination address filtering on the packet, discarding it if they do not wish to receive multicast packets destined to this multicast address.

A.6. Conceptual Data Structures

In order to participate in the Dynamic Source Routing Protocol, a node needs four conceptual data structures: a Route Cache, a Route Request Table, a Send Buffer, and a Retransmission Buffer. These data structures MAY be implemented in any manner consistent with the external behavior described in this document.

A.6.1. Route Cache

All routing information needed by a node participating in an ad hoc network using DSR is stored in a *Route Cache*. Each node in the network maintains its own Route Cache. The node adds information to the Cache as it learns of new links between nodes in the ad hoc network, for example through packets carrying either a Route Reply or a Routing Header. Likewise, the node removes information from the cache as it learns existing links in the ad hoc network have broken, for example through packets carrying a Route Error or through the link-layer retransmission mechanism reporting a failure in forwarding a packet to its next-hop destination. The Route Cache is indexed logically by destination node address, and supports the following operations:

void Insert(Route RT) Inserts information extracted from source route RT into the Route Cache.

Route Get(Node DEST) Returns a source route from this node to DEST (if one is known).

void Delete(Node FROM, Interface INDEX, Node TO) Removes from the route cache any routes which assume that a packet transmitted by node FROM over its interface with the given INDEX will be received by node TO.

Each implementation MAY choose the cache replacement and cache search strategies for its Route Cache that are most appropriate for its particular network environment. For example, some environments may choose to return the shortest route to a node (the shortest sequence of hops), while others may select an alternate metric for the Get() operation.

The Route Cache SHOULD support storing more than one source route for each destination.

If there are multiple cached routes to a destination, the Route Get() operation SHOULD prefer routes that do not traverse a hop with an interface index of IF_INDEX_MA or IF_INDEX_ROUTER. This will prefer routes that lead directly to the target node over routes that attempt to reach the target via any internet infrastructure connected to the ad hoc network.

If a node **S** is using a source route to some destination **D** that includes intermediate node **N**, **S** SHOULD shorten the route to destination **D** when it learns of a shorter route to node **N** than the one that is listed as the prefix of its current route to **D**.

A node **S** using a source route to destination **D** through intermediate node **N**, MAY shorten the source route if it learns of a shorter path from node **N** to node **D**.

The Route Cache replacement policy SHOULD allow routes to be categorized based upon "preference", where routes with a higher preferences are less likely to be removed from the cache. For example, a node could prefer routes for which it initiated a Route Discovery over routes that it learned as the result of promiscuous snooping on other packets. In particular, a node SHOULD prefer routes that it is presently using over those that it is not.

A.6.2. Route Request Table

The Route Request Table is a collection of records about Route Request packets that were recently originated or forwarded by this node. The table is indexed by the home address of the target of the route discovery. A record maintained on node **S** for node **D** contains the following:

- The time that **S** last originated a Route Discovery for **D**.
- The remaining amount of time that **S** must wait before the next attempt at a Route Discovery for **D**.
- The Time-to-live (TTL) field in the IP header of last Route Request originated by **S** for **D**.
- A FIFO cache of the last ID_FIFO_SIZE Identification values from Route Request packets targeted at node **D** that were forwarded by this node.

Nodes SHOULD use an LRU policy to manage the entries of in their Route Request Table.

ID_FIFO_SIZE MUST NOT be set to an unlimited value, since, in the worst case, when a node crashes and reboots the first ID_FIFO_SIZE Route Request packets it sends may appear to be duplicates to the other nodes in the network.

A.6.3. Send Buffer

The Send Buffer of some node is a queue of packets that cannot be transmitted by that node because it does not yet have a source route to each respective packet's destination. Each packet in the Send Buffer is stamped with the time that it is placed into the Buffer, and SHOULD be removed from the Send Buffer and discarded SEND_BUFFER_TIMEOUT seconds after initially being placed in the Buffer. If necessary, a FIFO strategy SHOULD be used to evict packets before they timeout to prevent the buffer from overflowing.

Subject to the rate limiting defined in Section A.8.4, a Route Discovery SHOULD be initiated as often as possible for the destination address of any packets residing in the Send Buffer.

A.6.4. Retransmission Buffer

The Retransmission Buffer of a node is a queue of packets sent by this node that are awaiting the receipt of an acknowledgment from the next hop in the source route (Section A.7.3).

For each packet in the Retransmission Buffer, a node maintains (1) a count of the number of retransmissions and (2) the time of the last retransmission.

Packets are removed from the buffer when an acknowledgment is received, or when the number of retransmissions exceeds DSR_MAXRXTSHIFT. In the later case, the removal of the packet from the Retransmission Buffer SHOULD result in a Route Error being returned to the initial source of the packet (Section A.8.5).

A.7. Packet Formats

Dynamic Source Routing makes use of four options carrying control information that can be piggybacked in any existing IP packet.

The mechanism used for these options is based on the design of the Hop-by-Hop and Destination Options mechanisms in IPv6 [25]. The ability to generate and process such options must be added to an IPv4 protocol stack. Specifically, the Protocol field in the IP header is used to indicate that a Hop-by-Hop Options or Destination Options extension header exists between the IP header and the remaining portion of a packet's payload (such as a transport layer header). The Next Header field in each extension header will then indicate the type of header that follows it in a packet.

A.7.1. Destination Options Headers

The Destination Options header is used to carry optional information that need be examined only by a packet's destination node(s). The Destination Options header is identified by a Next Header (or Protocol) value of 60 in the immediately preceding header, and has the following format:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  |  Hdr Ext Len  |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               +
|                                                               |
.                                                               .
.                            Options                            .
.                                                               .
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Next Header 8-bit selector. Identifies the type of header immediately following the Destination Options header. Uses the same values as the IPv4 Protocol field [95].

Hdr Ext Len 8-bit unsigned integer. Length of the Destination Options header in 4-octet units, not including the first 8 octets.

Options Variable-length field, of length such that the complete Destination Options header is an integer multiple of 4 octets long. Contains one or more TLV-encoded options.

The following destination option is used by the Dynamic Source Routing protocol:

- DSR Route Request option (Section A.7.1)

This destination option MUST NOT appear multiple times within a single Destination Options header.

DSR Route Request Option

The DSR Route Request destination option is encoded in type-length-value (TLV) format as follows:

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |  Option Type  | Option Length |         Identification        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                         Target Address                        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |C| IN Index[1] |C| IN Index[2] |C| IN Index[3] |C| IN Index[4] |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |C|OUT Index[1] |C|OUT Index[2] |C|OUT Index[3] |C|OUT Index[4] |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            Address[1]                         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            Address[2]                         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            Address[3]                         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            Address[4]                         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |C| IN Index[5] |C| IN Index[6] |C|  IN Index[7] |C| IN Index[8]|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |C|OUT Index[5] |C|OUT Index[6] |C| OUT Index[7] |C|OUT Index[8]|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            Address[5]                         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                               ...                             |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

IP fields:

Source Address MUST be the home address of the node originating this packet. Intermediate nodes that repropagate the request do not change this field.

Destination Address MUST be the limited broadcast address (255.255.255.255).

Hop Limit (TTL) Can be varied from 1 to 255, for example to implement expanding-ring searches.

Route Request fields:

Option Type Value: TBD. A node that does not understand this option MUST discard the packet and the Option Data may change en-route (the top three bits are 011).

Option Length 8-bit unsigned integer. Length of the option, in octets, excluding the Option Type and Option Length fields.

Identification A unique value generated by the initiator (original sender) of the Route Request. This value allows a recipient to determine whether or not it has recently seen this a copy of this Request; if it has, the packet is simply discarded. When propagating a Route Request, this field MUST be copied from the received copy of the Request being forwarded.

Target Address The home address of the node that is the target of the Route Request.

Change Interface (C) bit[1..n] A flag associated with each interface index that indicates whether or not the corresponding node repropagated the Request over a different physical interface type than over which it received the Request.

IN Index[1..n] IN Index[i] is the index of the interface over which the node indicated by Address[i] received the Route Request option. These are used to record a reverse route from the target of the request to the originator, over which a Route Reply MAY be sent.

OUT Index[1..n] OUT Index[i] is the interface index that the node indicated by Address[i-1] used when rebroadcasting the Route Request option.

Address[1..n] Address[i] is the home address of the ith hop recorded in the Route Request option.

A.7.2. Hop-by-Hop Options Headers

The Hop-by-Hop Options header is used to carry optional information that must be examined by every node along a packet's delivery path. The Hop-by-Hop Options header is identified by a Next Header (or Protocol) value of TBD in the IP header, and has the following format:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  |  Hdr Ext Len  |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               +
|                                                               |
.                                                               .
.                            Options                            .
.                                                               .
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Next Header 8-bit selector. Identifies the type of header immediately following the Hop-by-Hop Options header. Uses the same values as the IPv4 Protocol field [95].

Hdr Ext Len 8-bit unsigned integer. Length of the Hop-by-Hop Options header in 4-octet units, not including the first 8 octets.

Options Variable-length field, of length such that the complete Hop-by-Hop Options header is an integer multiple of 4 octets long. Contains one or more TLV-encoded options.

The following hop-by-hop options are used by the Dynamic Source Routing protocol:

- DSR Route Reply option (Section A.7.2)
- DSR Route Error option (Section A.7.2)
- DSR Acknowledgment option (Section A.7.2)

All of these destination options MAY appear one or more times within a single Hop-by-Hop Options header.

