# PERFORMANCE EVALUATION OF ROUTING PROTOCOLS FOR QOS SUPPORT IN RURAL MOBILE AD HOC NETWORKS

by

Chad Brian Bohannan

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April, 2008

© Copyright

by

Chad Brian Bohannan

2008

All Rights Reserved

APPROVAL

of a thesis submitted by

Chad Brian Bohannan

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Dr. Jian (Neil) Tang

Approved for the Department of Computer Science

Dr. John Paxton

Approved for the Division of Graduate Education

Dr. Carl Fox

# STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfullment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Chad Brian Bohannan

April, 2008

# ACKNOWLEDGEMENTS

I would like to thank Dr. Jian (Neil) Tang for his patience and wisdom while I struggled to learn what I needed to develop QASR. I would also like to thank Dr. Richard Wolff and Doug Galarus for their input throughout the project.

## Funding Acknowledgment

This work was supported by the Safecom Program and the Department of Homeland Security under Award No. 2007-ST-086-000001. However, any opinions, findings, conclusions, or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of Safecom or the DHS.

TABLE OF CONTENTS

1. INTRODUCTION . . . . . 1

2. WIRELESS DATA NETWORKING . . . . . 3

Wireless Link . . . . . 3
TDMA . . . . . 4
CSMA/CA . . . . . 5
Interference Model . . . . . 7
Wireless Routing . . . . . 8
DSR . . . . . 9
AODV . . . . . 10
Metrics . . . . . 11
Quality of Service . . . . . 12
DiffServ . . . . . 13
IntServ . . . . . 14

3. QUALITY AWARE SOURCE ROUTING . . . . . 15

Overview . . . . . 15
Neighborhood Information Exchange . . . . . 16
Location . . . . . 16
Flow State . . . . . 17
Protocol Overhead Estimation . . . . . 17
Protocol Overhead Rate Control . . . . . 18
Yang's Bandwidth Estimation . . . . . 20
Yang's Delay Estimation . . . . . 22
Route Metric . . . . . 24
Route Discovery . . . . . 25
Discovery Filter . . . . . 26
Route Selection . . . . . 27

4. SIMULATION . . . . . 28

Scenarios . . . . . 28
Random Waypoint . . . . . 29
Grand Canyon Junction . . . . . 29
Wind River Canyon . . . . . 31
Black Mountain . . . . . 32
Code Configuration . . . . . 33
Results . . . . . 34

TABLE OF CONTENTS – CONTINUED

5. ANALYSIS . . . . . 41

Throughput . . . . . 41
Overhead . . . . . 42
Delay . . . . . 44
Jitter . . . . . 45
Packet Delivery Ratio . . . . . 46
QoS Acceptance Ratio . . . . . 47

6. CONCLUSION . . . . . 48

Future Work . . . . . 49
Location Awareness . . . . . 49
Terrain Awareness . . . . . 49
Parameter Tuning . . . . . 49
Priority Discrimination . . . . . 50
Background Priority Traffic . . . . . 51
Cross-Layer Optimization . . . . . 51

REFERENCES . . . . . 52

APPENDICES . . . . . 54

APPENDIX A QASR Implementation Details . . . . . 55

# LIST OF TABLES

Table Page

1 Delay Estimation Parameters [1] . . . . . 23

2 Simplified Delay Estimation Parameters . . . . . 24

3 Simulation Execution Parameters . . . . . 33

# LIST OF FIGURES


| Figure | | Page |
|---|---|---|
| 1 | The OSI Network Stack Model | 4 |
| 2 | Example of a TDMA Transmission Schedule | 5 |
| 3 | View of a CSMA Packet Transmission Process | 6 |
| 4 | Interference Model Showing Transmission and Interference Ranges | 7 |
| 5 | QSR Protocol Overhead as a Function of Neighborhood Density | 20 |
| 6 | Vehicles Placed on Roadways Near Grand Canyon Junction | 30 |
| 7 | Rescue Workers Searching the Wind River Canyon | 31 |
| 8 | Emergency Workers Fighting a Wildfire on Black Mountain | 32 |
| 9 | Random Waypoint MANET Node Density Performance Statistics | 35 |
| 10 | Random Waypoint MANET Traffic Demand Performance Statistics | 36 |
| 11 | Grand Canyon Junction MANET Node Density Performance Statistics | 37 |
| 12 | Grand Canyon Junction MANET Traffic Demand Performance Statistics | 38 |
| 13 | River Search MANET Traffic Demand Performance Statistics | 39 |
| 14 | Black Mountain MANET Traffic Demand Performance Statistics | 40 |

# LIST OF ALGORITHMS


| Algorithm | | Page |
|---|---|---|
| 1 | Simple Retransmit Algorithm | 18 |
| 2 | Protocol Overhead Rate Control Algorithm | 19 |
| 3 | Yang's Local Available Bandwidth Estimation [2] | 21 |
| 4 | Yang's Neighborhood Available Bandwidth Estimation [2] | 22 |
| 5 | QASR Route Metric | 24 |
| 6 | Route Discovery Admission Control | 26 |

# ABSTRACT

We evaluate several routing protocols, and show that the use of bandwidth and delay estimation can provide throughput and delay guarantees in Mobile Ad Hoc Networks (MANETs). This thesis describes modifications to the Dynamic Source Routing (DSR) protocol to implement the Quality Aware Source Routing (QASR) network routing protocol operating on an 802.11e link layer. QASR network nodes exchange node location and flow reservation data periodically to provide information necessary to model and estimate both the available bandwidth and the end-to-end delay of available routes during route discovery. Bandwidth reservation is used to provide end-to-end Quality of Service, while also utilizing the differentiated Quality of Service provided by the 802.11e link layer. We show that QASR performs significantly better in several performance metrics over the DSR and the Ad-Hoc On-Demand Distance Vector (AODV) protocols, and performs more consistently in all quality metrics when traffic demand exceeds network capacity.

# 1

# INTRODUCTION

Wireless networking has been an increasingly active topic for research and development in recent years, as the technology becomes more compact, less demanding of power, and generally more pervasive. More devices than ever employ the 802.11 wireless technology, which is well suited to support ad hoc networks. PDAs, phones, and laptops can all put multimedia demands on a wireless network. Streaming video and VOIP applications have become commonplace. Urban wireless "hotspots" are widely available, and network users have come to expect wireless network service from their environment.

Emergency workers such as fire fighters, police, and paramedics need access to information in order to do their jobs as safely and effectively as possible [3]. Modern emergency workers are coming to expect constant access to a wireless data network that is fast and reliable. The needs of emergency workers are both diverse and demanding. Police officers may commonly require access to vehicle record databases. Paramedics may require real-time medical data flows to a destination hospital, while also looking up patient medical histories. Fire fighters may need voice communication in a diverse range of rural environments.

While Mobile Ad Hoc Networks (MANETs) are not currently used for emergency multimedia traffic, MANET technology can potentially provide data access in rural environments where traditional infrastructure is not cost effective. Issues that stand in the way of MANET adoption include limited range of 802.11 radios and questionable network reliability.

The issues and limitations of wireless data networks are most pronounced in a rural environment, where fixed infrastructure does not exist. The questions of connectivity

and reliability of wireless ad hoc networks in realistic scenarios must be addressed before they can be seriously considered as a communications option.

In this thesis, we will evaluate techniques and algorithms that provide improved reliability to wireless networking. In particular, we will focus on rural environments for performance analysis. Rural environments are characterized by irregular terrain that is often poorly accessible, and has little connectivity (wireless or otherwise) to existing network infrastructure. An example rural scenario where a wireless network could be helpful is in the fighting of a mountain wildfire.

# 3

# WIRELESS DATA NETWORKING

Wireless data networking exists in a variety of forms. To place the work of this thesis in context, relevant technologies are presented and compared in this chapter. By contrasting similar technologies we will provide the reader with both a justification for the work of this thesis as well as brief review of the field of wireless data networking.

The components of data networks are generally classified in terms of their position in the Open Systems Interconnection (OSI) model (Figure 1). In this section we will describe the operation of two very different link layers, then describe the operation of two similar network protocols. The remaining layers above the network layer will not be addressed in this thesis beyond their role in placing demands on the network layer.

## Wireless Link

A Medium Access Control (MAC), or data link layer in a network stack is responsible for providing single hop communication between two nodes. The link layer determines how a radio is used to move data between nodes. Both WiFi routers and cell-phone systems can provide reliable link layer connectivity to end users. Cell systems use proven technologies to cover large areas, and support many users concurrently with consistent performance. Compared to a cell system, a single WiFi access point cannot cover as much area, nor support as many users at a time. However, WiFi radios are much smaller, cheaper, and more mobile than cell system, and can be deployed quickly. In a rural emergency scenario, however, wireless solutions need the reliability and coverage area of a cell system, and the flexibility of mobile ad hoc wireless.

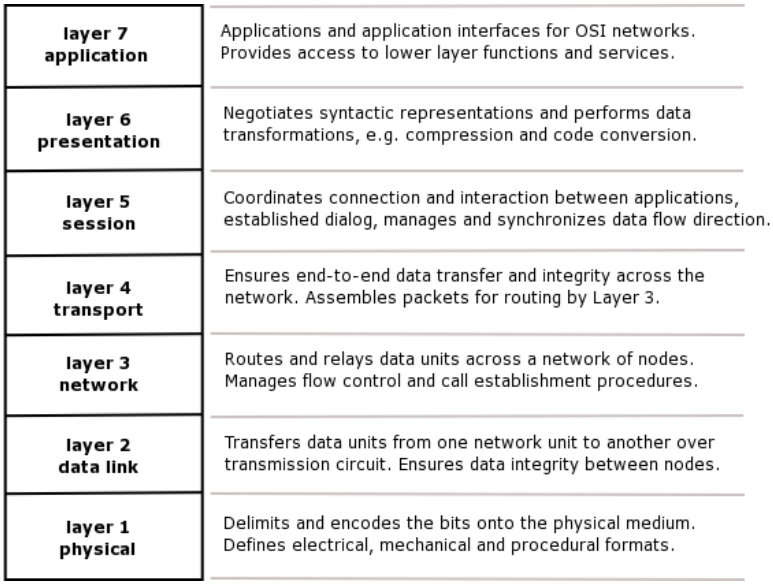


Figure 1: The OSI Network Stack Model

TDMA

Cell system radio links frequently operate on *Time Division Multiple Access* (TDMA), which segments the available airtime into a repeating cycle of time slices. The TDMA link protocol provides a fixed infrastructure point with efficient use of radio link capacity.

This link layer protocol provides excellent quality of service for two reasons: calls have consistent bandwidth and zero jitter. Consistent bandwidth is guaranteed by the allocation of time slices to a handset during call setup. Zero jitter is inherent to the regular structure of the time slices. These features provide excellent voice quality and has been proven through widespread deployment of TDMA-based technologies such as GSM in the United States and Europe.

Very efficient time-slot scheduling algorithms provide for near optimal usage of wireless spectrum. An example schedule is given in Figure 2. TDMA networks use centralized algorithms to facilitate both the placement of cell sites, as well as the

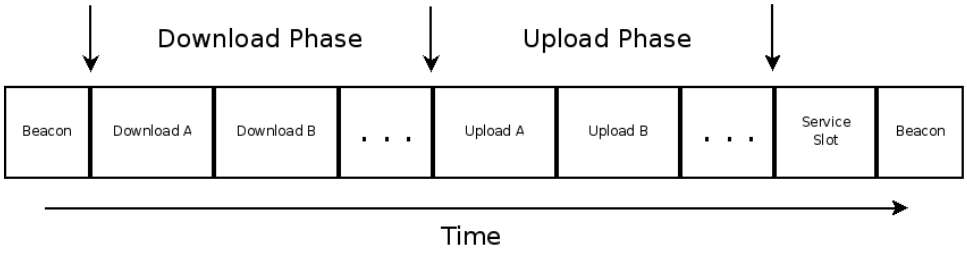


Figure 2: Example of a TDMA Transmission Schedule

time-slot schedules each system may transmit in.

TDMA infrastructure is ideal for multimedia applications such as streaming video and voice applications. Additionally, routing considerations are relatively simple as the infrastructure is immobile. Ideally, TDMA network infrastructure would provide network coverage wherever emergency workers are deployed. The cost of deploying infrastructure in rocky and irregular terrain is prohibitive, however. Irregular terrain leads to "dead zones" where no infrastructure connectivity exists, and the cost of covering these zones grows dramatically with the irregularity of the terrain.

Mobile Ad hoc Networks (MANETs) are networks of peers that utilize neighbors to retransmit data, rather than fixed infrastructure. MANETS must be capable of restructuring themselves arbitrarily at any time. The complexity of scheduling transmissions in a mobile ad hoc TDMA network is known to be NP-complete [4, 5] and generally only solved in stationary, centrally coordinated, or otherwise planned environments [6]. Until a satisfactory distributed solution can be found, a more flexible link technology that is more tolerant of mobility should be explored.

### CSMA/CA

The IEEE 802.11 MAC, *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA), is well suited to an ad hoc network. The sequence of operations is

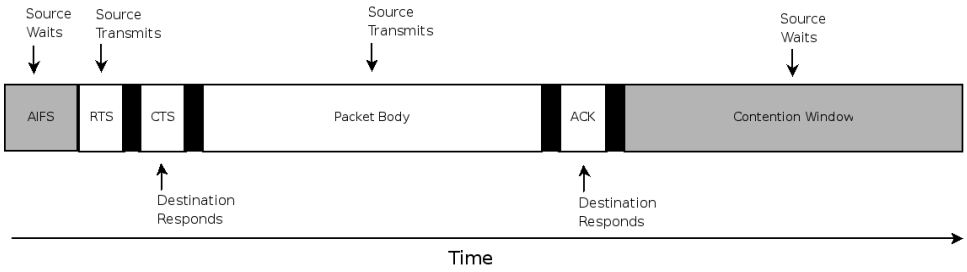


Figure 3: View of a CSMA Packet Transmission Process

diagrammed in Figure 3. It is entirely decentralized, with the notion that each node in the network is a peer of other nodes that may be in the area. When a node has data to transmit, it listens to the channel for some short duration to determine if there is any other traffic on the network. If there is none, the node sends a very short Request-To-Send (RTS) packet, which describes the length of the packet to be sent, as well as the destination. This packet is short to reduce the risk that it might collide with another packet, or in other words, for collision avoidance. The target node is then expected to send a Clear-To-Send (CTS), which also contains the length of the data packet, for the benefit of nodes within range of the receiver node, but not within range of the transmitting node. The RTS/CTS pair informs all nodes within the ranges of either side of the communication that a packet will be in transmission for a known interval. Neighbor nodes are expected to respect this interval, even if they cannot sense traffic during the interval, which facilitates collision avoidance. When the packet body has been received correctly, the recipient replies with an Acknowledgment (ACK).