DSR Route Reply Option

The DSR Route Reply hop-by-hop option is encoded in type-length-value (TLV) format as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                |  Option Type  | Option Length |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Target Address                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|C|OUT Index[1] |C|OUT Index[2] |C|OUT Index[3] |C|OUT Index[4] |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[1]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[2]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[3]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[4]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|C|OUT Index[5] |C|OUT Index[6] |C|OUT Index[7] |C|OUT Index[8] |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[5]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              ...                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Option Type Value: TBD. A node that does not understand this option should ignore this option and continue processing the packet, and the Option Data does not change en-route (the top three bits are 000).

Option Length 8-bit unsigned integer. Length of the option, in octets, excluding the Option Type and Option Length fields.

Reserved Sent as 0; ignored on reception.

Target Address The home address of the node to which the Route Reply must be delivered.

Change Interface (C) bit[1..n] If the C bit associated with a node **N** is set, it implies **N** will be forwarding the packet out a different interface than the one over which it was received (i.e., the node sending the packet to **N** should not expect a passive acknowledgment).

OUT Index[1..n] OUT Index[i] is the interface index of the ith hop listed in the Route Reply option. It denotes the interface that should be used by Address[i-1] to reach Address[i] when using the specified source route.

Address[1..n] Address[i] is the home address of the ith hop listed in the Route Reply option.

DSR Route Error Option

The DSR Route Error hop-by-hop option is encoded in type-length-value (TLV) format as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Option Type  | Option Length |     Index     |   Error Type  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Error Source Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Error Destination Address                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Unreachable Node Address                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Option Type Value: TBD. A node that does not understand this option should ignore the option and continue processing the packet, and the Option Data does not change en-route (the top three bits are 000).

Option Length 8-bit unsigned integer. Length of the option, in octets, excluding the Option Type and Option Length fields.

Index The interface index of the network interface over which the node designated by Error Source Address tried to forward a packet to the node designated by Unreachable Node Address.

Error Type The type of error encountered. Values between 0 and 127 inclusive indicate a hard failure of connectivity between the Error Source Address and the Unreachable Node Address. Values between 128 and 255 inclusive indicate a soft failure.

```
NODE_UNREACHABLE                1

PATH_STATE_NOT_FOUND            129
```

Error Source Address The home address of the node originating the Route Error (e.g., the node that attempted to forward a packet and discovered the link failure).

Error Destination Address The home address of the node to which the Route Error must be delivered (e.g, the node that generated the routing information claiming that the hop Error Source Address to Unreachable Node Address was a valid hop).

Unreachable Node Address The home address of the node that was found to be unreachable (the next hop neighbor to which the node at “Error Source Address” was attempting to transmit the packet).

DSR Acknowledgment Option

The DSR Acknowledgment hop-by-hop option is encoded in type-length-value (TLV) format as follows:

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                                   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                   |  Option Type  | Option Length |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                         Identification                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       ACK Source Address                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                     ACK Destination Address                   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       Data Source Address                     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Option Type Value: TBD. A node that does not understand this option should ignore the option and continue processing the packet, and the Option Data does not change en-route (the top three bits are 000).

Option Length 8-bit unsigned integer. Length of the option, in octets, excluding the Option Type and Option Length fields.

Identification A 32-bit value that when taken in conjunction with Data Source Address, uniquely identifies the packet being acknowledged.

The Identification value is computed as ((ip_id << 16) | ip_off) where ip_id is the value of the 16-bit Identification field in the IP header of the packet being acknowledged, and ip_off is the value of the 13-bit Fragment Offset field in the IP header of the packet being acknowledged.

When constructing the Identification, ip_id and ip_off MUST be in host byte-order. The entire Identification value MUST then be converted to network byte-order before being placed in the Acknowledgment option.

ACK Source Address The home address of the node originating the Acknowledgment.

ACK Destination Address The home address of the node to which the Acknowledgment must be delivered.

Data Source Address The IP Source Address of the packet being acknowledged.

A.7.3. DSR Routing Header

As specified for IPv6 [25], a Routing header is used by a source to list one or more intermediate nodes to be "visited" on the way to a packet's destination. This function is similar to IPv4's Loose Source and Record Route option, but the Routing header does not record the route taken as the packet is forwarded. The specific processing steps required to implement the Routing header must be added to an IPv4 protocol stack. The Routing header is identified by a Next Header value of 43 in the immediately preceding header, and has the following format:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  |  Hdr Ext Len  |  Routing Type | Segments Left |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
.                                                               .
.                       type-specific data                      .
.                                                               .
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The type specific data for a Routing Header carrying a DSR Source Route is:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|S|                        Reserved                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|C|OUT Index[1] |C|OUT Index[2] |C|OUT Index[3] |C|OUT Index[4] |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Address[1]                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Address[2]                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Address[3]                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Address[4]                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|C|OUT Index[5] |C|OUT Index[6] |C|OUT Index[7] |C|OUT Index[8] |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Address[5]                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               ...                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Routing Header Fields:

Next Header 8-bit selector. Identifies the type of header immediately following the Routing header.

Hdr Ext Len 8-bit unsigned integer. Length of the Routing header in 4-octet units, not including the first 8 octets.

Routing Type Value: TBD

Segments Left Number of route segments remaining, i.e., number of explicitly listed intermediate nodes still to be visited before reaching the final destination.

Type Specific Fields:

Acknowledgment Request (R) The Acknowledgment Request (R) bit is set to request an explicit acknowledgment from the next hop. After processing the Routing Header, The IP Destination Address lists the address of the next hop.

Salvaged Packet (S) The Salvaged Packet (S) bit indicates that this packet has been salvaged by an intermediate node, and thus that this Routing Header was generated by Address[1] and not the IP Source Address (Section A.8.5).

Reserved Sent as 0; ignored on reception.

Change Interface (C) bit[1..n] If the C bit associated with a node **N** is set, it implies **N** will be forwarding the packet out a different interface than the one over which it was received (i.e., the node sending the packet to **N** should not expect a passive acknowledgment and MAY wish to set the R bit).

OUT Index[1..n] Index[i] is the interface index that the node indicated by Address[i-1] must use when transmitting the packet to Address[i]. Index[1] indicates which interface the node indicated by the IP Source Address uses to transmit the packet.

Address[1..n] Address[i] is the home address of the ith hop in the Routing header.

Note that Address[1] is the first intermediate hop along the route. The address of the originating node is the IP Source Address. The only exception to this rule is for packets that are salvaged, as described in Section A.8.5. A packet that has been salvaged has an alternate route placed on it by an intermediate node in the network, and in this case, the address of the originating node (the salvaging node) is Address[1]. Salvaged packets are indicated by setting the S bit in the DSR Routing header.

A.8. Detailed Operation

A.8.1. Originating a Data Packet

When node **A** originates a packet, the following steps MUST be taken before transmitting the packet:

1. If the destination address is a multicast address, piggyback the data packet on a Route Request targeting the multicast address. The following fields MUST be initialized as specified:

   ```
   IP.Source_Address      = Home address of node A
   IP.Destination_Address = 255.255.255.255
   Request.Target_Address = Multicast destination address
   ```

   DONE.

2. Otherwise, call Route_Cache.Get() to determine if there is a cached source route to the destination.

3. If the cached route indicates that the destination is directly reachable over one hop, no Routing Header should be added to the packet. Initialize the following fields:

   ```
   IP.Source_Address      = Home address of node A
   IP.Destination_Address = Home address of the Destination
   ```

   DONE.

4. Otherwise, if the cached route indicates that multiple hops are required to reach the destination, insert a Routing Header into the packet as described in Section A.8.2. DONE.

5. Otherwise, if no cached route to the destination is found, insert the packet into the Send Buffer and initiate Route Discovery as described in Section A.8.4.

A.8.2. Originating a Packet with a DSR Routing Header

When a node originates a packet with a Routing Header, the address of the first hop in the source route MUST be listed as the IP Destination Address as well as Address[1] in the Routing Header. The final destination of the packet is listed as the last hop in the Routing Header (Address[n]). At each intermediate hop i, Address[i] is copied into the IP Destination Address and the packet is retransmitted.

For example, suppose node **A** originates a packet destined for node **D** that should pass through intermediate hops **B** and **C**. The packet MUST be initialized as follows:

```
IP.Source_Address      = Home address of node A
IP.Destination_Address = Home address of node B
RT.Segments_Left       = 2
RT.Out_Index[1]        = Interface index used by A to reach B
RT.Out_Index[2]        = Interface index used by B to reach C
RT.Out_Index[3]        = Interface index used by C to reach D
RT.Address[1]          = Home address of node B
RT.Address[2]          = Home address of node C
RT.Address[3]          = Home address of node D
```

A.8.3. Processing a Routing Header

Excluding the exceptions listed here, a DSR Routing Header is processed using the same rules as outlined for Type 0 Routing Headers in IPv6 [25]. The Routing Header is only processed by the node whose address appears as the IP destination of the packet. The following additional rules apply to processing the type specific data of a DSR Source Route:

Let

```
SegLft = the value of Segments Left when the packet was received
NumAddrs = the total number of addresses in the Routing Header
```

1. The address of the next hop, Address[NumAddrs - SegLft + 1], is copied into the IP.Destination_Address of the packet. The existing IP.Destination_Address is NOT copied back into the Address list of the Routing Header.

2. The interface used to transmit the packet to its next hop from this node MUST be the interface denoted by Index[NumAddrs - SegLft + 1].

3. If the Acknowledgment Request (R) bit is set, the node MUST transmit a packet containing the DSR Acknowledgment option to the previous hop, Address[NumAddrs - SegLft - 1], performing Route Discovery if necessary. (Address[0] is taken as the IP.Source_Address)

4. Perform Route Maintenance by verifying that the packet was received by the next hop as described in Section A.8.5.

A.8.4. Route Discovery

Route Discovery is the on-demand process by which nodes actively obtain source routes to destinations to which they are actively attempting to send packets. The destination node for which a Route Discovery is initiated is known as the "target" of the Route Discovery. A Route Discovery for a destination SHOULD NOT be initiated unless the initiating node has a packet in the Send Buffer requiring delivery to that destination. A Route Discovery for a given target node MUST NOT be initiated unless permitted by the rate-limiting information contained in the Route Request Table. After each Route Discovery attempt, the interval between successive Route Discoveries for this target must be doubled, up to a maximum of MAX_REQUEST_PERIOD.

Route Discoveries for a multicast address SHOULD NOT be rate limited, and SHOULD always be permitted.

Originating a Route Request

The basic Route Discovery algorithm for a unicast destination is as follows:

1. Originate a Route Request packet with the IP header Time-to-Live field initialized to 1. This type of Route Request is called a non-propagating Route Request and allows the originator of the Request to inexpensively query the route caches of each of its neighbors for a route to the destination.

2. If a Route Reply is received in response to the non-propagating Request, use the returned source route to transmit all packets for the destination that are in the Send Buffer. DONE.

3. Otherwise, if no Route Reply is received within RING0_REQUEST_TIMEOUT seconds, transmit a Route Request with the IP header Time-to-Live field initialized to MAX_ROUTE_LEN. This type of Route Request is called a propagating Route Request. Update the information in the Route Request Table, to double the amount of time before any subsequent Route Discovery attempt to this target.

4. If no Route Reply is received within the time interval indicated by the Route Request Table, GOTO step 1.

The Route Request option SHOULD be initialized as follows:

```
IP.Source_Address      = This node’s home address
IP.Destination_Address = 255.255.255.255
Request.Target         = Home address of intended destination
Request.OUT_Index[1]   = Index of interface used to transmit the Request
```

The behavior of a node processing a packet containing both a Routing Header and a Route Request Destination option is unspecified. Packets SHOULD NOT contain both a Routing Header and a Route Request Destination option. [This is not exactly true: A Route Request option appearing in the second Destination Options header that IPv6 allows after the Routing Header would probably do-what-you-mean, though we have not triple-checked it yet. Namely, it would allow the originator of a route discovery to unicast the request to some other node, where it would be released and begin the flood fill. We call this a Route Request Blossom since the unicast portion of the path looks like a stem on the blossoming flood-fill of the request.]

Packets containing a Route Request Destination option SHOULD NOT be retransmitted, SHOULD NOT request an explicit DSR Acknowledgment by setting the R bit, SHOULD NOT expect a passive acknowledgment, and SHOULD NOT be placed in the Retransmission Buffer. The repeated transmission of packets containing a Route Request Destination option is controlled solely by the logic described in this section.

Processing a Route Request Option

When a node **A** receives a packet containing a Route Request option, the Route Request option is processed as follows:

1. If Request.Target_Address matches the home address of this node, then the Route Request option contains a complete source route describing the path from the initiator of the Route Request to this node.

   (a) Send a Route Reply as described in Section A.8.4.

   (b) Continue processing the packet in accordance with the Next Header value contained in the Destination Option extension header. DONE.

2. Otherwise, if the combination (IP.Source_Address, Request.Identification) is found in the Route Request Table, then discard the packet, since this is a copy of a recently seen Route Request. DONE.

3. Otherwise, if Request.Target_Address is a multicast address then:

   (a) If node **A** is a member of the multicast group indicated by Request.Target_Address, then create a copy of the packet, setting IP.Destination_Address = REQUEST.Target_Address, and continue processing the copy of the packet in accordance with the Next Header field of the Destination option.

   (b) If IP.TTL is non-zero, decrement IP.TTL, and retransmit the packet. DONE.

   (c) Otherwise, discard the packet. DONE.

4. Otherwise, if the home address of node **A** is already listed in the Route Request (IP.Source_Address or Request.Address[]), then discard the packet. DONE.

5. Let

```
m = number of addresses currently in the Route Request option
n = m + 1
```

6. Otherwise, append the home address of node **A** to the Route Request option (Request.Address[n]).
7. Set Request.IN_Index[n] = index of interface packet was received on.
8. If a source route to Request.Target_Address is found in our Route Cache and the rules of Section A.8.4 permit it, return a Cached Route Reply as described in Section A.8.4. DONE.
9. Otherwise, for each interface on which the node is configured to participate in a DSR ad hoc network:
   (a) Make a copy of the packet containing the Route Request.
   (b) Set Request.OUT_Index[n+1] = index of the interface.
   (c) If the outgoing interface is different from the incoming interface, then set the C bit on both Request.OUT_Index[n+1] and Request.IN_Index[n]
   (d) Link-layer re-broadcast the packet containing the Route Request on the interface jittered by T milliseconds, where T is a uniformly distributed, random number between 0 and BROADCAST_JITTER. DONE.

Generating Route Replies using the Route Cache

A node SHOULD use its Route Cache to avoid propagating a Route Request packet received from another node. In particular, suppose a node receives a Route Request packet for which it is not the target and which it does not discard based on the logic of Section A.8.4. If the node has a Route Cache entry for the target of the Request, it SHOULD append this cached route to the accumulated route record in the packet and return this route in a Route Reply packet to the initiator without propagating (re-broadcasting) the Route Request. Thus, for example, if node **F** in the example network shown in Figure A.8.4 needs to send a packet to node **D**, it will initiate a Route Discovery and broadcast a Route Request packet. If this broadcast is received by node **A**, node **A** can simply return a Route Reply packet to **F** containing the complete route to **D** consisting of the sequence of hops: **A**, **B**, **C**, and **D**.

Before transmitting a Route Reply packet that was generated using information from its Route Cache, a node MUST verify that:

```
B->C->D
+---+     +---+     +---+     +---+
| A |---->| B |---->| C |---->| D |
+---+     +---+     +---+     +---+

+---+
| F |                     +---+
+---+                     | E |
                          +---+
```

Figure A.1 An example network where A knows a route to D via B and C.

1. The resulting route contains no loops.
2. The node issuing the Route Reply is listed in the route that it specifies in its Reply. This increases the probability that the route is valid, since the node in question should have received a Route Error if this route stopped working. Additionally, this requirement means that a Route Error traversing the route will pass through the node that issued the Reply based on stale cache data, which is critical for ensuring stale data is removed from caches in a timely manner. Without this requirement, the next Route Discovery initiated by the original requester might also be contaminated by a Route Reply from this node containing the same stale route.

Originating a Route Reply

Let REQPacket denote a packet received by node **A** that contains a Route Request option which lists node **A** as the REQPacket.Request.Target_Address. Let REPPacket be a packet transmitted by node **A** that contains a corresponding Route Reply. The Route Reply option transmitted in response to a Route Request MUST be initialized as follows:

1. If REQPacket.Request.Address[] does not contain any hops, then node **A** is only a single hop from the originator of the Route Request. Build a Route Reply packet as follows:

   ```
   REPPacket.IP.Source_Address    = REQPacket.Request.Target_Address
   REPPacket.Reply.Target         = REQPacket.IP.Source_Address
   REPPacket.Reply.OUT_Index[1]   = REQPacket.Request.OUT_index[1]
   REPPacket.Reply.OUT_C_bit[1]   = REQPacket.Request.OUT_C_bit[1]
   REPPacket.Reply.Address[1]     = The home address of node A
   ```

   GOTO step 3.

2. Otherwise, build a Route Reply packet as follows:

   ```
   REPPacket.IP.Source_Address    = The home address of node A
   REPPacket.Reply.Target         = REQPacket.IP.Source_Address
   REPPacket.Reply.OUT_Index[1..n]= REQPacket.Request.OUT_index[1..n]
   REPPacket.Reply.OUT_C_bit[1..n]= REQPacket.Request.OUT_C_bit[1..n]
   REPPacket.Reply.Address[1..n]  = REQPacket.Request.Address[1..n]
   ```

3. Send the Route Reply jittered by T milliseconds, where T is a uniformly distributed random number between 0 and BROADCAST_JITTER. DONE.

If sending a Route Reply packet to the originator of the Route Request requires performing a Route Discovery, the Route Reply hop-by-hop option MUST be piggybacked on the packet that contains the Route Request. This prevents a loop wherein the target of the new Route Request (which was itself the originator of the original Route Request) must do another Route Request in order to return its Route Reply.

If sending the Route Reply to the originator of the Route Request does not require performing Route Discovery, a node SHOULD send a unicast Route Reply in response to every Route Request targeted at it.

Processing a Route Reply Option

Upon receipt of a Route Reply, a node should extract the source route (Target_Address, OUT_Index[1]:Address[1], .. OUT_Index[n]:Address[n]) and insert this route into its Route Cache. All the packets in the Send Buffer SHOULD be checked to see whether the information in the Reply allows them to be sent immediately.

A.8.5. Route Maintenance

Route Maintenance requires that whenever a node transmits a data packet, a Route Reply, or a Route Error, it must verify that the next hop (indicated by the Destination IP Address) correctly receives the packet.

If the sender cannot verify that the next hop received the packet, it MUST decide that its link to the next hop is broken and MUST send a Route Error to the node responsible for generating the Routing Header that contains the broken link (Section A.8.5).

The following ways may be used to verify that the next hop correctly received a packet:

- The receipt of a passive acknowledgment (Section A.8.5).
- The receipt of an explicitly requested acknowledgment (Section A.8.5).
- By the presence of positive feedback from the link layer indicating that the packet was acknowledged by the next hop (Section A.8.5).
- By the absence of explicit failure notification from the link layer that provides reliable hop-by-hop delivery such as MACAW or 802.11 (Section A.8.5).