The inherently decentralized coordination of CSMA does not provide as efficient bandwidth usage as TDMA, but it is fundamentally more robust to changes in network topology. With careful network management, reasonable and consistent network results can be achieved. Network management failure can result in very

poor performance of this MAC protocol.

The 802.11 protocol defines various parameters of the radio signal encoding as well as CSMA/CA parameters such as wait times. Wait times in the CSMA scheme are random within certain protocol defined windows. This produces a probabilistic system that can only be effectively measured by statistical methods. Some of these methods will be discussed later in this thesis.

Interference Model

WiFi networks operate over a common link channel, meaning that only one node can successfully transmit in a particular area at a time. When two transmissions are received simultaneously at a node, those transmissions are said to collide, as the signaling on the channel mixes, destroying the data in both packets. The *transmission range* is the area around a node in which other nodes can receive packets. It is sometimes called a one-hop radius. The *interference range* of a node is the area around the node in which is is possible for a transmission from that node to produce a collision at another node.

To model the effect of interference on the performance of the network, we must consider the interference range of nodes. An accurate model would consider the effect of terrain, weather, frequency, transmit power, and other variables that effect radio propagation. For the sake of simplicity, we use a protocol model where the interference range is a circle with a radius of twice the transmission radius. A node $x$ is said to be *in the neighborhood* of $y$ if $x$ is *within the interference range* of $y$. In Figure 4, nodes A, B, and C are all included in the interference range of node A, and are therefor within the neighborhood of A.

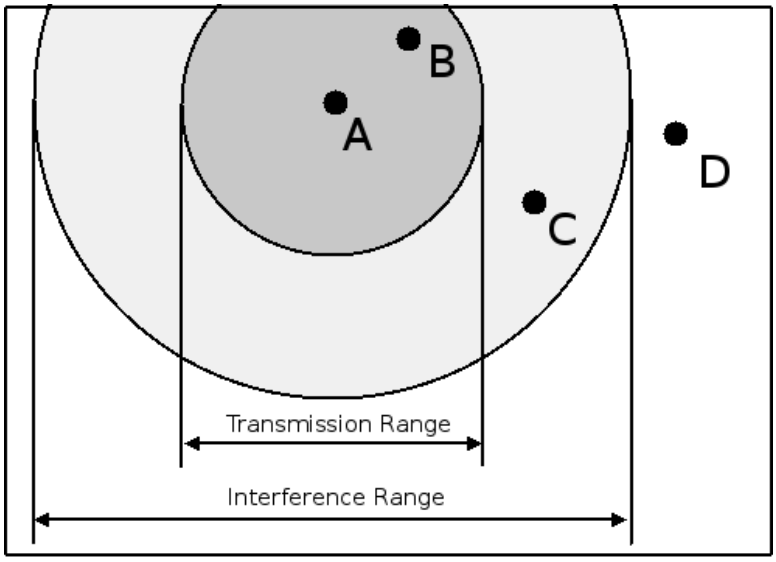


Figure 4: Interference Model Showing Transmission and Interference Ranges

## Wireless Routing

It is the purpose of this thesis to implement Quality of Service awareness into a wireless routing protocol, and evaluate the performance of the resulting protocol against unmodified protocols. In this section we will introduce the unmodified protocols, and describe their methods of operation. There are a number of methods for routing in wireless mesh networks [7]. The fundamental goal is the same, which is to discover routes from source nodes to destination nodes through a potentially mobile set of peers, which constitute the network, then use those routes to move packets across the network. Two common protocols are Dynamic Source Routing (DSR) and Ad-Hoc On-Demand Distance Vector (AODV) routing[7]. DSR is the base protocol on which the work for this thesis (QASR) is based. The mesh standards being developed for IEEE 802.11s inherit much of their behavior from AODV. In this section we describe the fundamental operations of DSR and AODV. AODV and DSR are both

known as a reactive protocols because they do not gather or store information about the structure of the network in the absence of traffic demand. Combined with route error detection, these protocols can repair themselves quickly and provide seemingly continuous routing to mobile users, with no control traffic overhead when users place no demand on the network.

DSR

In a source routing protocol such as DSR, it is the responsibility of the source node to explore the topology of the network when a route is needed. When a path is discovered, the burden of storing the route remains with the source node. Subsequently, in order to route packets via intermediate nodes, the routing data for each packet is encapsulated within each packet. This encapsulation does impose some traffic overhead.

In the DSR protocol, network routes are discovered using a flood-search approach. A Route Request (RREQ) packet is broadcast by a source node. Each intermediate node that correctly receives the RREQ re-broadcasts the packet with the exception of the destination node. At each intermediate node the rebroadcast packet is modified to add that node's address to the source-route list. To prevent cycles in the network, intermediate nodes should not rebroadcast a given RREQ more than once.

The destination node responds to the RREQ with a RRPLY packet. A RRPLY packet is returned to the source node not by a broadcast-flood, but instead by following the source-route list in reverse.

The DSR protocol also specifies that nodes that are not on the source-route list may cache the route for later use. Route caching is used to accelerate the route discovery process by allowing intermediate nodes to return previously discovered routes to source nodes before the RREQ node reaches the destination. Multiple,

differing RRPLY messages can be returned to the source node, leaving the decision of which route to use to the source node. The primary metric for route selection when multiple routes are available is minimum hop count.

An error occurs when a node in the source route cannot deliver a packet to the next hop designated in the source list. When an error occurs, a RERR message is created and sent via the prior hops listed in the source route. When a RERR is received by the source node, it means that the particular route is broken, and the source node must perform a new route discovery.

Routes are maintained by intermediate nodes sending Acknowledgment Request (ACKREQ) packets to the next-hop node on the route. If an ACKREQ times out, traffic is held in a buffer while subsequent ACKREQ packets are sent. After some number of retries, the buffered traffic is discarded, and a RERR is sent to the source node.

AODV

AODV uses a similar flood-search approach. The fundamental difference between DSR and AODV is that AODV distributes the storage for discovered routes throughout the network. This means that rather than the IP packet being augmented to encapsulate routing data, the intermediate nodes retain memory of the next hop along the route in routing tables. An advantage of this approach is the empowerment of intermediate nodes to repair broken routes locally, rather than strictly depending on the source node for routing information. Additionally, there is no route encapsulation overhead, as all the intermediate nodes store the data they require.

The AODV protocol begins a route exploration by initiating a RREQ broadcast flood. When the destination node receives the request, it transmits a RRPLY packet containing a distance-vector of zero. Neighboring nodes increment the distance-vector, add the destination node to their routing-tables, and retransmit the RRPLY.

When the RRPLY flood is complete, each node in the network should contain the destination node in its routing table with the lowest distance vector RRPLY it received during the flood. The source node may begin routing traffic as soon as its routing table contains an entry for the destination node, and the node can transmit data packets to the appropriate one-hop neighbor. The table entry can also be updated in the event that improved routes are discovered, even if another node initiates the discovery process. Nodes with unused table entries clear the entries after some timeout of about 3 seconds.

Routes in AODV are maintained using periodic *hello* packets, similar to ACKREQ packets in DSR. When a critical number of *hello* packets are sent without a response, newly arrived traffic is stored in a temporary buffer while route repair operations are attempted[1]. If these operations fail, the buffered traffic is discarded, and a RERR is sent to the source node.

The critical number of lost *hello* packets is a user-defined parameter which effectively controls how quickly the network fails broken routes. A small value will result in fast link breakage, and may add unnecessary route discovery traffic during periods of network congestion. A large value will cause noticeable service interruption as failed links are used until sufficient *hello* packets fail.

Metrics

To select a route from the set of feasible routes, a metric must be used to sort the various routes by quality. A *cost metric* is one in which minimal cost is desired. Both

[1]In the OPNET 12.1 implementation, AODV will continue to transmit packets until the critical number of *hello* packets failures is reached. In the DSR implementation, traffic is cached when the first ACKREQ fails.

DSR and AODV use hop count as their cost metric, meaning they select the route with the fewest intermediate nodes to the destination.

Additional information can be used to compute a routing metric, such as node location, velocity, and traffic load. More sophisticated routing metrics can be constructed by modeling the traffic in such a way as to estimate the available bandwidth along a route, as well as predict the average end-to-end delay of a particular route. Routing metrics are generally cost metrics, meaning that lower values are preferred over higher values such as with hop-count and delay.

### Quality of Service

In networking, Quality of Service (QoS) generally refers to guarantees in the throughput and delay of packet streams within a network. By providing these network quality guarantees, networks can support multimedia services such as streaming video and VOIP. Other metrics that can be used to define QoS are jitter in the packet latency, packet loss rate, and route discovery delay.

Jitter is an important metric when considering QoS, and can be measured in a number of ways. The jitter measure used in this thesis is "cycle-to-cycle" jitter. This measure is taken by recording the difference in end-to-end delay of two successive packets of the same flow. For example: if a packet arrives at the destination node having taken $40ms$ to traverse the network, and the following packet from the same flow takes $30ms$, then the jitter for the second packet is $10ms$.

The two most common forms of QoS provisioning on land-based IP networks are DiffServ and IntServ [8]. Providing QoS on a wireless ad hoc network is challenging [9, 10, 11]. These methods effectively address network *saturation*. When a network is described as becoming saturated, it generally means that the capacity of a network

has been reached, and no additional traffic can be supported. In this thesis we use a more strict, node-centric definition of saturation. A node in a network is in the *saturation condition* if the transmission queue size is greater than 1. This means a node is saturated if there are any packets that are waiting for other packets to be complete transmission, rather than waiting on the wireless link to complete their own transmission.

DiffServ

DiffServ[12] is a simple provisioning scheme that describes a bias between classes of service. DiffServ creates a set of 8 *Differentiated Services* that are specified by 3 bits in the Type of Service field of the IP packet header[13]. Each routing node along the path prioritizes traffic according to this differentiation. The saturation management provided by DiffServ is that higher priority packets will be transmitted before low priority packets, even if the lower priority packets were queue earlier.

The IEEE 802.11e Quality of Service provisioning operates on the DiffServ model. This specification describes differentiation of 4 priorities by defining separate data queues. the mapping from 8 DiffServ priorities to 4 802.11e priorities is done using a simple mapping where the top 2 DiffServ priories map to the top 802.11e priority, and so on.

Each 802.11e priority class also redefines certain parameters in the 802.11 protocol. The most important of these parameters is the *contention window* (CW) size. Packet streams with longer CW durations generally see longer average packet delays than streams with shorter CW durations. The effect of these parameters changes results in lower average packet transmission delay for high priority traffic, compared to lower

priority traffic[2].

802.11e does not provide end-to-end QoS. It was designed primarily for use in WiFi *access points*, which are wireless gateways that centralize and coordinate access to high capacity wired networks. Communication with an access point is single-hop, meaning that only nodes within range of the access point radio have access to its resources. Multi-hop (or end-to-end) QoS requires additional support from the network layer in the OSI model.

IntServ

One example of QoS support at at the network layer is IntServ [14]. In the IntServ provisioning system, RSVP [15] is used to reserve bandwidth along the route that will be used. RSVP (from the French phrase "*Rpondez S'il Vous Plat*", or "*please respond*") requires explicit coordination on the part of all the nodes along a desired route. Intermediate routing nodes communicate to reserve the required network capacity. Capacity is provided at each node by assigning tokens to the flow at a fixed rate. When a route is active these tokens are consumed by routing packets for the data flow. The unused tokens can be described as being stored in a bucket, where the depth of the bucket allows for bursty data flow, but still restricts the average data rate of the flow. A video stream, for instance, may transmit bursts of packets for each frame, at a steady rate of ten frames per second. The RSVP method of resource allocation prevents *overbooking*. Overbooking is the acceptance of too much traffic onto a network at some location such that some nodes in the vicinity become saturated.

[2]Lower average delays can only be expected in a heterogenous traffic environment. If all traffic is of high priority, than the short contention window size will result in more frequent packet collisions, and thus longer average delays will result.

Explicit bandwidth reservation protocols such as IntServ require memory and computational resources on the part of intermediate routing nodes within a network in order to provide end-to-end QoS. In exchange for this resource overhead, superior QoS guarantees can be provided relative to differentiated QoS, which can only be implemented by prioritizing traffic at each link interface.

# 16

# QUALITY AWARE SOURCE ROUTING

In this chapter we will present a MANET protocol based on DSR developed for the purpose of evaluating the incorporation of Quality of Service provisioning in a mesh routing protocol. The objective is to satisfy connection requests between a source and a destination node for a flow with known bandwidth, delay, and jitter requirements. We are constrained by the service requirements of preexisting flows, such that meeting the requirements of a new connection request cannot disrupt any preexisting flows. Finally, we wish to minimize the average end-to-end delay for all flows in the network, maximize the total routing capacity of the network, and maximize the longevity of discovered routes.

## Overview

Quality Aware Source Routing (QASR) incorporates RSVP-like bandwidth reservation into the DSR route discovery process. QASR nodes share location and flow state data locally to provide bandwidth reservations during route discovery. Nodes use the data collected from neighbor nodes to compute an available bandwidth estimate and end-to-end delay estimate and perform admission control operations during the route discovery process.

The QASR routing metric is a weighted sum of the estimated end-to-end delay, minimum available bandwidth, and node speed. This provides route selection that is biased against long routes, network congestion and potentially short lived links. In contrast, the DSR routing metric a minimum hop count.

At each intermediate node, when a RREQ passes admission control the RREQ packet is updated to provide the destination node with end-to-end QoS data. The

data includes the minimum available bandwidth computed along the route and the sum of the estimated average packet delays at each hop. Additionally, the QASR routing metric is computed and the value added to the end-to-end cost metric of the RREQ.

The QASR route discovery process includes several modifications to DSR. First, route caching is completely disabled. This ensures that bandwidth and delay estimates are accurate for every route discovery. Second, a destination node does not respond immediately to each RREQ it receives. Instead, the destination node aggregates RREQ packets for a short period, then selects the best route according to the QASR routing metric before returning a RRPLY.

When a route is accepted and the source node uses the route, the source node and each intermediate node will then update their flow state data. The neighborhoods of those node are then updated with the new flow data through the neighborhood information sharing process.

### Neighborhood Information Exchange

Nodes in QASR periodically share location and flow state information with their neighbors. For the sake of brevity this process will be refereed to as Information eXchange (IX).

#### Location

To determine if a node is within the interference range of another node, each node must share its location information with it's neighbors. As a result, nodes must have some way to determine their location. We assume that all nodes in the network have

an absolute location service, such as GPS. For this thesis, development was within the OPNET simulation environment which provides this information through its API.

Flow State

To model the state of a neighborhood's traffic conditions, each node includes a summary of the traffic in that it routes. The summary is broken into priority classes to allow distinction between the various real-time traffic priorities. This provides nodes with a simplified view of the quantity of traffic in their neighborhood. Although the separation of traffic into various priorities does not currently effect traffic handling in QASR, a discussion of the utility of priority distinction is discussed in the Future Work section of Chapter 6.

Protocol Overhead Estimation

It is not necessarily possible for a node to broadcast its location and flow state packets directly to all the nodes in its interference neighborhood. As a result, neighbor nodes must rebroadcast the packets in order to ensure each node in the neighborhood to receives the information. Inherent in the IX process is the addition of *protocol overhead* to the traffic load on the network. This must also be modeled and included within the IX packet data if we wish to preserve the accuracy of our flow state measurement. We present two algorithms for rebroadcasting packets and their respective bandwidth models. The first is shown in Algorithm 1.

If we define $\mathbf{N}$ as the set of nodes in the neighborhood of node $n_i$, and the number of nodes in that neighborhood as $|\mathbf{N}|$, then the number of packets generated in a neighborhood per IX period for $n_i$ can be described by Equation 1.

$$(|\mathbf{N}| - 1)|\mathbf{N}| + |\mathbf{N}| = |\mathbf{N}|^2 \tag{1}$$

$\mathbf{N} \leftarrow neighborhood(n_i)$
**for all** $n_i \in \mathbf{N}$ **do**
transmit an IX packet every $IX_{\text{period}}$ seconds
**for all** $n_j \in \mathbf{N}, n_i \neq n_j$ **do**
store IX data
retransmit IX packet
**end for**
**end for**

Algorithm 1: Simple Retransmit Algorithm

Equation 1 shows that the number of IX packets transmitted grows by the square of node density. To account for this traffic, each node computes the overhead from IX traffic it generates, including an estimate of rebroadcasts of neighbor data. If $IX_{\text{period}}$ is measured in seconds, than the IX packets produced by each node per second $IX_{\text{traffic}}$ is given by equation 2.

$$IX_{\text{traffic}} = \frac{|\mathbf{N}|^2}{IX_{\text{period}}} \tag{2}$$

While this may be acceptable for small networks, is not a scalable solution. In the following section, a modification to the rebroadcast algorithm is presented and analyzed.

Protocol Overhead Rate Control

To alleviate the overhead traffic growth of the Simple Retransmit Algorithm, we present a modification to that algorithm called Protocol Overhead Rate Control (PORC), shown in Algorithm 2.

Ideally, the protocol overhead would be constant and therefore independent of node density. It is not feasible to bound $|\mathbf{N}|$, however, and so $O(|\mathbf{N}|)$ is the best

$\mathbf{N} \leftarrow neighborhood(n_i)$
**for all** $n_i \in \mathbf{N}$ **do**
transmit an IX packet every $IX_{\text{period}}$ seconds
**for all** $n_j \in \mathbf{N}, n_i \neq n_j$ **do**
store IX data
$p_r \leftarrow \frac{C}{|\mathbf{N}|}$
$rand \leftarrow rand(0,1)$
**if** $rand < p_r$ **then**
retransmit IX packet
**end if**
**end for**
**end for**

Algorithm 2: Protocol Overhead Rate Control Algorithm

that can be achieved. We must then choose a subset of **N** the will retransmit the IX packets to produce an $O(|\mathbf{N}|)$ growth in protocol overhead.

A node includes itself in the subset of **N** using probability computed from $\frac{C}{|\mathbf{N}|}$, where C is average number of rebroadcasts desired for each unique IX packet. The traffic from this subset can be then modeled as shown in Equation 3.

$$\left(\frac{C}{|\mathbf{N}|}\right)(|\mathbf{N}|-1)|\mathbf{N}| + |\mathbf{N}| = (C+1)|\mathbf{N}| - C \tag{3}$$

It can be seen from Equation 3 that overhead growth as the node density of a neighborhood increases grows linearly. The $IX_{\text{traffic}}$ is then given by Equation 4.

$$IX_{\text{traffic}} = \lim_{n\to\infty} \frac{(C+1)|\mathbf{N}| - C}{|\mathbf{N}| \times IX_{\text{period}}} = \frac{C}{IX_{\text{period}}} \tag{4}$$

For any value of $C$ greater than 1, the neighborhood protocol overhead will grow quadratically for $0 < |\mathbf{N}| \leq C$, then transitions to linear growth for $|\mathbf{N}| > C$. The per-node overhead contribution will asymptotically approach $C+1$ packets per $IX_{\text{period}}$ as $n$ grows. To verify this, a set of traffic-free simulations were run with increasing node density where all nodes are very close together (all nodes are within one hop)

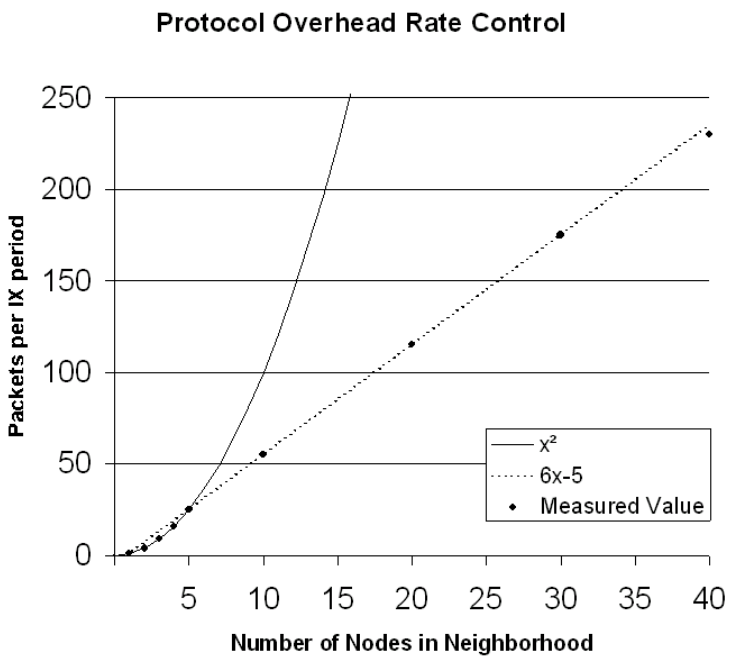


Figure 5: QSR Protocol Overhead as a Function of Neighborhood Density

with the PORC constant $C = 5$. The measured overhead is plotted in Figure 5 against the functions used to model the overhead. It can be seen that the overhead is linear when the node density is greater than 5.

## Yang's Bandwidth Estimation

Yang's bandwidth estimation model [2] is used to conduct admission control for new routes to prevent overbooking. Admission control is performed by estimating the available bandwidth for a new RREQ, and comparing it to the bandwidth requested by the RREQ. If the request cannot be met, the RREQ is simply dropped at the intermediate node with no further action, otherwise the RREQ is handled according to the DSR protocol rules.

Upon receipt of a RREQ, Yang's algorithm computes two values: the *local* available bandwidth, and the *neighborhood* available bandwidth. The lesser of these two values is used as the upper bound for the route acceptance decision. If the flow requirement exceeds the available bandwidth, the RREQ is dropped without further action.

We present Yang's Local Available Bandwidth Estimate [2] in Algorithm 3. This algorithm predicts the local available bandwidth given the total bandwidth of the shared channel $B$, the vectors $\mathbf{R}, \mathbf{L}, \mathbf{W}$ containing the rates of traffic flows in the neighborhood, the size of the flow packets, and the contention window sizes, respectively. The last parameter is $\alpha$, the number of hops the potential route would have have within the same neighborhood.

*local_available_bandwidth*$(B, \mathbf{N}, \mathbf{R}, \mathbf{L}, \mathbf{W}, \alpha)$
**for** $i = 1$ **to** $|\mathbf{N}|$ **do**
$\eta_i^* := \frac{B}{\mathbf{W}_i \mathbf{R}_i}$
**end for**
$Sort(\mathbf{R}, \eta^*); Sort(\mathbf{L}, \eta^*); Sort(\mathbf{W}, \eta^*)$
$V_f \leftarrow \alpha \frac{L_f}{W_f}$
$X_0 \leftarrow 0$
$Y_0 \leftarrow \sum_{i=1}^{n} \frac{\mathbf{R}_i}{\mathbf{L}_i} B$
**for** $i = 1$ *to* $|\mathbf{N}|$ **do**
$X_i \leftarrow X_{i-1} + \frac{\mathbf{L}_i}{\mathbf{W}_i}$
$Y_i \leftarrow Y_{i-1} - \frac{\mathbf{R}_i}{\mathbf{L}_i} B$
$V_{f,i}^* \leftarrow \eta_i^*(1 - Y_i) - X_i$
**if** $V_{f,i-1}^* \leq V_f < V_{f,i}^*$ **then**
$\eta \leftarrow \frac{X_{i-1} + V_f}{1 - Y_{i-1}}$; $BREAK$
**end if**
**end for**
$U_f^1 \leftarrow \frac{V_f}{\alpha \eta} B$
**return** $U_f^1$
Sort(*array*,*index*)
Sort *array* in ascending order by *index*

Algorithm 3: Yang's Local Available Bandwidth Estimation [2]

Next we present Algorithm 4, Yang's Neighborhood Available Bandwidth Estimate[2].

The available bandwidth for a flow is calculated as the minimum of the estimates from Algorithm 3 and Algorithm 4.

The additional parameter $\gamma$ is supplied to Algorithm 4 identifying a particular node. The meanings of the terms $\alpha$ and $\eta_i^*$ are consistent with their meaning in Algorithm 3.

$neighborhood_available_bandwidth(B, \mathbf{N}, \mathbf{R}, \mathbf{L}, \mathbf{W}, \alpha, \gamma)$

$\eta_\gamma^* \leftarrow \frac{B}{\mathbf{W}_\gamma \mathbf{R}_\gamma}$
$X \leftarrow \sum_{j:\eta_j^* \leq \eta_\gamma^*} \frac{\mathbf{L}_j}{\mathbf{W}_j}$
$Y \leftarrow \sum_{i:\eta_i^* \leq \eta_\gamma^*} \frac{\mathbf{L}_i}{\mathbf{W}_i}$
$U_f^n \leftarrow \frac{B}{\alpha}[(1-Y) - \frac{1}{\eta_\gamma^*}X]$
**return** $U_f^n$

Algorithm 4: Yang's Neighborhood Available Bandwidth Estimation [2]

## Yang's Delay Estimation

Yang has also published an algorithm to model and estimate the average delay for traffic at each hop in a network [1]. The delay estimate is used both for denial of routes with excessive delay, and as an element in the QASR routing metric. The end-to-end delay is estimated by summing the estimates computed at each hop. This estimation algorithm uses the same flow rate data collected for bandwidth estimation and does not impose any additional protocol overhead.

Yang's Delay Estimate is presented in Equation 5, as the estimate for the average per-packet delay, $d_i$, at a particular node. The parameters are listed and described in Table 1.

| | |
|---|---|
| $T_d$ | time in seconds to send a packet |
| $\epsilon$ | $slottime$ |
| $\lambda_i$ | packet transmission rate at node $n_i$ |
| $x_i$ | physical transmission rate |
| $\alpha_{j_i}, \beta_{j,i}$ | Discount Factors |
| $p_i^b$ | $\sum \alpha_{j,i} \frac{\lambda_j}{x_j} + \frac{\lambda_i}{x_i}$ |
| $\gamma$ | $\frac{\epsilon}{1.1788(T_d+\epsilon)}$ |
| $G_{i,j}$ | $\beta_{i,j}\lambda_j T_d$ |
| $W_i$ | contention window size at node $n_i$ |
| $H_1$ | $AIFS$ period |
| $H_2$ | $1.1788(T_d + \epsilon)p_i^b$ |

Table 1: Delay Estimation Parameters [1]

$$E(d_i) = H_1 + H_2 \left[ 1 + \gamma - \prod_{j \in \mathbf{N}} (1 - G_{i,j} \frac{2}{W_j}) \right] \frac{W_i}{2} \tag{5}$$

The $\alpha$ and $\beta$ discount factors represent the probability that two nodes that interfere with $n_i$ also interfere with each other. These values can be computed, but for the sake of simplicity in QASR, $\alpha$ and $\beta$ are assumed to be 1.0, thus leading to a pessimistic estimate of the delay.

For additional simplicity, packet sizes are assumed to be constant, at 1024 bits per packet. Given a transmission rate of 1Mbps using the 802.11 protocol, including the RTS/CTS, we can compute $T_d = 1.62ms$, and simplify several dependent expressions. Given the parameters from Table 2, the delay estimate simplifications produce the parameters in Table 2, and Equation 6, which was implemented in QASR for this thesis.

$$E(d_i) = H_1 + H_2 (\sum_{i}^{|\mathbf{N}|} \frac{\lambda_i}{x_i}) \left[ H_3 - \prod_{j}^{|\mathbf{N}|} (1 - H_4 \frac{\lambda_j}{W_j}) \right] \frac{W_i}{2} \tag{6}$$

| | |
|---|---|
| $H_1$ | 1 $\mu$Sec |
| $H_2$ | 1.922 $mSec$ |
| $H_3$ | 1.0105 |
| $H_4$ | 2.62 $mSec$ |
| $x_i$ | physical transmission rate |
| $\lambda_i$ | packet transmission rate at node $n_i$ |
| $W_i$ | contention window size at node $n_i$ |

Table 2: Simplified Delay Estimation Parameters

Route Metric

The routing metric of a flow $f$ in QASR is the end-to-end sum of a weighted triple consisting of the estimated delay, percent bandwidth consumed, and node mobility, which is measured as speed. Delay is normalized a peak delay parameter $D$. Bandwidth is normalized by the total bandwidth for the channel $B$. Speed is normalized by an estimated top speed of $S$. These parameters are then mixed by the constants $K_D$, $K_B$, and $K_S$, as shown in Algorithm 5.