Nodes MUST NOT perform Route Maintenance for packets containing a Route Request option or packets containing only an Acknowledgment option. Sending Acknowledgments for packets containing only an Acknowledgment option would create an infinite loop whereby acknowledgments would be sent for acknowledgments. Acknowledgments should be always sent for packets containing a Routing Header with the R bit set (e.g., packets which contain only an Acknowledgment and a Routing Header for which the last forwarding hop requires an explicit acknowledgment of receipt by the final destination).

Using Network-Layer Acknowledgments

For link layers that do not provide explicit failure notification, the following steps SHOULD be used by a node **A** to perform Route Maintenance.

When receiving a packet:

- If the packet contains a Routing Header with the R bit set, send an explicit acknowledgment as described in Section A.8.3.
- If the packet does not contain a Routing Header, the node MUST transmit a packet containing the DSR Acknowledgment option to the previous hop as indicated by the IP.Source_Address. Since the receiving node is the final destination, there will be no opportunity for the originator to obtain a passive acknowledgment, and the receiving node must infer the originator's request for an explicit acknowledgment.

When sending a packet:

1. Before sending a packet, insert a copy of the packet into the Retransmission Buffer and update the information maintained about this packet in the Retransmission Buffer.
2. If after processing the Routing Header, RH.Segments_Left is equal to 0, then node **A** MUST set the Acknowledgment Request (R) bit in the Routing Header before transmitting the packet over its final hop.

3. If after processing the Routing Header and copying RH.Address[n] to IP.Destination_Address, node **A** determines that RH.OUT_C_bit[n+1] is set, then node **A** MUST set the Acknowledgment Request (R) bit in the Routing Header before transmitting the packet (since the C bit was set during Route Discovery by the node now listed as the IP.Destination_Address to indicate that it will propagate the packet out a different interface, and that node **A** will not receive a passive acknowledgment).
4. Set the retransmission timer for the packet in the Retransmission Buffer.
5. Transmit the packet.
6. If a passive or explicit acknowledgment is received before the retransmission timer expires, then remove the packet from the Retransmission Buffer and disable the retransmission timer. DONE.
7. Otherwise, when the Retransmission Timer expires, remove the packet from the Retransmission Buffer.
8. If DSR_MAXRXTSHIFT transmissions have been done, then attempt to salvage the packet (Section A.8.5). Also, generate a Route Error. DONE.
9. GOTO step 1.

Using Link Layer Acknowledgments

If explicit failure notifications are provided by the link layer, then all packets are assumed to be correctly received by the next hop and a Route Error is sent only when a explicit failure notification is made from the link layer.

Nodes receiving a packet without a Routing Header do not need to send an explicit Acknowledgment to the packet's originator, since the link layer will notify the originator if the packet was not received properly.

Originating a Route Error

If the next hop of a packet is found to be unreachable as described in Section A.8.5, a Route Error packet (Section A.7.2) MUST be returned to the node whose cache generated the information used to route the packet.

When a node **A** generates a Route Error for packet P, it MUST initialize the fields in the Route Error as follows:

```
Error.Source_Address      = Home address of node A
Error.Unreachable_Address = Home address of the unreachable node
```

- If the packet contains a DSR Routing Header and the S bit is NOT set, the packet has been forwarded without the need for salvaging up to this point.

```
    Error.Destination_Address = P.IP.Source_Address
```

- Otherwise, if the packet contains a DSR Routing Header and the S bit IS set, the packet has been salvaged by an intermediate node, and thus this Routing Header was placed there by the salvaging node.

```
    Error.Destination_Address = P.RoutingHeader.Address[1]
```

- Otherwise, if the packet does not contain a DSR Routing Header, the packet must have been originated by this node **A**.

```
Error.Destination_Address = Home address of node A
```

Send the packet containing the Route Error to Error.Destination_Address, performing Route Discovery if necessary.

As an optimization, Route Errors that are discovered by the packet's originator (such that Error.Source_Address is equal to Error.Destination_Address) SHOULD be processed internally. Such processing should invoke all the steps that would be taken if a Route Error option was created, transmitted, received, and processed, but an actual packet containing a Route Error option SHOULD NOT be transmitted.

Processing a Route Error Option

Upon receipt of a Route Error via any mechanism, a node SHOULD remove any route from its Route Cache that uses the hop (Error.Source_Address, Error.Index to Error.Unreachable_Address). This includes all Route Errors overheard, and those processed internally as described in Section A.8.5.

When the node identified by Error.Destination_Address receives the Route Error, it SHOULD verify that the source route responsible for delivering the Route Error includes the same hops as the working prefix of the original packet's source route (Error.Destination_Address to Error.Source_Address). If any hop listed in the working prefix is not included in the Route Error's source route, then the originator SHOULD forward the Route Error back along the working prefix (Error.Destination_Address to Error.Source_Address) so that each node along the working prefix will remove the invalid route from its Route Cache.

If the node processing a Route Error option discovers its home address is Error.Destination_Address and the packet contains additional Route Error option(s) later on the inside of the Hop by Hop options header, we call the additional Route Errors nested Route Errors. The node MUST deliver the first nested Route Error to Nested_Error.Destination_Address, performing Route Discovery if needed. It does this by removing the Route Error option listing itself as the Error.Destination_Address, finding the first nested Route Error option, and originating the remaining packet to Nested_Error.Destination_Address. This mechanism allows for the proper handling of Route Errors that are discovered while delivering a Route Error.

Salvaging a Packet

When node **A** attempts to salvage a packet originated at node **S** and destined for node **D**, it MUST perform the following steps:

1. Generate and send a Route Error to **S** as explained in Section A.8.5.
2. Call Route_Cache.Get() to determine if it has a cached source route to the packet's ultimate destination **D** (which is the last Address listed in the Routing Header).
3. If node **A** does not have a cached route for node **D**, it MUST discard the packet. DONE.
4. Otherwise, let Salvage_Address[1] through Salvage_Address[m] be the sequence of hops returned from the Route Cache. Initialize the following fields in the packet's header:

```
RT.Segments_Left   = m - 2;
RT.S               = 1
RT.Address[1]      = Home address of Node A
RT.Address[2]      = Salvage.Address[1]
...
RT.Address[n]      = Salvage.Address[m]
```

The IP Source Address of the packet MUST remain unchanged. When the Routing Header in the outgoing packet is processed, RT.Address[2], will be copied to the IP Destination Address field.

A.9. Optimizations

A number of optimizations can be added to the basic operation of Route Discovery and Route Maintenance as described in Sections A.8.4 and A.8.5 that can reduce the number of overhead packets and improve the average efficiency of the routes used on data packets. This section discusses some of those optimizations.

A.9.1. Leveraging the Route Cache

The data in a node's Route Cache may be stored in any format, but the active routes in its cache form a tree of routes, rooted at this node, to other nodes in the ad hoc network. For example, the illustration below shows an ad hoc network of six mobile nodes, in which mobile node **A** has earlier completed a Route Discovery for mobile node **D** and has cached a route to **D** through **B** and **C**:

```
B->C->D
+---+     +---+     +---+     +---+
| A |---->| B |---->| C |---->| D |
+---+     +---+     +---+     +---+


+---+
| F |                      +---+
+---+                      | E |
                           +---+
```

Since nodes **B** and **C** are on the route to **D**, node **A** also learns the route to both of these nodes from its Route Discovery for **D**. If **A** later performs a Route Discovery and learns the route to **E** through **B** and **C**, it can represent this in its Route Cache with the addition of the single new hop from **C** to **E**. If **A** then learns it can reach **C** in a single hop (without needing to go through **B**), **A** SHOULD use this new route to **C** to also shorten the routes to **D** and **E** in its Route Cache.

Promiscuous Learning of Source Routes

A node can add entries to its Route Cache any time it learns a new route. In particular, when a node forwards a data packet as an intermediate hop on the route in that packet, the forwarding node is able to observe the entire route in the packet. Thus, for example, when any intermediate node **B** forwards packets from **A** to **D**, **B** SHOULD add the source route information from that packet's Routing Header to its own Route Cache. If a node forwards a Route Reply packet, it SHOULD also add the source route information from the route record being returned in the Route Reply, to its own Route Cache.

In addition, since all wireless network transmissions at the physical layer are inherently broadcast, it may be possible for a node to configure its network interface into promiscuous receive mode, such that the node is able to receive all packets without link layer address filtering. In this case, the node MAY add to its Route Cache the route information from any packet it can overhear.

A.9.2. Preventing Route Reply Storms

The ability for nodes to reply to a Route Request not targeted at them by using their Route Caches can result in a Route Reply storm. If a node broadcasts a Route Request for a node that its neighbors have in their Route Caches, each neighbor may attempt to send a Route Reply, thereby wasting bandwidth and increasing the rate of collisions in the area. For example, in the network shown in Section A.9.1, if both node **A** and node **B** receive **F**'s Route Request, they will both attempt to reply from their Route Caches. Both will send their Replies at about the same time since they receive the broadcast at about the same time. Particularly when more than the two mobile nodes in this example are involved, these simultaneous replies from the mobile nodes receiving the broadcast may create packet collisions among some or all of these replies and may cause local congestion in the wireless network. In addition, it will often be the case that the different replies will indicate routes of different lengths. For example, **A**'s Route Reply will indicate a route to **D** that is one hop longer than that in **B**'s reply.

For interfaces which can promiscuously listen to the channel, mobile nodes SHOULD use the following algorithm to reduce the number of simultaneous replies by slightly delaying their Route Reply:

1. Pick a delay period

   $$d = H * (h - 1 + r)$$

   where h is the length in number of network hops for the route to be returned in this node's Route Reply, r is a random number between 0 and 1, and H is a small constant delay to be introduced per hop.