$per_hop_route_metric(n_i, f, D, B, S)$
$B_i \leftarrow \frac{B - available_bandwidth(n_i)}{B}$
$D_i \leftarrow \frac{delay_estimate(f)}{D}$
$S_i \leftarrow \frac{speed(i)}{S}$
$M_i \leftarrow K_B B_i + K_D D_i + K_S S_i$
**return** $M_i$

Algorithm 5: QASR Route Metric

The values for $K_B$, $K_D$, and $K_S$ are tunable parameters. For the evaluations in this thesis, they are each equal to $\frac{1}{3}$. Tuning of these parameters is discussed further in the Future Work section of Chapter 6.

## Route Discovery

QASR routing operates on the same principles as Dynamic Source Routing (DSR) in that the sequence of hops each packet takes along its route is included with each packet. In this way, the source node of a flow determines the flow's route.

A route discovery begins when the networking layer receives application layer data and a known route is unavailable. The QoS requirements for the data are assumed to be provided with the data.

Like the DSR protocol, QASR floods the network with RREQ packages, seeking a route to the destination node. QASR then applies admission control at the source node, and each intermediate node. In this phase of the discovery, each node examines the RREQ packet, making use of the partial-path, the destination address, and the partial-path aggregate metrics such as route-cost metric and the end-to-end estimated delay.

The bulk of the Quality Awareness in QASR is exhibited by Route Discovery Admission Control, in Algorithm 6. Algorithm 6 is used to decide if a flow request $f$ will be accepted at the intermediate node $n_i$, or denied to prevent overbooking neighborhood resources.

In contrast to the DSR protocol, QASR allows multiple RREQ per unique flow per node. In DSR, if a node is engaged in a route exploration for some destination node, it will not broadcast subsequent RREQ packets for that destination. This feature is necessary for QASR to ensure that a second flow does not cause overbooking by sharing a previously discovered route. A consequence of this is that data may take multiple paths from a source node to a destination node, if the data belong to multiple flows.

Given a flow request $f$ at node $n_i$ with a demand of $|f|$:
**if** $destination_node(f) \in one_hop(n_i)$ **then**
$\alpha_i \leftarrow hop_count(f)$
**else**
$\alpha_i \leftarrow hop_count(f) + 1$
**end if**
$B_L \leftarrow local_available_bandwidth(n_i, \alpha)$
$B_N \leftarrow neighborhood_available_bandwidth(n_i, \alpha)$
$D_e \leftarrow delay_estimate(f)$
$D_r \leftarrow delay_requirment(f)$
**if** $B_L >= |f|$ **and** $B_N >= |f|$ **and** $D_N < D_f$ **then**
admit $f$.
**else**
deny $f$.
**end if**

Algorithm 6: Route Discovery Admission Control

Discovery Filter

The portion of a route contained by RREQ packet during a QASR discovery is called a partial-path. In the event that a node receives more than one RREQ in a short period of time for a unique flow, it is likely that the partial-paths of the various RREQ may differ.

When a RREQ packet arrives at a node, the node evaluates the routing metric for the partial-path of the request. This value is stored for the unique flow request in the Partial Path Admission Threshold (PPAT). To allow superior quality route discoveries to supersede previously discovered routes, the RREQ will be rebroadcast if the metric for the new partial-path exceeds the threshold set by the previous best discovery for that flow. For each improved RREQ, the PPAT is updated.

Each network node must preserve a PPAT for each discovery process. These metric values are stored in the Partial Path Admission Threshold Table (PPATT).

Values stored in the PPATT include a unique flow identifier, the best partial-path evaluation for that flow, and the time discovery broadcast for the flow. Entries in the PPATT are periodically removed as they age and the related discovery process can safely be assumed to be complete.

### Route Selection

When a particular RREQ reaches its destination node, it is unlikely to be unique for a flow. If the request is the first one received for a flow, the destination node opens a Route Accept Window, a time duration in which discovery packets for the flow will be collected. The duration of this window should be set to be equal to the delay requirement of the flow, guaranteeing that enough time to collect all routes with acceptable delay are collected before route selection occurs.

At the expiration of the acceptance window, the destination node selects the best route from the set of acquired routes and sends a RRPLY. For a short period of time following the expiration of the Route Accept Window, a Route Denial Window is used to catch later RREQ packets. At the expiration of the Route Denial Window, all collected RREQ packets are disposed of, and any other memory associated with the discovery process is cleared. This allows future RREQ packets to be accepted in the event of a route failure.

# SIMULATION

The simulation environment used to model the protocol behavior and construct scenario configurations was OPNET 12.1 [16], a commercial network communications simulator. OPNET allows for scenario modeling with mobility terrain considerations on the wireless communications process. A detailed description of the models used and the methods used to implement QASR is available in Appendix A.

A rural environment, in the context of this thesis, is one in which the location is too remote to expect traditional communications infrastructure to be consistently available. In the scenarios present here, no communications infrastructure is modeled. Additionally, many rural environments have irregular terrain that make future deployment of communication infrastructure with acceptable coverage economically infeasible. It is these environments in which MANETs are being investigated to provide wireless communication resources to emergency workers.

## Scenarios

In this section we describe the several scenarios in which we test the performance of QASR in comparison with DSR and AODV. First we present a generic MANET scenario which only models mobility. Each following scenario includes mobility and irregular terrain, simulating rural network deployments. The white lines in the figures represent the node trajectories during the simulation. The darker lines represent the elevation profile for the area.

Random Waypoint

The first scenario set we present models a mobile network without terrain. This scenario will provide a baseline reference for the remaining scenarios. The scenario set consists of mobile nodes who's movement is defined by OPNET's Random Waypoint Model. The scenario area is 4km by 4km, and node movement is constrained within this area.

The simulations study network performance while varying network node density and traffic load. To study density, scenarios were run with an increasing number of nodes in the scenario area. The first scenario consists of 15 randomly placed nodes. Each node in the simulation imposes a traffic demand to another node in the network. This set of demands is uniform among the 15 nodes, summing to a total of 100Mbps and remains consistent as nodes are added in successive scenarios. Each successive scenario adds 15 nodes which are also randomly placed. The last scenario in the density set contains 90 nodes.

To study the performance under load, the number of nodes is held constant at 45, and the traffic demand of the 15 flows is increased. The first scenario has a total demand of 20kbps. Successive scenarios add 20kbps of load by increasing the rate of all the flows uniformly.

Grand Canyon Junction

The Grand Canyon Junction scenario is a vehicular MANET situated around Grand Canyon Junction, Wyoming, in Yellowstone National Park (Figure 7). In this scenario the network nodes are modeled as vehicles traveling on roadways with several intersections. The vehicles turn at intersections in one of the available directions

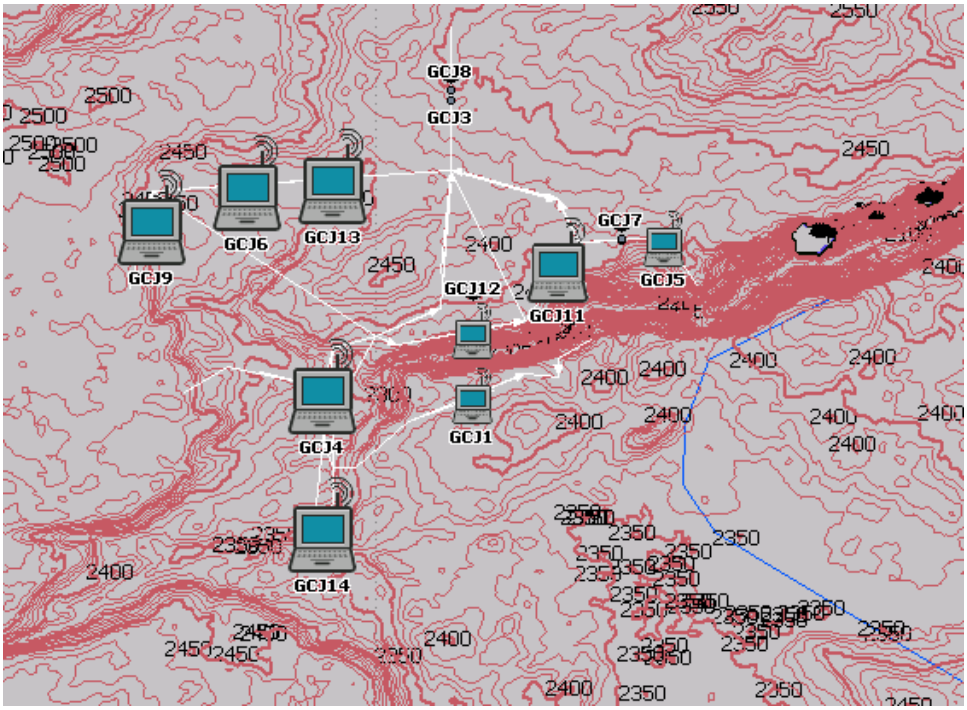


Figure 6: Vehicles Placed on Roadways Near Grand Canyon Junction

according to a probability function. The simulation area is approximately 4km by 4km, with about 34.6km of roadway, and 5 intersections.

For this simulation, we vary node density and traffic demand in the same way as the Random Waypoint scenario set. This scenario provides a rural traffic scenario in which we can vary the traffic density, in a location with irregular terrain. The radio propagation model is line-of-sight, such that if the terrain obstructs a straight line view between two nodes, they will be unable to communicate directly. While the scenario area is comparable to that of the Random Waypoint scenario set, node locations and movement are restricted to roadways, rather than being randomly distributed.

The additions of terrain and the roadway restriction have contradictory effects on the node density. The terrain blocks communication to some extent, reducing the size

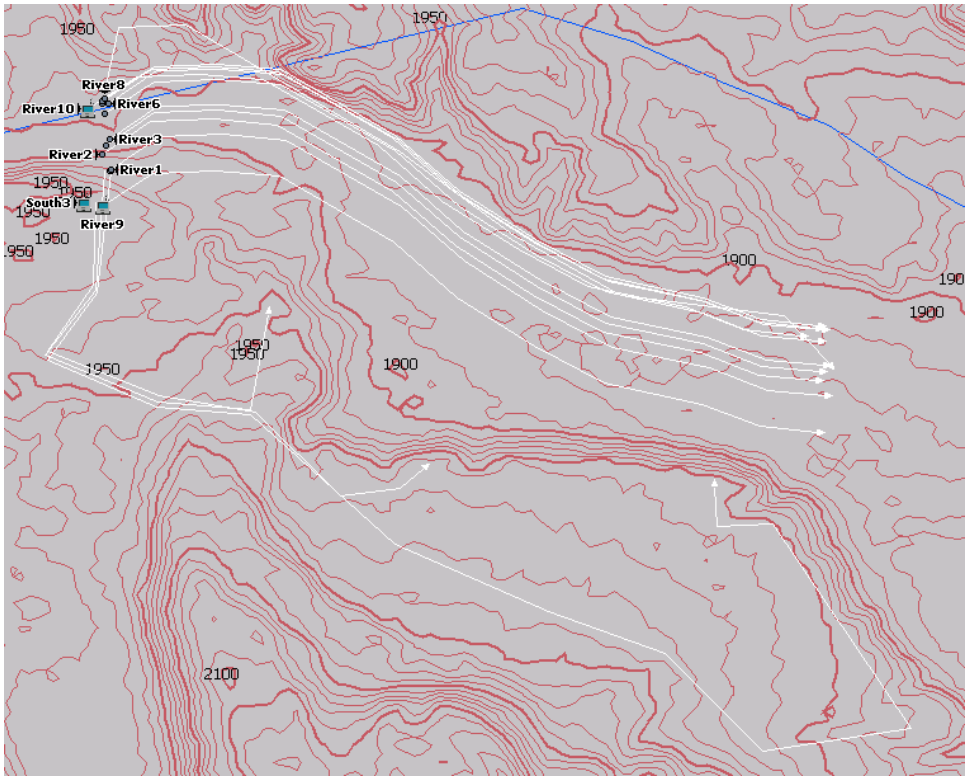


Figure 7: Rescue Workers Searching the Wind River Canyon

of a node's neighborhood. The roadway restriction has the opposite effect, as nodes are less spread out over the available area.

Wind River Canyon

The Wind River Canyon scenario represents a coordinated search along the Wind River Canyon (Figure 7) 30 km southeast of Dubois, Wyoming in a search operation for a missing person. The river valley is 1.6km across at its widest in the context of our simulation. Nodes in the network represent both rescue workers on foot walking along the bed of the river valley, as well vehicles that maneuver to overlook the search effort, in order to visually monitor the health and safety of the rescue workers themselves.

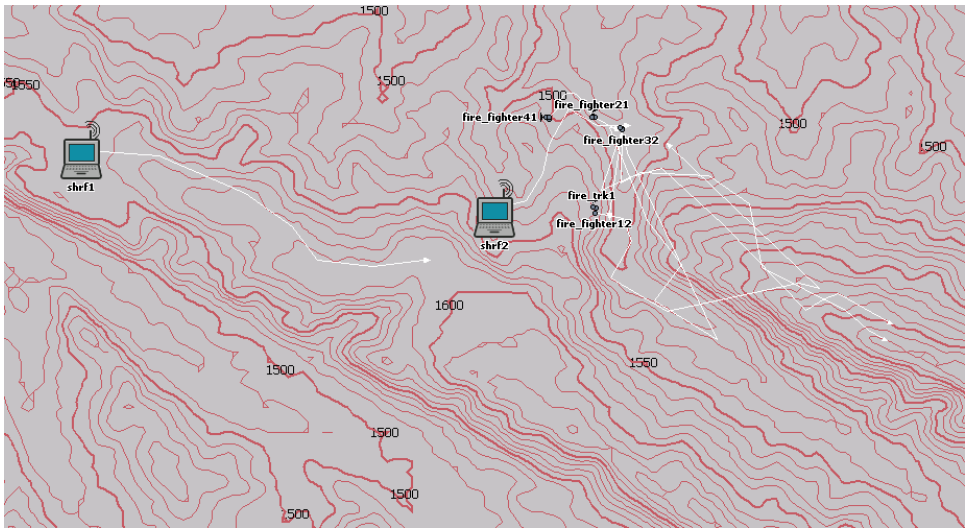


Figure 8: Emergency Workers Fighting a Wildfire on Black Mountain

The traffic in this scenario is scheduled by 17 nodes with 20 flows between various members of the search party. Several of the flows involve a particular vehicle node labeled *South1*. *South1* suffers from intermittent signal loss as it traverses irregular terrain to find an effective overlook point.