2. Delay transmitting the Route Reply from this node for a period of d.

3. Within the delay period, promiscuously receive all packets at this node. If a packet is received by this node during the delay period that is addressed to the target of this Route Discovery (the target is the final destination address for the packet, through any sequence of intermediate hops), and if the length of the route on this packet is less than h, then cancel the delay timer and do not transmit the Route Reply from this node; this node may infer that the initiator of this Route Discovery has already received a Route Reply, giving an equally good or better route.

A.9.3. Piggybacking on Route Discoveries

As described in Section A.5.1, when one node needs to send a packet to another, if the sender does not have a route cached to the destination node, it must initiate a Route Discovery, buffering the original packet until the Route Reply is returned. The delay for Route Discovery and the total number of packets transmitted can be reduced by allowing data to be piggybacked on Route Request packets. Since some Route Requests may be propagated widely within the ad hoc network, though, the amount of data piggybacked must be limited. We currently use piggybacking when sending a Route Reply or a Route Error packet, since both are naturally small in size. Small data packets such as the initial SYN packet opening a TCP connection [90] could easily be piggybacked.

One problem, however, arises when piggybacking on Route Request packets. If a Route Request is received by a node that replies to the request based on its Route Cache without propagating the Request (Section A.9.1), the piggybacked data will be lost if the node simply discards the Route Request. In this case, before discarding the packet, the node must construct a new packet containing the piggybacked data from the Route Request packet. The source route in this packet MUST be constructed to appear as if the new packet had been sent by the initiator of the Route Discovery and had been forwarded normally to this node. Hence, the first portion of the route is taken from the accumulated route record in the Route Request packet and the remainder of the route is taken from this node's Route Cache. The sender address in the packet MUST also be set to the initiator of the Route Discovery. Since the replying node will be unable to correctly recompute an Authentication header for the split off piggybacked data, data covered by an Authentication header SHOULD NOT be piggybacked on Route Request packets.

A.9.4. Discovering Shorter Routes

Once a route between a packet source and a destination has been discovered, the basic DSR protocol MAY continue to use that route for all traffic from the source to the destination as long as it continues to work, even if the nodes move such that a shorter route becomes possible. In many cases, the basic Route Maintenance procedure will discover the shorter route, since if a node moves enough to create a shorter route, it will likely also move out of transmission range of at least one hop on the existing route.

Furthermore, when a data packet is received as the result of operating in promiscuous receive mode, the node checks if the Routing Header packet contains its address in the unprocessed portion of the source route (Address[NumAddrs - SegLft] to Address[NumAddrs]). If so, the node knows that packet could bypass the unprocessed hops preceding it in the source route. The node then sends what is called a gratuitous Route Reply message to the packet's source, giving it the shorter route without these hops.

The following algorithm describes how a node **A** should process packets with an IP.Destination_Address not addressed to **A** or the IP broadcast address or a multicast address that are received as a result of **A** being in promiscuous receive mode:

1. If the packet is not a data packet containing a Routing Header, drop the packet. DONE.
2. If the home address of this node does not appear in the portion of the source route that has not yet been processed (indicated by Segments Left), then drop the packet. DONE.
3. Otherwise, the node **B** that just transmitted the packet (indicated by Address[NumAddrs - SegLft - 1]) can communicate directly with this node **A**. Create a Route Reply. The Route Reply MUST list the entire source route contained in the received packet with the exception of the intermediate nodes between node B and node A.
4. Send this gratuitous Route Reply to the node listed as the IP.Source_Address of the received packet. If Route Discovery is required it MAY be initiated, or the gratuitous Route Reply packet MAY be dropped.

A.9.5. Rate Limiting the Route Discovery Process

One common error condition that must be handled in an ad hoc network is the case in which the network effectively becomes partitioned. That is, two nodes that wish to communicate are not within transmission range of each other, and there are not enough other mobile nodes between them to form a sequence of hops through which they can forward packets. If a new Route Discovery was initiated for each packet sent by a node in this situation, a large number of unproductive Route Request packets would be propagated throughout the subset of the ad hoc network reachable from this node. In order to reduce the overhead from such Route Discoveries, we use exponential back-off to limit the rate at which new Route Discoveries may be initiated from any node for the same target. If the node attempts to send additional data packets to this same node more frequently than this limit, the subsequent packets SHOULD be buffered in the Send Buffer until a Route Reply is received, but it MUST NOT initiate a new Route Discovery until the minimum allowable interval between new Route Discoveries for this target has been reached. This limitation on the maximum rate of Route Discoveries for the same target is similar to the mechanism required by Internet nodes to limit the rate at which ARP requests are sent to any single IP address [13].

A.9.6. Improved Handling of Route Errors

All nodes SHOULD process all of the Route Error messages they receive, regardless of whether the node is the destination of the Route Error, is forwarding the Route Error, or promiscuously overhears the Route Error.

Since a Route Error packet names both ends of the hop that is no longer valid, any of the nodes receiving the error packet may update their Route Caches to reflect the fact that the two nodes indicated in the packet can no longer directly communicate. A node receiving a Route Error packet simply searches its Route

Cache for any routes using this hop. For each such route found, the route is effectively truncated at this hop. All nodes on the route before this hop are still reachable on this route, but subsequent nodes are not.

An experimental optimization to improve the handling of errors is to support the caching of "negative" information in a node's Route Cache. The goal of negative information is to record that a given route was tried and found not to work, so that if the same route is discovered again shortly after the failure, the Route Cache can ignore or downgrade the metric of the failed route.

We have not currently included this caching of negative information in our simulations, since it appears to be unnecessary if nodes also promiscuously receive Route Error packets.

A.9.7. Increasing Scalability

We recently designed and began experimenting with ways to integrate ad hoc networks with the Internet and with Mobile IP [82]. In addition to this, we are also exploring ways to increase the scalability of ad hoc networks by taking advantage of their cooperative nature and the fact that some hierarchy can be imposed on an ad hoc network, just be assigning addresses to the nodes in a reasonable way. These ideas are described in a workshop paper [17].

A.10. Path-State and Flow-State Mechanisms

This section describes the current design of our framework for supporting better-than-best-effort Quality of Service in DSR networks. The framework dovetails into DSR's existing mechanisms, and, like DSR itself, is completely on-demand in nature — no packets are sent unless there is user data to transfer. The framework is based on the introduction of two kinds of soft-state, called *path-state* and *flow-state*, at the intermediate nodes along the path between senders and destinations.

Taken together, the path-state and flow-state extensions extend the Quality of Service provided by DSR networks in the following ways:

- They eliminate the need for all data packets to carry a source route, increasing the efficiency of the protocol in general.
- They provide accurate measurements of the state of the network to higher layers protocols for use in adaptation.
- They enable senders to explicitly manage the consumption of resources across the network.
- They enable the network to provide better-than-best-effort service via admission control and per-flow resource management.

A.10.1. Overview

Path-state allows intermediate nodes to forward packets according to a predetermined source route, even when most packets do not include the full source route. Conceptually, the originator of each data packet initially includes both a source route and a unique *path identifier* in each packet it sends. As intermediate nodes forward the packet, they cache the source route from the packet, indexed by the path identifier. The source can then send subsequent packets carrying only the path identifier, since intermediate nodes will be able to forward the packet based on the source route for the path that they have cached.

While path-state allows the elimination of the source route from each packet, thereby reducing the overhead of the DSR protocol, it also provides a way for sources to monitor the state of each path through the network. When a source wishes to know the characteristics of a path through the network, it piggybacks a *path-metrics* header onto any data or control packet traversing the path. As the packet propagates through the network, each intermediate node updates the set of *path-metrics* carried by the packet to reflect the local network conditions seen at the node. These metrics are reflected back to the sender by the destination, along with the path identifier, and enable the sender to track the value of these metrics for each of the source routes it is currently using.

We are currently experimenting with metrics that are easy for nodes to measure, that require constant size to represent regardless of source route length, and that would enable the sender's network layer to provide useful feedback to higher layers on the state of the network. For example, by including "available bandwidth" or "battery level" as a metric, senders can load balance the consumption of resources across the network. We have also considered the possibility of replacing the TCP congestion control algorithm with a "leaky-bucket" system controlled by the reflected path-metrics — our measured performance results show this could dramatically improve TCP throughput in environments where many packets are lost due to packet corruption. The feedback could also be used as inputs to other researcher's systems for improving the transport layer, such as Liu and Singh's ATCP [66], or for adaptation at higher layers, as in Odyssey [78].

Flow-state allows a source to differentiate its traffic into *flows*, and then request better-than-best-effort handling for these flows. With the additional information provided by the flow-state, the network can provide admission control and *promise* a specific Quality of Service (QoS) to each flow. Since the ad hoc network may frequently change topology, the flow-state mechanisms are directly integrated into the routing protocol to minimize their reaction time and provide *notifications* to a flow when the network must break its promise for a specific level of QoS.

A.10.2. Path-State and Flow-State Identifiers

The metadata that intermediate nodes in the network must process is divided into *path-state* and *flow-state*, where path-state is the fundamental unit of routing information and flow-state is the fundamental unit of Quality of Service information.

Path-state is associated with a particular set of hops through the network from some source **S** to a destination **D** (i.e., a particular source route in the network). It consists of the information needed to route packets along the path, and information about the carrying capacity of the path, such as the unused bandwidth along the path or the minimum latency of the path.

Flow-state is specific to a particular class of packets flowing between **S** and **D** that is routed over a given path. Flow-state is used to record Quality of Service promises that have been made for a particular flow, and allows packets from **S** to **D** that take the same path through the network to be treated differently by intermediate nodes. For example, all the TCP connections between **S** and **D** that take the same path will share the same path-state, but may have independent flow-state. At any point in time, **S** may use multiple paths for its traffic to **D** and each of these paths may be comprised of many flows. However, a single network layer flow may not be multiplexed over different paths.

To represent paths and flows inside the network, we use a scheme inspired by the Virtual Path Index and Virtual Circuit Index of ATM networks [105, p. 451]. Paths are identified by the logical concatenation of the source node address and a 16-bit path identifier where the low 5 bits are 0. Flows are identified by a path identifier where the low 5 bits are used to distinguish between the different flows that use the same path.

This scheme has two main advantages. First, each source node can independently generate globally unique path- and flow-identifiers. Second, the hierarchical relation of flow-identifiers to path-identifiers means that many flows from the same source node can share the same path-state, which reduces the overhead of maintaining the routing information. The drawback is that if a flow must be rerouted, its flow identifier will change. However, when a flow is rerouted the QoS metadata must be renegotiated anyway, so changing flow identifiers will not create needless additional work in the network.

A.10.3. Path-State Creation, Use, and Maintenance

The path-state portion of the protocol has two major goals. The first goal is to ensure sufficient state exists at the nodes along a path from a source **S** to a destination **D** so that packets from **S** to **D** do not need to carry the complete source route. The second goal is to allow **S** to discover the characteristics of a particular path to **D** so that it can adapt its sending pattern to the capabilities of the path, or even choose a different path entirely.

The next sections describe how the path-state is created, how the characteristics of the path are discovered, and what metrics can be used to characterize the path.

Creating Path-State for Routing

To create the path-state, we assume that Route Discovery proceeds as normal in DSR. Once the source node **S** has obtained a source route to the destination **D**, it begins sending data packets to **D** as normally done in DSR, with each packet carrying a full source route header. Internally, **S** assigns a path-identifier to that particular source route and stores the path-identifier in its route cache along with the source route. **S** then includes the path-identifier as part of the source route header as shown in Figure A.2(a). As each intermediate node processes the source route to forward the packet, it also stores the source route in its route cache, indexed by the source and path-identifier.

```
A -----------------> B -----------------> C -----------------> D

   +-------------+      +-------------+      +-------------+
   |src: A       |      |src: A       |      |src: A       |
   |dst: D       |      |dst: D       |      |dst: D       |
   |path-id: 15  |      |path-id: 15  |      |path-id: 15  |
   |rt: A,(B),C,D|      |rt: A,B,(C),D|      |rt: A,B,C,(D)|
   +-------------+      +-------------+      +-------------+
   |   payload   |      |   payload   |      |   payload   |

        (a) Packet with path identifier and source route.

A -----------------> B -----------------> C -----------------> D

   +-------------+      +-------------+      +-------------+
   |src: A       |      |src: A       |      |src: A       |
   |dst: D       |      |dst: D       |      |dst: D       |
   |path-id: 15  |      |path-id: 15  |      |path-id: 15  |
   +-------------+      +-------------+      +-------------+
   |   payload   |      |   payload   |      |   payload   |

            (b) Packet with path identifier only.
```

Figure A.2 Path identifiers assigned to a source route by the originating node **A** enable later packets to omit the source route.

After sending a packet containing both the source route and the path-identifier into the network, **S** can begin sending subsequent packets to **D** without a full source route — carrying only the path-identifier as shown in Figure A.2(b). Each intermediate node receiving such a packet queries its route cache to find the route the packet is supposed to take, and determines its next hop. As explained in Section A.10.5, if the cached source route is not available at some intermediate node, **S** will receive a ROUTE ERROR and can then correct the situation.

Monitoring Characteristics of the Path

In order to support network layer services such as balancing the traffic load across the network, end-systems must have a method for determining the characteristics of the paths through the network that they could use. While many schemes have been proposed by which the end-systems themselves can measure the characteristics of a path (e.g., TCP congestion window and RTT calculations [3, 104, 111] and SPAND [102]), we hypothesize that, particularly in the in the dynamic environment of an ad hoc network, more useful, more accurate, and more timely information can be developed by enlisting the aid of the nodes along the path to measure the path characteristics.

We propose that each node can measure the activity around itself, and thereby determine information such as: the mean latency it adds to the packets it forwards and the latency variation (jitter); the number of additional packets per second it believes it can process; or the unused amount of wireless media capacity in the air around the node. Experimentation will be required to discover exactly which metrics will prove to be accurately measurable and useful, though Section A.10.3 provides several proposals. If the metrics kept by each node on a path are combined, the result should be a characterization of the path that the packet sender can use to organize or adapt its offered load.

To implement this scheme, we first define a new type of extension header for DSR than can be piggybacked onto a packet in the same way as the existing DSR headers. This new header is called the *path metrics header* (written as MEASURE) and conceptually consists of the path-identifier of the path along which the metrics are measured, the type of the Measure, and the metrics themselves encoded in a TLV format (Section A.10.6).

Whenever a sender **S** wishes to measure the characteristics of a path it is using, it includes the MEASURE header in any packet it sends along that path, setting the `type` of the header to **record**. As each node along the path forwards the packet, it updates the variables inside the MEASURE header with the metrics it has measured locally. When the header reaches the final destination **D**, **D** sets the `type` of the MEASURE header to **return** and piggybacks the header into any packet headed back to **S**. Since the path metrics header includes the path-identifier of the path along which it was measured, **S** can include the data into its route cache for future use, and can treat the receipt of the path metrics header as a positive acknowledgment that the path-state between **S** and **D** for the given path-identifier is correctly set up. This could lead **S** to cease including source routes in the packets it sends along the path, as described in Section A.10.3.

If we find that it is valuable to immediately provide **S** with the path metrics of every discovered route, we could alter Route Discovery slightly to generate this information. Currently, if an intermediate node has a cached route that it can use to answer a ROUTE REQUEST, it generates a ROUTE REPLY itself. Instead, we could require it to place its proposed route on the ROUTE REQUEST (turning it from a flood-fill broadcast into a unicast packet) and send the packet to the destination so it will measure the metrics of the complete path. The destination will then return the metrics to the source along with the ROUTE REPLY as described above.

> We have been intending to experiment with this alteration to Route Discovery for some time, since it offers two benefits, even without path-state metrics. It should decrease the number of broken routes returned by Route Discovery since each cached route is tested before being returned, and it should save us from jeopardizing one data packet for every bad route in someone's cache. The cost is some extra latency on Route Discovery.

Candidate Metrics

In order to limit the additional overhead that collecting and distributing path-state metrics will place on the network, all the metrics must have the property that the amount of space required to express the metric does not increase as the number of hops on the path increases. Experimentation will be required to determine which metrics are most accurately measured and most useful, but our initial set of candidates includes the following:

- Interface queue length — Our previous work [71] has shown that this is a good estimator of local congestion.

- Rate of interface queue draining — When an interface is backlogged, the rate at which packets leave the queue directly measures the usable capacity of that interface.

- Quiet time fraction — When an interface is not backlogged, the usable capacity of the interface can be estimated by promiscuously listening to the media and measuring the fraction of time during which it is not in use (though this will overestimate the capacity).

- Fraction Free Air Time — The fraction of time our interface would be able to send a packet. That is, the fraction of time the interface does not sense carrier, is not deferring, and is not backed off. Current experiments show this is an excellent predictor of congestion and available capacity.
- Forwarding latency and variation — This can be measured as the time between when a packet is received and when it is acknowledged by the next hop.
- Unidirectional links — Paths containing unidirectional links are usable, but undesirable as they increase the overhead of Route Maintenance.
- Packet loss rate — Signal quality information from the interface itself, or the frequency of hop-by-hop retransmission, can be used to estimate the loss rate of each link.
- Likelihood of path breakage — Intermediate nodes may know ahead of time that they intend to shutdown or move such that paths through them will no longer work.

These metrics all have the property that they can be expressed in a single value that each node can measure locally. As a packet with a path metrics header passes through a node, the metrics in the header can be updated to reflect the node's metrics using a combination function like minimum, maximum, sum, or weighted average that produces another single value to replace the one already in the header. This updating will be done at the last possible moment before the packet is forwarded, in order to assure the packet has the most current metrics on it when it leaves.

A.10.4. Flow-State Creation, Use, and Maintenance

The flow-state portion of the protocol enables a sender to obtain promises from all nodes along a path to a destination that a certain set of resources are available along the path, and that the intermediate nodes are committed to making these resources available for the particular flow. This allows a sender to obtain better-than-best-effort Quality of Service for a flow by obtaining promises from the intermediate nodes to reserve the resources needed to provide that QoS.

Unlike prior QoS work in wired networks, at this point we cannot formally characterize or bound exactly what type of services the flow-state protocol will be able to offer. The goal is to provide CBR and TCP streams with the ability to specify and obtain a minimum bandwidth and delay/jitter bound. If the environment is particularly harsh, it is possible that only best-effort service will be offerable. It is this intuition that leads us to the system of promises and notifications. Experimentally, we hope to determine how stable and effective this system will be in a multi-hop ad hoc network environment.

Requesting Promises along Existing Paths

Similar to the use of the path metrics header, at any time a promise can be requested or changed along any path an originator is currently using. Once an originating node has created a path-identifier for a route through the network, it can request a promise of network resources along that route by first generating a new flow-identifier to identify the promise. The originator then fills out a *flow-request header* (written as FLOW REQUEST) and inserts it into any packet sent along that path.