Black Mountain

The Black Mountain scenario represents an emergency fire response to a small wildfire near an oil pumping station approximately 35km east of Thermopolis, Wyoming (Figure 8). In this scenario, two county sheriff officers, four water cannon trucks and a water pumper truck are deployed to fight a grass fire that is moving up the western slope of Black Mountain toward the pumping installation.

The fire fighting vehicles deploy to the north and south side of the mountain to contain the blaze while the sheriffs provide overwatch from safe locations. Direct communication between fire fighters is blocked by the mountain itself, so that the officers must provide communications assistance.

| | | |
|---|---|---|
| Channel Bandwidth | (QASR) | 750 $kbps$ |
| Delay Limit | (QASR) | 50 $mSec$ |
| Speed Limit | (QASR) | 27 $mph$ |
| PORC Constant | (QASR) | 3 |
| $K_B$ | (QASR) | $\frac{1}{3}$ |
| $K_D$ | (QASR) | $\frac{1}{3}$ |
| $K_S$ | (QASR) | $\frac{1}{3}$ |
| Intermediate Node Buffer Size | (DSR/QASR/AODV) | 10 $packets$ |
| Allowed Hello Loss | (AODV) | 1 |
| Max Maintenance Retransmit | (DSR/QASR) | 2 |
| Maintenance Holdoff Time | (DSR/QASR) | 1.0 $sec$ |
| Delay QoS Tolerance | (DSR/QASR/AODV) | $50ms$ |
| Jitter QoS Tolerance | (DSR/QASR/AODV) | $50ms$ |
| Radio Transmit Power | (DSR/QASR/AODV) | $5mW$ |
| Radio Interference Range | (DSR/QASR/AODV) | $1500M$ |
| Channel Bitrate | (DSR/QASR/AODV) | $1Mbps$ |
| Radio Frequency | (DSR/QASR/AODV) | $2.4GHz$ |
| Communication Range | (DSR/QASR/AODV) | $1km$ |
| Interference Range | (DSR/QASR/AODV) | $1.5km$ |

Table 3: Simulation Execution Parameters

Code Configuration

Table 3 presents the values for configurable parameters for the evaluated protocols. Parameters for AODV and DSR are applied through the OPNET user interface. QASR parameters are set directly in the source code. A more thourough description of the QASR implementation is available in Appendix A.

The link layer used is the 802.11e MAC. The traffic demand is balanced over the top 3 802.11e priorities, known collectively as the Realtime priorities. The lowest priority, known as the Background priority, is not used. Use of the Background priority for non-realtime traffic support is discussed in the Future Work section of Chapter 6. The RTS/CTS option is enable for all traffic.

The delay and jitter QoS tolerance values come from the Safecom Statement of

Requirments [3]. These parameters were given as QoS requirements for critical real-time traffic in emergency scenarios.

## Results

In this section we present the various statistics collected in evaluating QASR against the performance of AODV and DSR in the scenarios described in the network topologies section. Throughput is the sum of all data successfully delivered from each flow's source node to its destination node. Overhead is the sum of all other traffic sent by nodes for protocol specific operations such as route discovery, route maintenance, and information exchange. Delay is the average end-to-end delay of received data packets in all flows. Jitter is the average of the difference in the end-to-end delay of successive packets in a particular flow. Packet Delivery Ratio is the fraction of number of data packets that are successfully delivered over the number of data packets presented to the network for routing. Quality of Service Acceptance Ratio is the number of data packets that arrived within the Delay and Jitter QoS tolerance limits over the number of data packets successfully delivered.

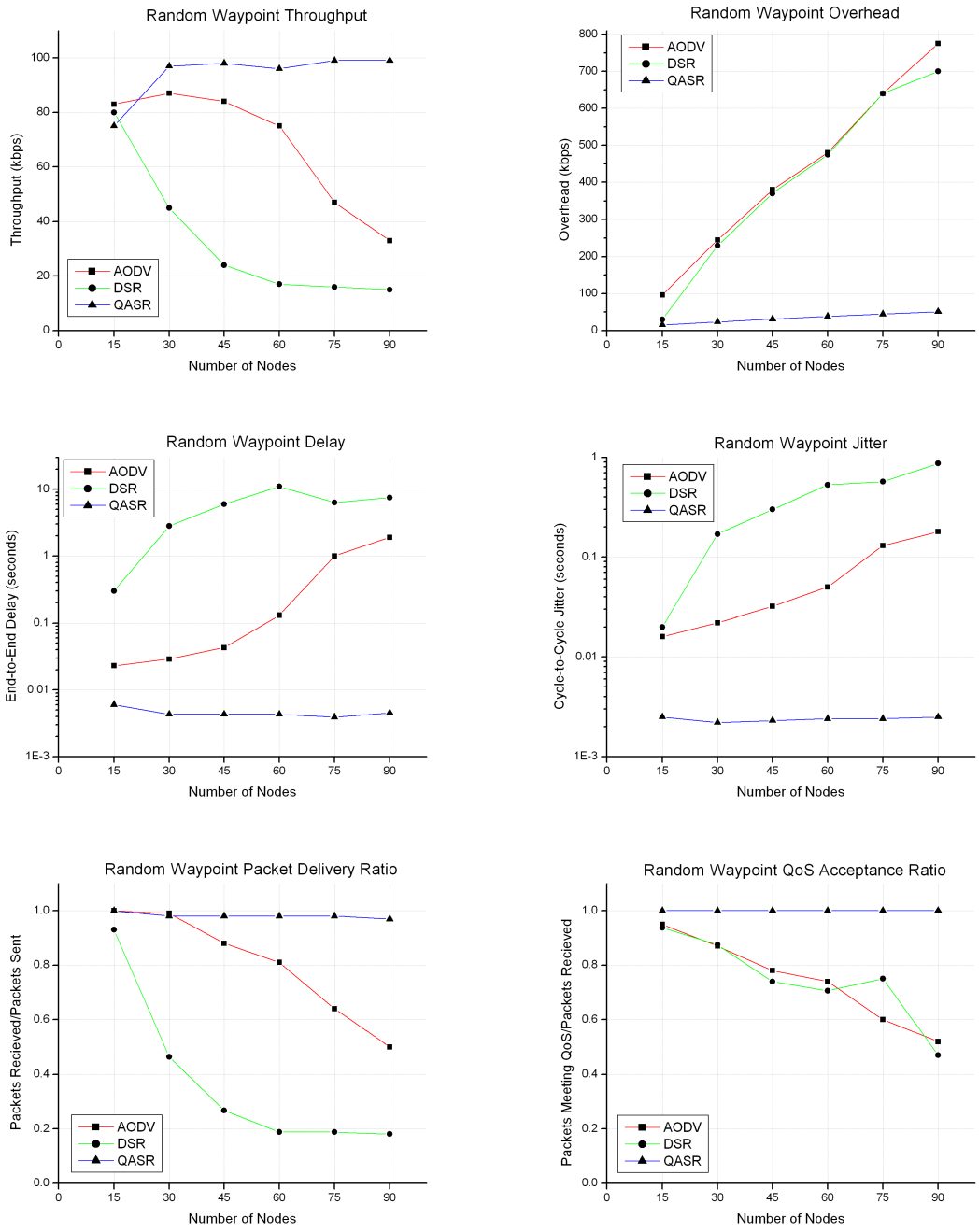


Figure 9: Random Waypoint MANET Node Density Performance Statistics

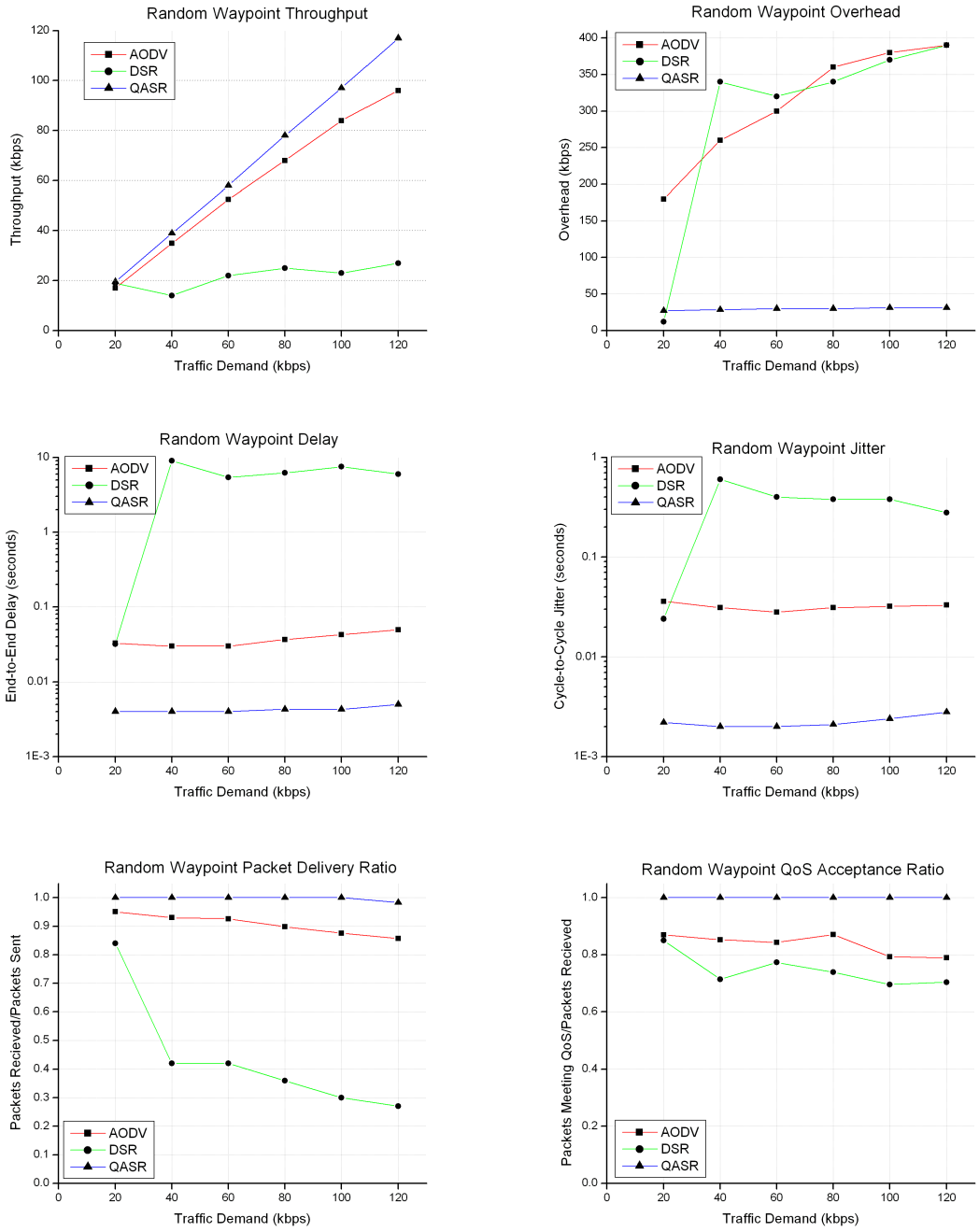


Figure 10: Random Waypoint MANET Traffic Demand Performance Statistics

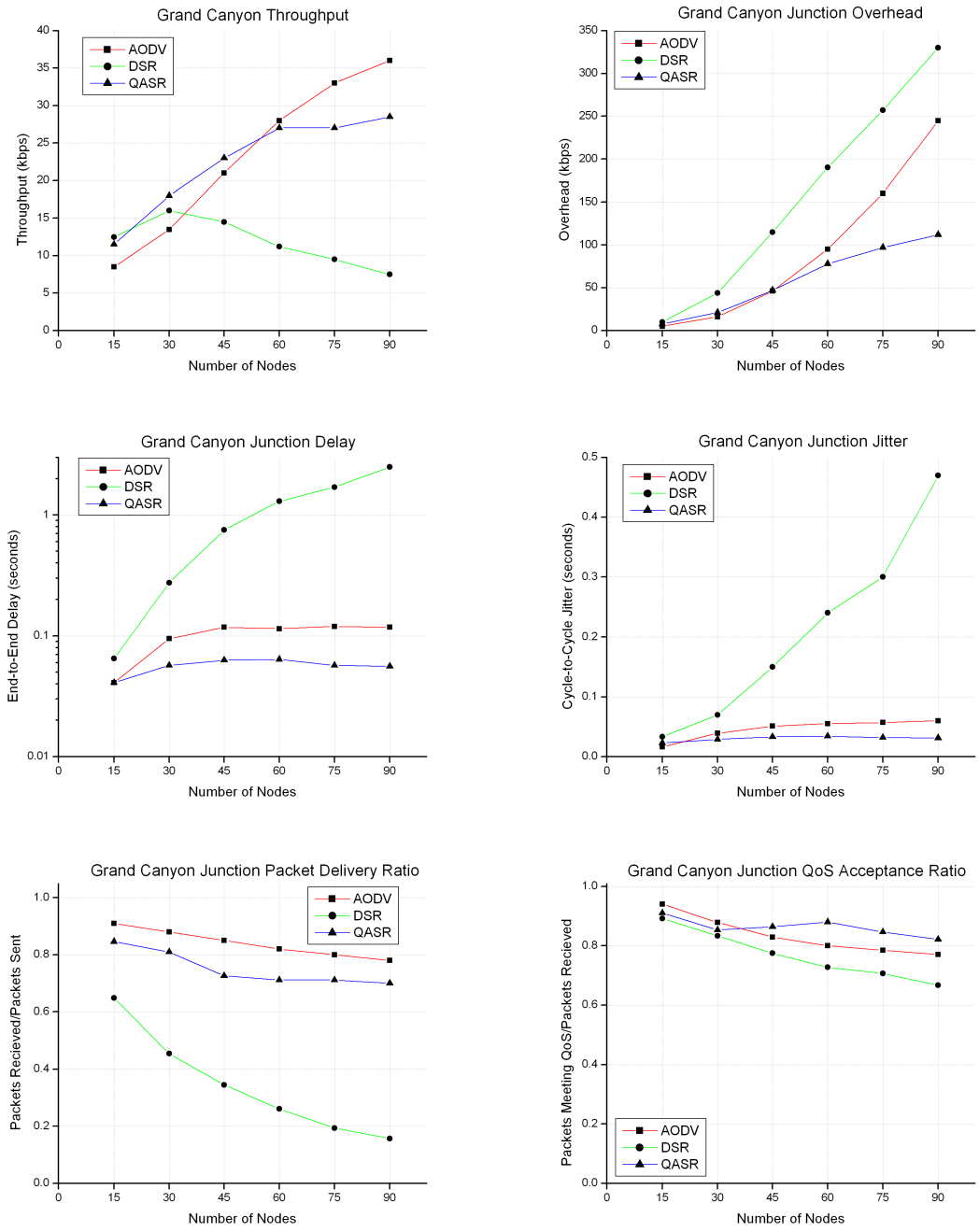


Figure 11: Grand Canyon Junction MANET Node Density Performance Statistics

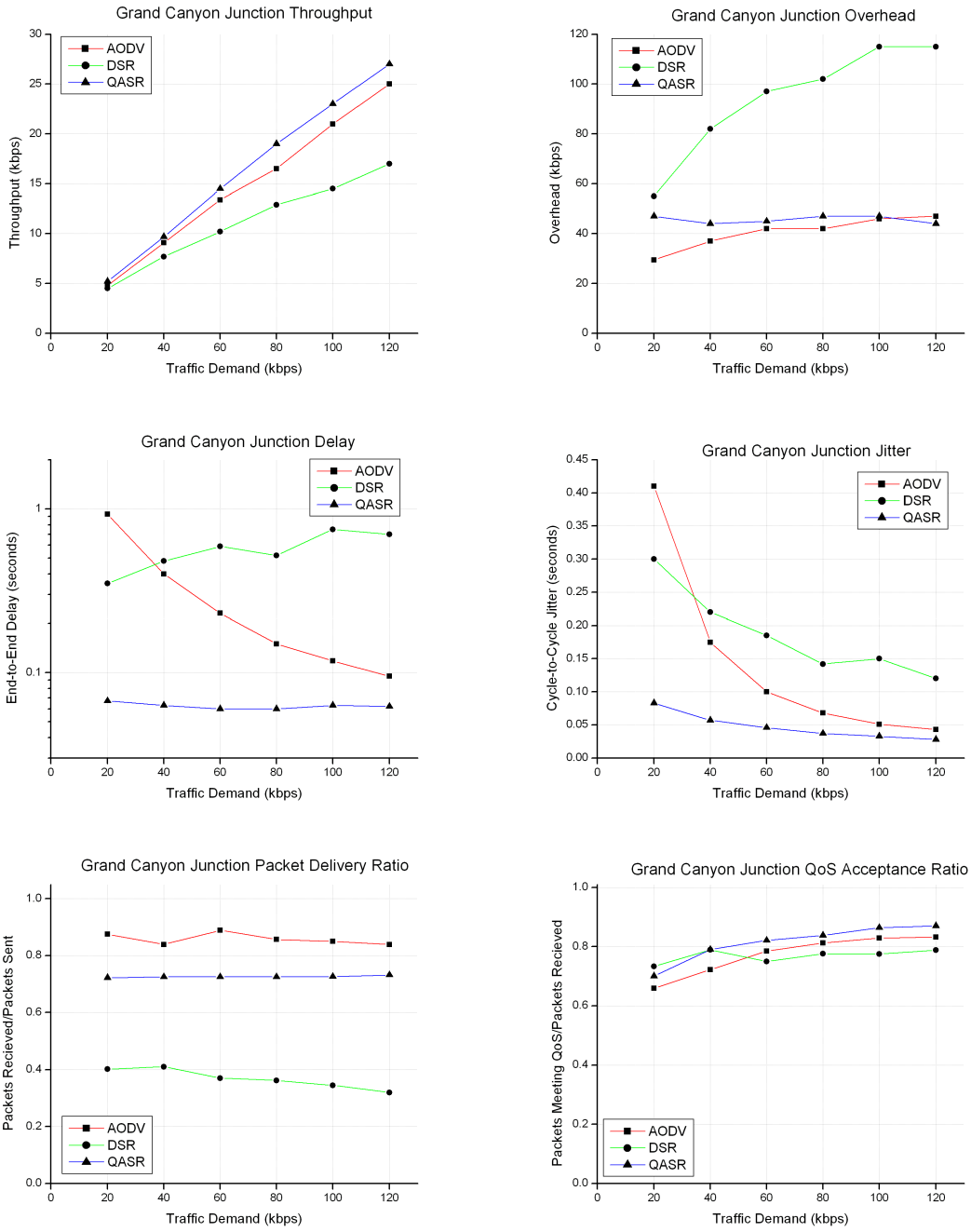


Figure 12: Grand Canyon Junction MANET Traffic Demand Performance Statistics

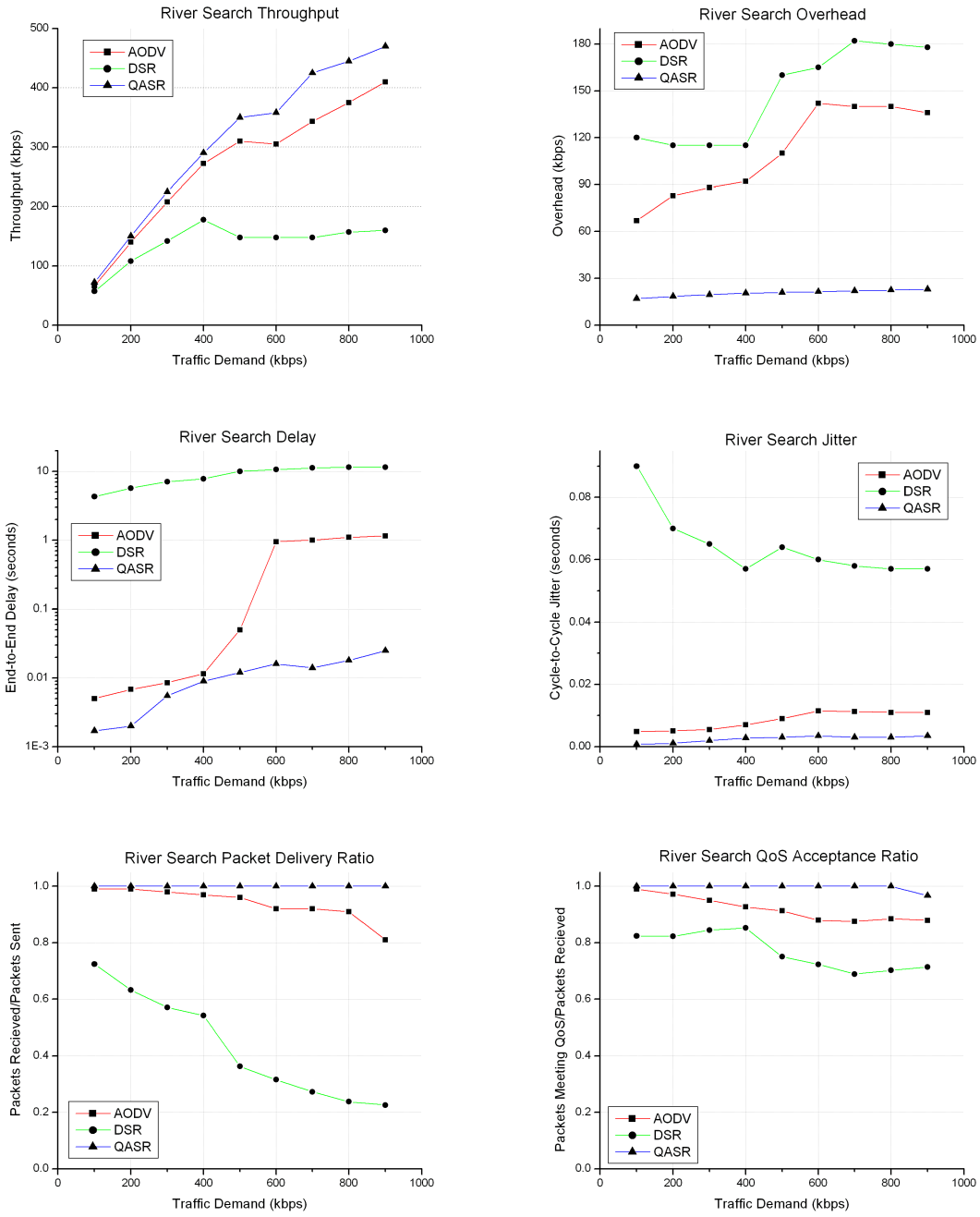


Figure 13: River Search MANET Traffic Demand Performance Statistics

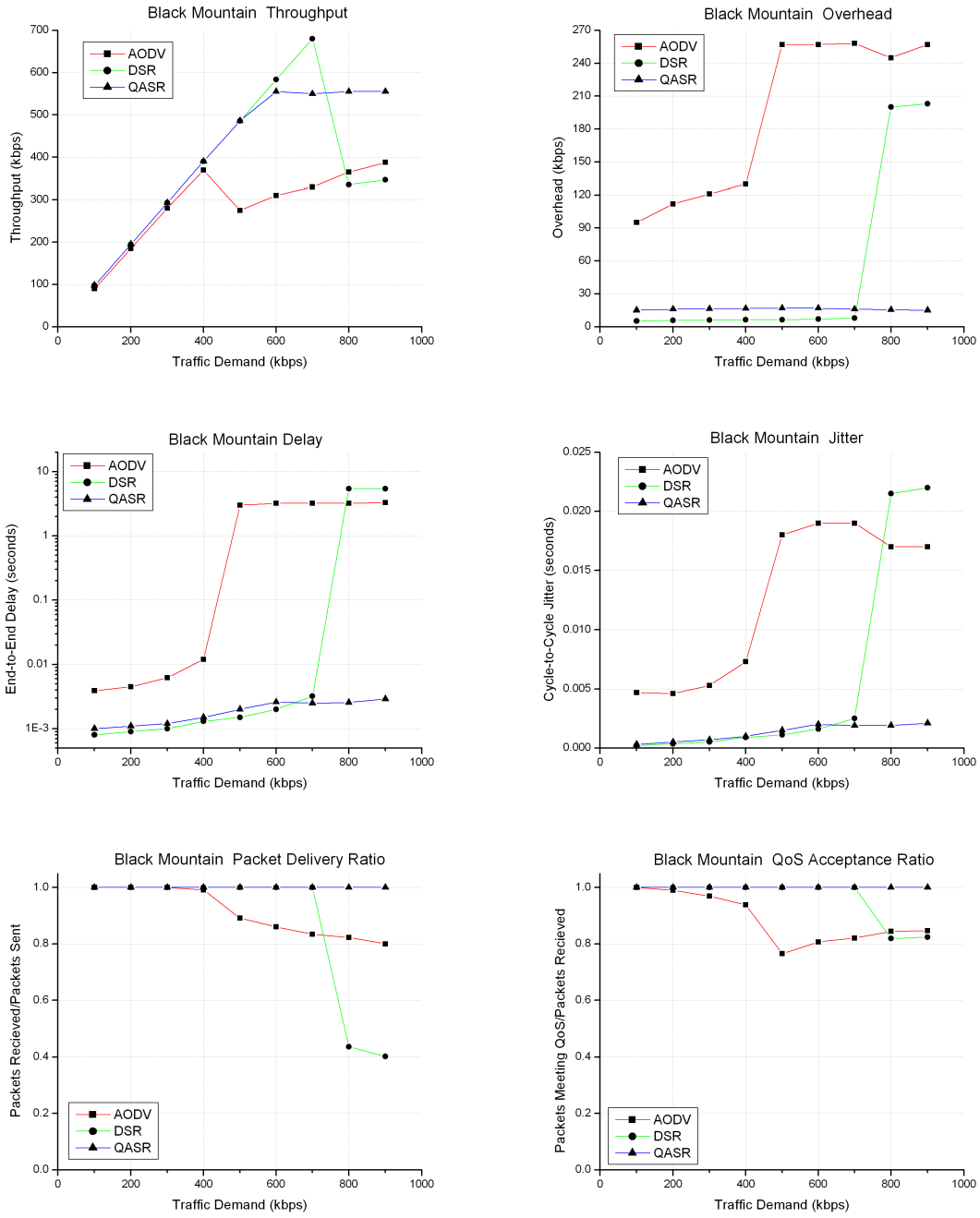


Figure 14: Black Mountain MANET Traffic Demand Performance Statistics

# ANALYSIS

In this chapter we provide tables containing data aggregated from the figures in Chapter 4, and provide in-depth analysis of the evaluated protocols.

## Throughput

On average, in the scenarios used in this thesis to evaluate these protocols, QASR delivered 26% more data than AODV, and 55% more data than DSR. End-to-end throughput increases with demand as traffic demands are met by the protocols, and with density as additional nodes provide connectivity between otherwise partitioned sections of the network.

In each scenario, QASR throughput is competitive with DSR and AODV. In the Grand Canyon Junction scenario AODV is seen to route significantly more traffic than QASR at higher node densities. In the Black Mountain scenario it can be seen that DSR throughput also exceeds that of QASR. These performance gaps represent the pessimistic route admission in QASR. This reflects the saturation avoidance behavior of QASR, and it can be seen that further increases in demand cause DSR performance to degrade dramatically as particular nodes suffer network congestion and begin to saturate. Increasing the bandwidth limit parameter for QASR could potentially increase the average throughput capacity of a QASR network, but with an increased probability of node saturation. When the bandwidth limit is set too high, the performance of QASR is consistently worse than that of DSR, because the additional protocol overhead of QASR leads to saturation at lower traffic demands.

The Random Waypoint (RW) and Grand Canyon Junction (GCJ) scenarios show interesting behavior relating to the differences in node placement and movement

restriction, as well as the effect of terrain. As was described in Chapter 4, these two scenario sets share many parameters, such as total area, number of nodes, and traffic demands. The *total node density* is therefore roughly the same, but nodes are constrained to roadways in the GCJ scenario, whereas nodes in RW are initially placed more uniformly in the scenario area and then follow a random movement pattern. The result is the *local node density* is higher for the GCJ scenario than for RW, as there is significantly less area in which nodes are allowed in GCJ.

The second difference is the addition of terrain. The irregular terrain in the GCJ scenario leads to significantly fewer potential routes to choose from, worsened by the restrictions on node location. In the RW scenario, there are many potential routes to choose from, and the abundance of route options provides flexibility for QASR to avoid congestion. This is evidenced in the GCJ and RW density scenario, where QASR throughput is consistently high, while AODV and DSR performance degrades with increased density. This difference can be attributed to QASR's more sophisticated routing metric and route selection protocol.