Figure A.3 shows the conceptual layout of a FLOW REQUEST, which contains the new path-identifier assigned by the originator, the flow-identifier of the promise that this request supersedes (if any), the requested lifetime of the promise, and the QoS parameters that describe the requested promise itself. Section A.10.6 provides the detailed packet format. The use of the minimum and requested fields for the QoS parameters differs depending on whether the FLOW REQUEST is piggybacked on a ROUTE REQUEST or not, as described below.

```
+-------------------------------------+
|  flow-id        |     old flow-id   |
+-------------------------------------+
|             lifetime                |
+-------------------------------------+
| capacity   |    min   |    desired  |
|  latency   |    min   |    desired  |
|variation   |    min   |    desired  |
|     loss   |    min   |    desired  |
+-------------------------------------+
```

Figure A.3 Conceptual layout of the Flow Request header.

When a Flow Request piggybacked on a unicast packet is received by a node, the node performs the following steps:

- If the node is the destination of the packet, it converts the Flow Request into a Measure with type **return** and uses the current values in the `desired` fields of the Flow Request to populate the fields of the Measure. It then piggybacks the Measure onto any packet being returned to the originator.
- Else if the intermediate node has available enough resources to meet the `minimum` requested promise in the Flow Request, it:
  - Sets aside the maximum of its available resources and the `desired` resources. The set aside resources are held in a *tentative promise* pool until the promise is confirmed, or a relatively short timeout expires.
  - Nodes can recycle resources from listed `old flow-id`
  - Updates the `desired` fields of the Flow Request to reflect the resources set aside (there is questionable value in a down stream node allocating more resources to a flow than an upstream node can currently handle).
  - Forward the packet and piggybacked Flow Request to the next node on the path.
- Else, the node does not have enough resources to meet the `minimum` requested promise, so it sends the originator a Route Error piggybacked with a Measure reflecting the minimum of the current values of the `desired` fields in the Flow Request and the available resources. The `type` field is set to **refused**. Such a Measure enables the originator to learn three things: that its requested cannot be satisfied along the given path; the identity of the bottleneck node; and the available resources up to and through the bottleneck node.

When the originating node receives a Measure header of type **return** for a flow on which it has an outstanding Flow Request, it accepts the promised level of service by changing the type of the Measure header to **confirm** and piggybacking the header on any packet going along the flow. This informs the intermediate nodes to move the set aside resources from the tentative promise pool to the allocated pool, and enables upstream nodes to free any set aside resources in excess of the capacity of a bottleneck downstream node.

The use of the `old flow-id` to recycle resources is important for two reasons. First, it enables an originator to attempt to increase or decrease the amount of a current promise without losing the resources it already has promised. Second, both packet loss and the expanding ring search of Route Discovery may result in several FLOW REQUESTS being sent for the same flow. If subsequent FLOW REQUESTS for a flow were not able to notify intermediate nodes that they can reuse resources set aside while processing earlier FLOW REQUESTS, the network could quickly reach a state where admissible flows are being needlessly rejected.

Requesting Promises as Part of Route Discovery

The scheme for requesting promises described in the previous section has the advantage that it enables an originator to request or update a promise for a flow along any route currently in its route cache, regardless of how it obtained the route. For the common case in which a node wishes to obtain a resource promise for a new flow to a previously unknown destination, we can integrate the flow request with the Route Discovery for the destination.

Integrating the flow request with Route Discovery enables us to avoid the inefficiency of discovering routes that will not be usable by the flow due to insufficient resources. The integration of flow requests with Route Discovery also allows us to avoid a common pitfall of QoS schemes that layer a reservation signaling protocol on top of a unicast routing algorithm — schemes without tight integration will refuse admissible flows whenever the unicast routing algorithm directs the request packets into a congested area of the network, unless the signaling protocol also provides a method to backtrack the request and route around the congested area. Utilizing the same mechanisms currently used in Route Discovery, we can avoid the need for backtracking.

We call the combination of flow requests with Route Discovery *QoS-guided Route Discovery*, which originating nodes can invoke simply by piggybacking a FLOW REQUEST on the ROUTE REQUEST. Each node receiving the FLOW REQUEST uses the same algorithm described in Section A.10.4, with two exceptions:

- Nodes *silently discard* the ROUTE REQUEST if they can not meet minimum requirements
- Unless the ROUTE REQUEST indicates that replying from cache is forbidden, nodes with a cached route to destination unicast the ROUTE REQUEST along the cached route.

A node requiring a route with a QoS promise uses the following algorithm. First, it sends a ROUTE REQUEST that permits intermediate nodes to reply from cache. If the network is uncongested, this should frequently and quickly succeed in returning both a ROUTE REPLY and a MEASURE describing the available QoS along the discovered path. If after a timeout, the originating node has not received a ROUTE REPLY, it begins another Route Discovery, this time forbidding replies from cache, which will force an exploration of all feasible paths to the destination.

This scheme does risk an implosion of unicast REQUESTS at the target of the Route Discovery (e.g., if target is a popular server to which many nodes have cached routes). At the cost of additional complexity and soft-state, it would be possible to add hold-downs at the nodes surrounding the target so that only the first few REQUESTS are forwarded towards the target.

Providing Notifications of Changing Path Metrics

When a node detects that it must break a promise, it must notify the node to which it made that promise. It is an open question how the now reduced resources should be distributed among the flows. We currently pick the minimum set of promises to break that leave the other promises unchanged.

The difficulty in providing notification of a changed path metric is getting this information back to the source. When promise must be broken at a node **B**, it sends a MEASURE to the originator indicating what resources are now available. The use of MEASURE headers to determine the currently available resources along a path is more problematic, however, as for every MEASURE sent by the originator, the destination must send a response containing the measured metrics.

XXX - why not just send a “new measure reply” with the updated info?

If the traffic is TCP, the overhead of the responses are low, as they can be piggybacked on the ACK stream. For one-way CBR traffic though, introducing the overhead of a reverse stream to carry the changing metrics could be severe.

If the overhead of the responses becomes a problem, it may be possible to implement a enhanced piggyback mechanism. The approach is based on the fact that although no work has been exerted to create hop-by-hop routing information at each node, chances are good that each node can determine a next-hop for packets headed to any known destination by simply examining its route cache. By piggybacking the MEASURE header for one hop onto *any* packet that is headed to that next-hop, we can cheaply create a reverse flow of information that will eventually reach the originator of the MEASURE. Each node who receives a MEASURE with a type of **return** simply piggybacks the MEASURE for one-hop on packets that seem to be flowing the right direction back to the source. To insure the timeliness of the information, each MEASURE being returned to an originator could include a deadline by which the information is supposed to reach the originator. If it appears that hop-by-hop propagation will result in missing the deadline, the MEASURE can be unicast as a first-class packet to the originator.

A.10.5. Expiration of State from Intermediate Nodes

Since there is no guarantee that either the source or destination of a packet flow will be able to communicate with all of the nodes that carried the flow when they wish to terminate the flow, there must be time-based expiration mechanism by which intermediate nodes can purge the path-state and flow-state from their caches and reclaim the resources set aside to maintain it. However, if intermediate nodes were to purge the state of an active flow, the intermediate nodes would find themselves with packets to forward that do not contain a source route, but only contain a flow-identifier that references state they no longer hold. Since intermediate nodes do not necessarily know the timing with which the sender originates packets, an inactivity timer alone would have to be set very conservatively to prevent purging the path-state of low bit-rate connections.

To solve the expiration problem, we take advantage of the relatively “soft” nature of the path-state and flow-state. When the state is created, the source node specifies a time after which it should be discarded (This time will typically be on the order of a hundred seconds). The source node can thereby estimate how often it must refresh the state, for example, by sending packets that contain a full source route on them. Should the state have somehow expired at an intermediate node when a packet labeled with a flow or path identifier arrives, the intermediate node can return a ROUTE ERROR to the source node specifying “missing state information” as the cause of the ERROR and elicit the sender to refresh the missing state.

Since all path-state information is guaranteed to have expired from the network after a bounded amount of time, nodes can safely and unambiguously reuse path- and flow-identifiers after that period.

A.10.6. Packet Formats

Identifier Option

Path and flow identifiers are carried as an option inside the Hop-by-Hop options header. This option MAY NOT appear more than once in a single Hop-by-Hop Options header.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Option Type  | Option Length |     Path-ID         | Flow-ID |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Option Type Value: TBD. A node that does not understand this option should ignore this option and continue processing the packet, and the Option Data does not change en-route (the top three bits are 000).

Option Length 8-bit unsigned integer. Length of the option, in octets, excluding the Option Type and Option Length fields.

Path-ID The identifier assigned to this path by the node listed as the IP Source Address (Section A.10.2).

Flow-ID The identifier assigned by the node listed as the IP Source Address to a particular flow along the path identified by the Path-ID. If this portion is 0, the option names a path, but not a particular flow.

Discussion: This encoding of the path and flow identifiers will cost 8 bytes of additional header overhead in a data packet with no other extensions or options (4 bytes for the Hop-by-Hop options header, and 4 bytes for the identifier option). A more compact encoding would be to define that, in a DSR network, an IP destination address with a first octet of 127 actually encodes the path and flow identifiers as follows:

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |0 1 1 1 1 1 1 1|     reserved    |     Path-ID       | Flow-ID |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The DSR module of the final destination would replace the IP destination address with its actual value before passing the packet up the stack for further processing.

This encoding has the advantage that it requires no additional overhead in a data packet. The disadvantage is that if the packet was somehow received by a DSR-unaware node without first being processed by a DSR gateway node [17], the DSR-unaware node will either drop the packet or will attempt to receive it locally (since the IP destination address belongs to the loopback subnet).

Path-Metrics Option

Path-metrics are carried as an option inside the Hop-by-Hop options header.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   Option Type | Option Length |       Path-ID     | Flow-ID |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      Type     |  Metrics...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Option Type Value: TBD. A node that does not understand this option should ignore this option and continue processing the packet, and the Option Data does change en-route (the top three bits are 001).