QASR's routing advantage is removed in the GCJ scenario, however. In GCJ, there are much fewer options, and performance is more a function of how efficiently the protocols use these routes. Due to the increased node density, and by extension the overhead for QASR, QASR performance suffers slightly as the higher overhead consumes bandwidth, and QASR is less competitive against AODV in terms of throughput.

### Overhead

The protocol overhead of QASR is about 26% that of DSR, and about 17% that of AODV. The overhead for QASR is effectively constant, or linear with regard to node

density, in all scenarios. Protocol overhead for AODV and DSR varies as a function of density, demand, and also dramatically when the network begins to saturate. QASR limits the addition of unsustainable traffic to the network, as well as discourages link over-use via its routing metric, thus avoiding a dramatic change in overhead volume.

There are several issues that may be causing increased overhead for AODV and DSR. First, all three MANET protocols detect link-breakage by counting failures of HELLO or ACKREQ packets. This will detect link failure, but will also treat packet loss due to congestion as link failure. Due to the simple hop-count routing metric, AODV and DSR are likely to suffer link failure due to congestion, but then find the same or a similar route after conducting a route discovery. If the rediscovered route failed previously due to congestion, it is likely to do so again. This repetitive failure leads to high overhead, as all the nodes in the network add to the network load during discovery, with no net gain in network efficiency. This leads to a pattern of failures and discoveries that consumes network resources.

Also, AODV and DSR react to a RREQ arrival immediately, either by forwarding the RREQ, or responding with a RRPLY. The only delays imposed is the small jitter used to reduce the probably of colliding with neighbors that may be reacting to the same packet. QASR, on the other hand, imposes delays and limits on discovery frequency using Route Accept Windows, and Route Reject Windows. These windows were necessary into order to collect multiple RREQ packets, so that they can be sorted according to the QASR routing metric, and a single route selected. A side effect is that a QASR destination node is very unlikely to respond more than once during a particular route discovery process.

## Delay

Average end-to-end packet delay was $23.4ms$ for QASR, $511ms$ for AODV, and $3.84ms$ for DSR. It is unfair to compare the average delay results, as it would imply that QASR shows a delay performance over a hundered times better than DSR. This is not the case. A better view of delay performance is in terms of meeting Quality of Service requirements and can be seen in the QoS Acceptance Ratio, which is examined in Section **??**.

The average delay metric proved less useful than would be hoped for in the effort of evaluating protocols to meet QoS requirements. It can most clearly be seen the Random Waypoint and River search scenario, that DSR shows an unreasonably high average delay. In the River Search and Black Mountain scenario, AODV can be seen to transition dramatically at when demand is about $500kbps$. This sharp increase in delay corresponds to an increase in protocol overhead and a decrease in end-to-end throughput.

The variance of the delay statistic is not easily collected in OPNET, but we hypothesize that the variance of the average delay in these cases is very high. The QoS Acceptance Ratio measures the number of packets with end-to-end delay and cycle-to-cycle jitter less than $50ms$, and those that are greater. Given that we see reasonable ratios, we have to conclude that the average delay is dominated by a few packets that wait in queues for much greater than the average delay.

## Jitter

The average jitter measure for QASR is 28% that of AODV, and 7% that of DSR. The average cycle-to-cycle jitter includes the delays caused by link breakages, which

cause some packets to have very long delays at the source node. An increase in average delay generally corresponds to an increase in average jitter, because network congestion generally increases both measures. In scenarios where route discovery takes longer to succeed, such as the Grand Canyon Junction scenario, the jitter decreases with an increase in traffic demand. This distortion is the effect of large delays between packets due to frequent link breakage increasing the average of many packets with a lower jitter measure. The number of link breakages is increasing very gradually, and so the increase in number of packets per second leads to a decrease in the average jitter. If link breakages increased more dramatically with demand, average jitter would be seen to increase with demand.

Although all three protocols show distorted jitter measures in the Grand Canyon Junction scenario, the QASR measures are significantly less distorted than AODV and DSR, in spite of the additional delays imposed on the route discovery process by the QASR protocol. This implies that the number of route failures is less than that of AODV and DSR by such a large margin that the imposed delays have a nearly negligible effect on the average jitter measured.

### Packet Delivery Ratio

Of all data packets that were introduced to the network for routing, 3% more QASR packets were successfully delivered than AODV packets, and 43% more than DSR. AODV delivered, on average, nearly 40% more packets than DSR. This difference between AODV and DSR can be attributed to the failure modes while link failure is being detected by these protocols. As was discussed in Sections  and , DSR will begin queueing packets after the first ACKREQ failure. AODV will continue to transmit packets until enough HELLO packets fail. Because of this difference, if

there are many route failures due to congestion, DSR routes will fail with packets in queue that are guaranteed not to be delivered. When the route fails, these queued packets are discarded. AODV is less likely to have packets in queue when the route fails, and therefore more packets transmissions are attempted, and enough packets are successfully delivered that AODV shows a higher Delivery Ratio.

As discussed in Section , the link breakage rate for Grand Canyon Junction increases gradually, leading to a gradual decline in Delivery Ratio for DSR. In the River Search scenario, the intermittent connectivity with the node labeled *South1* singularly causes many routes to be discovered and broken very quickly, leading to many routes failures and high packet loss. With the higher bit rates used in the Black Mountain scenario congestion creates a stronger correlation between traffic demand and link breakage, and so the decrease in the DSR Packet Delivery ratio as demand increases is more pronounced. In the Black Mountain scenario, mobility is lower than the other scenario, and there are fewer hops required to route traffic. In this scenario, link breakage is entirely a dependant on traffic demand, and a sudden drop in the Delivery ratio can be seen when the network begins to saturate.

Routes in QASR are much less likely to break due to congestion, and so it shows very high delivery ratios. In the Grand Canyon Junction scenario the QASR delivery ratios are fairly consistent through both changes in density and demand, but also consistently less than that of AODV and higher than DSR. The underlying protocol on which QASR is based is DSR, and so the failure mode when link breakages are detected is to buffer packets. QASR is unlikely to suffer link breakage due to congestion, and so the margin between AODV and QASR likely involves irregular terrain causing frequent link interruptions, thus causing frequent flushing the packet queue. If this is the case, than the cross-layer optimization for detecting link integrity discussed in the Future Work section of Chapter 6 would likely close the performance

difference between AODV and QASR for this metric.

### QoS Acceptance Ratio

Of all data packets that arrive at their destination, 11% more QASR packets arrive within the delay and jitter requirement than AODV packets, and 15% more than DSR, averaged over all scenarios. This is a more clear view of QoS satisfaction than the average delay metric, and demonstrates QASR as a more effective protocol for satisfying QoS requirements.

The differing methods of fault-tolerance lead to differing conditions in which the protocols become saturated and effectively fail. The average end-to-end delay can be seen more as a measure of whether a network is saturated or not, as the time spent by a few packets waiting in queues dominates the average delay metric. Even for DSR, which shows an unreasonably high average delay, a significant portion of packets can be seen arriving within the QoS requirement boundaries. It is the primary function of QASR to prevent saturation, and thus it demonstrates a near-linear relationship between network load and average packet delay without dramatic changes in behavior.

# CONCLUSION

It was the goal of this thesis to develop QASR, a MANET protocol which implements bandwidth and delay estimation techniques to provide Quality of Service in rural environments. We showed that AODV and DSR, which use only hop count as a routing metric rather than using bandwidth and delay models and do not perform admission control during route discovery, show dramatic performance degradation due to their allowance of node saturation. QASR shows generally improved performance over both AODV and DSR in both throughput and delay because it uses models to prevent saturation.

We have shown through several simulations that directly address the networking challenges of MANET scenarios that QASR performs more consistently than the competing protocols. We have also shown that the protocol overhead cost of the QASR information exchange does not significantly impact the network, and to a large extent preserves the on-demand nature of the underlying DSR protocol. Clearly QASR preserves the robustness of its DSR heritage and improves service quality without excessive protocol overhead.

We conclude that QASR demonstrates the utility of the Yang's bandwidth and delay estimation models, and shows that they can be incorporated into a working MANET protocol. With these features, MANET technology could feasibly be deployed in rural scenarios, potentially providing emergency workers with network access in the absence of the infrastructure found in urban environments.

# Future Work

## Location Awareness

The QASR protocol currently depends on location information to determine the distance between nodes in the network. To gather location information, a network node would likely require GPS technology. This information is exclusively used in the calculation of interference ranges. It may be worth exploring the use of methods which do not require location awareness. One such method is to model an interference range as the two-hop neighborhood. This method is likely to be less accurate than using GPS, but the impact of the loss in accuracy may be acceptable.

## Terrain Awareness

Currently the QASR protocol is implemented with that assumption that GPS data is collected. GPS data provides absolute location information which, when combined with accurate elevation maps, can provide a more accurate interference model than the one currently used. The interference model is currently a straight line threshold model, meaning that if two nodes are within a certain straight line distance of each other then they are within each other's neighborhoods. In areas of irregular terrain, this may result in two nodes modeling each other as neighbors even if they do not in fact interfere. By incorporating the GPS data that is collected with elevation maps, more accurate link availability estimates can be obtained, leading to even greater throughput may be available in some network topologies.

## Parameter Tuning

The tunable parameters $K_B$, $K_D$, and $K_S$ did not demonstrate dramatic differences in protocol performance, as expected. In particular, the Speed parameter,

$K_S$ was expected to be more effective in selecting more stable routes. It may be that using speed is significantly less effective than using velocity. In particular, if an intermediate node had access to the velocity of its neighbors, then relative velocities could be computed, providing a bias towards routes along groups of nodes traveling in the same direction. The current approach, using speed, does not do this.

The Bandwidth tuning parameter, $K_B$, may also be more or less important in networks of various size. It may be that $K_B$ would serve better if implemented as a function of total network size. Not enough simulations were run to determine the importance of this parameter at differing network sizes, and it may be a worthwhile direction to explore.

Priority Discrimination

As discussed in Section , the flow state data at each node is separated into 4 categories according to the 802.11e priority scheme. A network QoS feature that should be considered is for higher priority routes to supersede lower priority routes. Such a feature would cause a RERR to be generated for lower priority routes in the event that a higher priority flow required access to otherwise unavailable network resources. This would cause the lower priority flow to be dropped from the network, allowing the higher priority traffic to be admitted without overbooking the network.

A notable obstacle to the implementation of this Priority Discrimination feature is the selection of which lower priority flow to drop. A higher priority flow may cause multiple lower priority flows to fail simultaneously, thus leading to multiple route discoveries being initiated simultaneously. The route discovery process begins with a flood search, which, when initiated by many nodes at once could temporarily, but significantly, degrade network performance.

### Background Priority Traffic

The development of QASR and the evaluation of the protocols in this thesis have focused entirely on supporting realtime traffic demands with strict QoS requirements. QASR treats the top 3 of the 4 802.11e priority classes as realtime. The 4th priority, named the Background priority, is intended to support non-realtime traffic, such as web access, file transfers, and other application data traffic using the TCP protocol.

Background traffic should be managed differently from realtime traffic, so that as much background traffic is routed as possible without critically effecting realtime flows. Background traffic does not have a known bandwidth requirement, and so admission control during route discovery cannot be used. Instead, all background traffic route requests should be allowed to succeed, and the estimate of available bandwidth should be used to dynamically determine the network resources given to background traffic at each intermediate node. As new realtime flows are admitted the available bandwidth would decrease, resulting in less background traffic being routed.

### Cross-Layer Optimization

The protocols evaluated in this thesis depend on their own means to determine link connectivity. This is an inefficient use of network resources. Each protocol sends a periodic message, either Acknowledgement Requests for DSR and QASR, or HELLO packets for AODV. These packets increase protocol overhead and their transmission period defines the temporal sensitivity to link breakages. These protocols would function better if instead they interfaced with the 802.11 MAC which employs per-packet acknowledgements. Such a modification would provide maximal temporal resolution for link sensitivity and reduce the periodic link-query traffic to zero.

# REFERENCES


[1] Yaling Yang and Robin Kravets. Achieving delay guarantees in ad hoc networks by adapting ieee 802.11 contention windows. *IEEE Transactions on Mobile Computing*, 2008.

[2] Yaling Yang and Robin Kravets. Throughput guarantees for multi-priority traffic in ad hoc networks. *Elsevier Journal of Ad Hoc Networks*, 2008.

[3] The SAFECOM Program Department of Homeland Security. Statement of requirments for public safety wireless communications and interoperability, January 2006.

[4] Injong Rhee, Ajit Warrier, Jeongki Min, and Lisong Xu. Drand: Distributed randomized tdma scheduling for wireless ad-hoc networks. *MobiHoc*, 2006.

[5] X. Wu., B.S. Sharif, O.R. Hinton, and C.C. Tsimenidis. Solving optimum tdma broadcast scheduling in mobile ad hoc networks: a competent permutation genetic algorithm approach. *Communications, IEE Proceedings*, Dec 2005.

[6] Yaling Yang and Robin Kravets. Contention-aware admission control for ad hoc networks. *IEEE Transactions on Mobile Computing*, 4:363–377, Aug 2005.

[7] Ian F. Akyildiz, Xudong Wang, and Weilin Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445–487, March 2005.

[8] Hannan Xiao, Winston Seah, Anthony Lo, , and Kee Chaing Chua. A flexible quality of service model for mobile ad-hoc netoworks. *IEEE Semiannual Vehicular Technology Conference*, 2000.

[9] Matthew Andres, Krishnan Kumaran, Kavita Ramanan, Alexander Stolyar, and Phil Whiting. Providing quality of service over a shared wireless link. *IEE Communications Magazine*, Febuary 2001.

[10] M.S. Corson. Issues in supporting quality of service in mobile ad hoc networks. *IEEE International Conference on Communication*, pages 1089–1094, May 1997.

[11] M. Gerharz, C. de Waal, M. Frank, and P. James. A practical view on quality-of-service support in wireless ad hoc networks, 2003.

[12] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An architecture for differentiated services, December 1998.

[13] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 headers, December 1998.

[14] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated services in the Internet architecture: an overview, June 1994.

[15] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin. RFC 2205: Resource ReSerVation Protocol (RSVP) — version 1 functional specification, September 1997.

[16] OPNET. http://www.opnet.com. 2007.