Option Length 8-bit unsigned integer. Length of the option, in octets, excluding the Option Type and Option Length fields.

Path-ID and Flow-ID The path identifier of the path that the metrics correspond to. If the Path-Metrics Option Type equals Measure, then the Path-ID and Flow-ID fields MUST equal those in any Identifier Option carried in the Hop-by-Hop Options Header.

Type One of

- **Measure** Each node processing the option should update the metrics to reflect the conditions at that node.
- **Reply** The metrics in this option SHOULD NOT be modified by any intermediate node. They represent the metrics measured along the identified path.
- **Confirm** The metrics in this option MUST NOT be modified by any intermediate node. They represent a confirmation by the sender that will transmit traffic conforming to the listed Quality of Service metrics along the identified flow.

Metrics The individual path-metrics, encoded as described in Section A.10.6. Unknown metrics SHOULD be ignored. If a single value is provide for the metric, it MUST be interpreted as the metrics value. If two values are provided for the metric, they MUST be interpreted as the range of values taken by the metric (low value first). It is undefined for there to be more than two values for the metric.

Flow Request Option

Flow-requests are carried as an option inside the Hop-by-Hop options header. They allow a sender to request that intermediate nodes reserve sufficient resources for a flow to provide that flow with the QoS characteristics described by the metrics.

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Option Type | Option Length |           Lifetime            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |         old         |   old   |         new         |   new   |
 |       Path-ID       | Flow-ID |       Path-ID       | Flow-ID |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Metrics ...
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Option Type Value: TBD. A node that does not understand this option should ignore this option and continue processing the packet, and the Option Data does change en-route (the top three bits are 001).

Option Length 8-bit unsigned integer. Length of the option, in octets, excluding the Option Type and Option Length fields.

old Path-ID and old Flow-ID The flow identifier provide in a previous request which this request supersedes.

new Path-ID and new Flow-ID The flow identifier that will be used with to identify the packets that should receive the QoS described by the included metrics.

Metrics The metrics that characterize the desired QoS, encoded as described in Section A.10.6. Unknown metrics SHOULD be ignored. If a range of values are provided for a metric, they MUST be interpreted as the minimum acceptable value and the desired value.

Encoding Path-Metrics

Each path-metric is encoded in a modified Type-Length-Value form as

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type       |R|   Length    |     Data...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type The type of metric

R bit If 0, the data is a list of discrete values the metric can or did take. If 1, the data represent a range of values the metric can or did take. If a single metric value is supplied, the range is assumed to be 0 <= metric <= value. If two metric values are supplied, the range is assumed to be value1 <= metric <= value2.

Option Length 8-bit unsigned integer. Length of the metric, in octets, excluding the Type and Length fields.

The currently defined metric types follow:

Padding Type: 0
The padding metric is special in that it contains no length field and no data.

Available Capacity Type: 1

Data encoded as

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Mantissa       |  Shift  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where the value is (Mantissa << Shift) bits per second.

Delay and Delay Variation Data encoded as

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Delay              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: 2 - Delay

The value is Delay milliseconds.

Type: 3 - Delay Variation

The value is the standard deviation of Delay, in milliseconds.

Link Bidirectionality Type: 16 - Link Bidirectionality

Data encoded as

```
  0                   1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | # Uni-links   | #Explicit ACK |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where # Uni-links is the number of uni-directional links on the path, and # Explicit ACK is the number of hops which require explicit acknowledgments.

Packet Loss Rate Data encoded as

```
  0                   1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   # Packets Lost              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where the loss rate is (# Packets Lost / 2 ** 16).

Type: 17 - Path Packet Loss Rate

The value is the expected packet loss rate of the entire path

Type: 18 - Worst Loss Rate

The value is the expected packet loss rate of the single worst link in the path.

A.11. Constants

BROADCAST_JITTER	10	milliseconds
MAX_ROUTE_LEN	15	nodes
Interface Indexes		
IF_INDEX_INVALID	0x7F	
IF_INDEX_MA	0x7E	
IF_INDEX_ROUTER	0x7D	
Route Cache		
ROUTE_CACHE_TIMEOUT	300	seconds
Send Buffer		
SEND_BUFFER_TIMEOUT	30	seconds
Request Table		
MAX_REQUEST_ENTRIES	32	nodes
MAX_REQUEST_IDS	8	identifiers
MAX_REQUEST_REXMT	16	retransmissions
MAX_REQUEST_PERIOD	10	seconds
REQUEST_PERIOD	500	milliseconds
RING0_REQUEST_TIMEOUT	30	milliseconds
Retransmission Buffer		
DSR_RXMT_BUFFER_SIZE	50	packets
Retransmission Timer		
DSR_MAXRXTSHIFT	2	

A.12. IANA Considerations

This document proposes the use of the Destination Options header and the Hop-by-Hop Options header, originally defined for IPv6, in IPv4. The Next Header values indicating these two extension headers thus must be reserved within the IPv4 Protocol number space.

Furthermore, this document defines four new types of destination options, each of which must be assigned an Option Type value:

- The DSR Route Request option, described in Section A.7.1
- The DSR Route Reply option, described in Section A.7.2
- The DSR Route Error option, described in Section A.7.2
- The DSR Acknowledgment option, described in Section A.7.2

DSR also requires a routing header Routing Type be allocated for the DSR Source Route defined in Section A.7.3.

In IPv4, we require two new protocol numbers be issued to identify the next header as either an IPv6-style destination option, or an IPv6-style routing header. Other protocols can make use of these protocol numbers as nodes that support them will processes any included destination options or routing headers according to the normal IPv6 semantics.

A.13. Security Considerations

This document does not specifically address security concerns. This document does assume that all nodes participating in the DSR protocol do so in good faith and with out malicious intent to corrupt the routing ability of the network. In mission-oriented environments where all the nodes participating in the DSR protocol share a common goal that motivates their participation in the protocol, the communications between the nodes can be encrypted at the physical channel or link layer to prevent attack by outsiders.

A.14. Location of DSR Functions in the ISO Reference Model

When designing DSR, we had to determine at what level within the protocol hierarchy to implement source routing. We considered two different options: routing at the link layer (ISO layer 2) and routing at the network layer (ISO layer 3). Originally, we opted to route at the link layer for the following reasons:

- Pragmatically, running the DSR protocol at the link layer maximizes the number of mobile nodes that can participate in ad hoc networks. For example, the protocol can route equally well between IPv4 [89], IPv6 [25], and IPX [45] nodes.
- Historically, DSR grew from our contemplation of a multi-hop ARP protocol [49, 51] and source routing bridges [87]. ARP [88] is a layer 2 protocol.
- Technically, we designed DSR to be simple enough that that it could be implemented directly in network interface cards, well below the layer 3 software within a mobile node. We see great potential for DSR running between clouds of mobile nodes around fixed base stations. DSR would act to transparently fill in the coverage gaps between base stations. Mobile nodes that would otherwise be unable to communicate with the base station due to factors such as distance, fading, or local interference sources could then reach the base station through their peers.

Ultimately, however, we decided to specify DSR as a layer 3 protocol since this is the only layer at which we could realistically support nodes with multiple interfaces of different types.

A.15. Implementation Status

We have implemented Dynamic Source Routing (DSR) under the FreeBSD 2.2.7 operating system running on Intel x86 platforms. FreeBSD is based on a variety of free software, including 4.4 BSD Lite from the University of California, Berkeley.

During the 7 months from August 1998 to February 1999, we designed and implemented a full-scale physical testbed to enable the evaluation of ad hoc network performance in the field. The last week of February and the first week of March included demonstrations of this testbed to a number of our sponsors and partners, including Lucent Technologies, Bell Atlantic, and DARPA. A complete description of the testbed is available as a Technical Report [71].

The software is currently being ported to FreeBSD 3.3.

Implementors notes:

- Added field to Route Error

A.16. Acknowledgments

The protocol described in this draft has been designed within the CMU Monarch Project, a research project at Carnegie Mellon University which is developing adaptive networking protocols and protocol interfaces to allow truly seamless wireless and mobile node networking [53, 91]. The current members of the CMU Monarch Project include:

- Robert V. Barron
- Josh Broch
- Yih-Chun Hu
- Jorjeta Jetcheva
- David B. Johnson
- Qifa Ke
- David A. Maltz

A.17. Chair’s Address

The Working Group can be contacted via its current chairs:

```
M. Scott Corson
Institute for Systems Research
University of Maryland
College Park, MD  20742
USA

Phone:  +1 301 405-6630
Email:  corson@isr.umd.edu


Joseph Macker
Information Technology Division
Naval Research Laboratory
Washington, DC  20375
USA

Phone:  +1 202 767-2001
Email:  macker@itd.nrl.navy.mil
```

A.18. Authors' Addresses

Questions about this document can also be directed to the authors:

```
Josh Broch
Carnegie Mellon University
Electrical and Computer Engineering
5000 Forbes Avenue
Pittsburgh, PA  15213-3890
USA

Phone:  +1 412 268-3056
Fax:    +1 412 268-7196
Email:  broch@cs.cmu.edu


David B. Johnson
Carnegie Mellon University
Computer Science Department
5000 Forbes Avenue
Pittsburgh, PA  15213-3891
USA

Phone:  +1 412 268-7399
Fax:    +1 412 268-5576
Email:  dbj@cs.cmu.edu


David A. Maltz
Carnegie Mellon University
Computer Science Department
5000 Forbes Avenue
Pittsburgh, PA  15213-3891
USA

Phone:  +1 412 268-3621
Fax:    +1 412 268-5576
Email:  dmaltz@cs.cmu.edu
```