# APPENDICES

# APPENDIX A

## QASR IMPLEMENTATION DETAILS

QASR was implemented in OPNET 12.1 as a modification to existing code. In particular, code modifications to implement the QASR protocol are completely contained in the OPNET process module dsr_rte.pr.m. This process module is a child process of the IP module, ip_rte.pr.m. As a child process, dsr_rte is dependant on the ip_rte module. The DSR protocol requires that packets be modified in order to traverse the network, and so all packets that are sent and received at a DSR node must be processed by the dsr_rte.

To contrast, in the case of the AODV protocol, the IP module sends packets to the AODV code only when a route to the destination is not found in the IP routing table. The AODV code then uses the destination address in the packets it receives to conduct discovery operations. When the discovery operations succeed, AODV updates the IP tables directly, and packets can be routed directly by the IP module, bypassing AODV.

It is necessary to be clear, then, to establish that all packets that are sent and received by a DSR node pass through the DSR code. Packets arrive in the dsr_rte module by the parent module calling dsr_rte_pk_arrival(). Within this function it is established whether the packet is arriving or being sent and then appropriate actions are performed for either direction.

This section contains two sections: a listing of functions which were added, followed by a listing of the dsr_rte functions which were modified. The dsr_rte module is compiled with the ENABLE_DSR_EXTENSIONS flag enabled to provide the QASR functionality, and disabled to maintain DSR functionality.

## A.1 Code Contribution

This section includes the most relevant portions of the source added to the dsr_rte module to implement QASR. We present the important functions that were introduced as new functions to the dsr_rte module, and omit trivial functions with intuitively obvious behavior.

```
001 #define ENABLE_DSR_EXTENSIONS
002
003 #define DSR_EXT_METRIC_SPEED      0.33    //normalized to MAX_SPEED mps
004 #define DSR_EXT_METRIC_BANDWIDTH  0.33  //normalized to CHANNEL_BANDWIDTH bps
005 #define DSR_EXT_METRIC_DELAY      0.33  //normalized to 20ms
006
007 #define ENABLE_DSR_YANG_BANDWIDTH
008 #define DSR_EXT_FLOW_STATE_BCAST_INTERVAL 2.0 //in seconds
009 #define DSR_EXT_FLOW_RESERVATION_CLEANUP 6.11 //in seconds, flow state cleanup interval
010 #define CHANNEL_BANDWIDTH 800000 //bps
011 #define MAX_SPEED 27 //in meters per second
012 #define DSR_EXT_INTERFERENCE_RANGE 1500.0
013 #define DSR_EXT_OVERHEAD_CONST 3.0
014
015 #define ENABLE_DSR_YANG_DELAY
016 #define DSR_EXT_Td 1.6203e-3 //length of time required to send a packet
017 #define DSR_EXT_DIFS 10e-6
018 #define DSR_EXT_C1 1.933e-3
019 #define DSR_EXT_C2 1.0205
020 #define DSR_EXT_C3 1e-5
021
022 #define ENABLE_DSR_QOS_EXTENSIONS    //define for QoS specific modifications
023 #define DSR_EXT_ACCEPT_WINDOW 0.04     //destination RREQ accept window, in seconds
024 #define DSR_EXT_DELAYED_ACCEPT_DELETE 0.4 //RREQ ignore window following a RREQ
025 #define DSR_EXT_PPATT_CLEANUP 0.2    //hidden node PPAT entry lifetime, in seconds
026
027 /* Initializes data structures used by the DSR Extensions */
028 static void dsr_extensions_init()
029 {
030     int i;
031     float jitter = 0.0;
032     /* initialize the data strucutres used in the DSR Extensions */
033     FIN(dsr_extensions_init())
034     neighbor_list = op_prg_list_create ();
035     connection_list = op_prg_list_create ();
036
037     PPATT = op_prg_list_create(); /* Partial Path Acceptance Threshold Table, filters RREQ to strictly allow increasing quality */
038     op_intrpt_schedule_call (jitter, 0, dsr_ext_cleanup_PPATT, 0); /* starts the PPATT cleanup cycle */
039 #ifdef ENABLE_DSR_YANG_BANDWIDTH
040     //create flow reservation tables
041     for(i=0;i<3;i++)
042         flow_reservations[i] = op_prg_list_create (); //local flow state reservations
043
044     jitter = op_dist_uniform (DSR_EXT_FLOW_STATE_BCAST_INTERVAL );
045     op_intrpt_schedule_call (jitter, 0, dsr_ext_broadcast_flow_state, 0); // send flow history
046     op_intrpt_schedule_call (0.0, 0, dsr_ext_cleanup_flow_reservations, 0); // cleanup flow data
047 #endif
048 #ifdef ENABLE_DSR_YANG_DELAY
049     if(global_delay_estimate_init == OPC_FALSE)
050     {
051         for(i=0; i<GLOBAL_DELAY_ESTIMATE_SIZE; i++)
052             global_delay_estimate[i] = 0.0;
053         global_delay_estimate_init = OPC_TRUE;
054     }
055 #endif
056     FOUT;
057 }
058
059 static int dsr_ext_get_cw_from_priority(priority)
060 {
061     if(priority == DSR_PRIORITY_CRITICAL)//set priority based constants
062         return 2;
063     else if(priority == DSR_PRIORITY_REALTIME)
```

```
064         return 3;
065     else if(priority == DSR_PRIORITY_IMPORTANT)
066         return 5;
067     else if(priority == DSR_PRIORITY_BESTEFFORT)
068         return 7;
069     else
070         dsr_rte_error ("dsr_ext_get_cw_from_priority definition failure", "", "");
071     return 0;
072 }
073
074 static double dsr_ext_get_pk_size_from_priority(priority)
075 {
076     if(priority == DSR_PRIORITY_CRITICAL)//set priority based constants
077         return 1300;
078     else if(priority == DSR_PRIORITY_REALTIME)
079         return 1300;
080     else if(priority == DSR_PRIORITY_IMPORTANT)
081         return 1300;
082     else if(priority == DSR_PRIORITY_BESTEFFORT)
083         return 1300;
084     else dsr_rte_error("dsr_ext_get_pk_from_priority","invalid priority","");
085     return 0;
086 }
087
088 // cleanup function that removes old entries
089 static void dsr_ext_cleanup_PPATT(void* ptr_flags, int code)
090 {
091     PPATT_Element               *elem_ptr;
092     PrgT_List_Cell                *list_itr, *temp_list_itr;
093     int                           i, num_elems;
094
095     FIN(dsr_ext_cleanup_PPATT(void* ptr_flags, int code));
096
097     num_elems = prg_list_size(PPATT);
098     list_itr = prg_list_head_cell_get(PPATT);
099     for( i=0; i< num_elems; i++)
100     {
101         elem_ptr = (PPATT_Element*)prg_list_cell_data_get(list_itr);
102         if( (op_sim_time() - elem_ptr->timestamp) > DSR_EXT_PPATT_CLEANUP)
103         {
104             //clean up this route metric element
105             op_prg_mem_free(elem_ptr);
106             temp_list_itr = list_itr;
107         }
108         else
109             temp_list_itr = OPC_NIL;
110
111         if( i < (num_elems-1))
112             list_itr = prg_list_cell_next_get(list_itr);
113
114         //after moving the iterator to the next cell, deallocate the old cell
115         if(temp_list_itr != OPC_NIL)
116             prg_list_cell_remove (PPATT, temp_list_itr);
117     }
118
119     op_intrpt_schedule_call (op_sim_time() + (DSR_EXT_PPATT_CLEANUP/2.0), 0, dsr_ext_cleanup_PPATT, ptr_flags);
120
121     FOUT;
122 }
123
124 /* decides whether a particular flow metric qualifies for rebroadcast */
125 static Boolean dsr_ext_is_better_route(InetT_Address src_addr, InetT_Address dest_addr, int flow_id, double metric)
126 {
127     PPATT_Element*                elem_ptr;
128     PrgT_List_Cell*             list_itr;
129     int                           i,count;
130
131     FIN(dsr_ext_is_better_route(<args>));
132
133     count = prg_list_size(PPATT);
134     list_itr = prg_list_head_cell_get(PPATT);
135     for( i=0; i< count; i++)
136     {
137         elem_ptr = (PPATT_Element*)prg_list_cell_data_get(list_itr);
138
139         //match all three critiria to identify a flow uniquely
140         if( inet_address_equal(elem_ptr->dest_addr, dest_addr) == OPC_TRUE &&
141             inet_address_equal(elem_ptr->src_addr, src_addr) == OPC_TRUE &&
142             elem_ptr->flow_id == flow_id)
143         {
144             elem_ptr->timestamp = op_sim_time();//update the access time for cleanup control
145             if( metric < elem_ptr->metric) //less than is better
146             {
147                 elem_ptr->metric = metric;//improve the threshhold
148                 FRET(OPC_TRUE);
149             }
150             else
```

```
151                 FRET(OPC_FALSE);
152         }
153         if( i < (count-1))
154             list_itr = prg_list_cell_next_get(list_itr);
155     }
156     //if we get this far, the destination is not in our table, ADD THE DESTINATION
157     elem_ptr = (PPATT_Element*)op_prg_mem_alloc(sizeof(PPATT_Element));
158     elem_ptr->dest_addr = inet_address_copy(dest_addr);
159     elem_ptr->src_addr = inet_address_copy(src_addr);
160     elem_ptr->flow_id = flow_id;
161     elem_ptr->metric = metric;
162     elem_ptr->timestamp = op_sim_time();
163 
164     //insert into PPATT list
165     op_prg_list_insert (PPATT, elem_ptr, OPC_LISTPOS_TAIL);
166 
167     FRET(OPC_TRUE);
168 }
169 
170 #ifdef ENABLE_DSR_YANG_DELAY
171 //this function adds traffic to the model given the expected load from this route discovery
172 static void dsr_ext_augment_lambda_data(List* lambda_data, Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr)
173 {
174     int                         i,j,count;
175     Dsr_Ext_Neighbor_Data*        neighbor_ptr;
176     DsrT_Route_Request_Option*    route_request_option_ptr = OPC_NIL;
177     IpT_Dgram_Fields*             ip_dgram_fd_ptr = OPC_NIL;
178     InetT_Address*                 hop_address_ptr;
179     PrgT_List_Cell                 *list_itr;
180     Dsr_Ext_Yangs_Eta_Data        *lambda_data_elem = OPC_NIL;
181     FIN(dsr_ext_augment_lambda_data(<args>));
182 
183     //for each hop in the source route,
184         //if the hop is in the interference range of this node,
185             //add it o the list
186     route_request_option_ptr = (DsrT_Route_Request_Option*) dsr_tlv_ptr->dsr_option_ptr;
187     count = op_prg_list_size(route_request_option_ptr->route_lptr);
188     list_itr = prg_list_head_cell_get (route_request_option_ptr->route_lptr);
189     for(i=0; i<count; i++)
190     {
191         hop_address_ptr = (InetT_Address*) prg_list_cell_data_get(list_itr);
192         neighbor_ptr = dsr_ext_get_neighbor_ptr(*hop_address_ptr);
193         if(neighbor_ptr->interference_range == OPC_TRUE)
194         {
195             //iterate through the priority classes
196             for(j=0; j<DSR_EXT_NUM_RATES; j++)
197             {
198                 if(neighbor_ptr->rates[j] <= 0.0)
199                     continue;
200 
201                 //create a new element to add to the eta_data list
202                 lambda_data_elem = (Dsr_Ext_Yangs_Eta_Data*) op_prg_mem_alloc (sizeof(Dsr_Ext_Yangs_Eta_Data));
203 
204                 lambda_data_elem->pk_size = dsr_ext_get_pk_size_from_priority(j);
205                 lambda_data_elem->cw_size = dsr_ext_get_cw_from_priority(j);
206                 lambda_data_elem->pk_rate = neighbor_ptr->rates[j];
207 
208                 op_prg_list_insert (lambda_data, lambda_data_elem, OPC_LISTPOS_TAIL);
209             }
210         }
211         if( i < (count-1) )
212             list_itr = prg_list_cell_next_get(list_itr);
213     }
214 
215     //include the source node as a transmitting node
216     op_pk_nfd_access(ip_pkptr, "fields", &ip_dgram_fd_ptr);
217     neighbor_ptr = dsr_ext_get_neighbor_ptr(ip_dgram_fd_ptr->src_addr);
218     if(neighbor_ptr->interference_range == OPC_TRUE)
219     {
220         //iterate through the priority classes
221         for(j=0; j<DSR_EXT_NUM_RATES; j++)
222         {
223             if(neighbor_ptr->rates[j] <= 0.0)
224                 continue;
225 
226             //create a new element to add to the eta_data list
227             lambda_data_elem = (Dsr_Ext_Yangs_Eta_Data*) op_prg_mem_alloc (sizeof(Dsr_Ext_Yangs_Eta_Data));
228 
229             lambda_data_elem->pk_size = dsr_ext_get_pk_size_from_priority(j);
230             lambda_data_elem->cw_size = dsr_ext_get_cw_from_priority(j);
231             lambda_data_elem->pk_rate = neighbor_ptr->rates[j];
232 
233             op_prg_list_insert (lambda_data, lambda_data_elem, OPC_LISTPOS_TAIL);
234         }
235     }
236     FOUT;
237 }
```

```
238
239 /* Computes an average packet delay estimate (not a jitter estimate) from yyang8, which describes an average packet delay model*/
240 static double dsr_ext_compute_yangs_delay(Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr)
241 {
242     int                         i, up, packet_window_size, lambda_list_size;
243     double                     lambda_sum, delay_estimate = 0.0, temp_pi;
244     List                         *lambda_data = OPC_NIL;
245     IpT_Dgram_Fields             *ip_dgram_fd_ptr = OPC_NIL;
246     PrgT_List_Cell               *list_itr = OPC_NIL;
247     Dsr_Ext_Yangs_Eta_Data       *lambda_data_elem = OPC_NIL;
248     FIN(dsr_ext_compute_yangs_delay(<args>));
249
250     op_pk_nfd_access( ip_pkptr, "fields", &ip_dgram_fd_ptr);
251     up = dsr_ext_map_ip_to_mac_priority(ip_dgram_fd_ptr->tos >> 5);
252     up = dsr_ext_map_ip_to_mac_priority(up);
253     packet_window_size = dsr_ext_get_cw_from_priority(up);
254
255     //get a list of the current neighborhood flows and their contention window sizes
256     lambda_data = op_prg_list_create();
257     dsr_ext_fill_yangs_eta_data(lambda_data, DSR_EXT_NUM_RATES-1);
258
259     //augment list with data from the dsr source route
260     dsr_ext_augment_lambda_data(lambda_data, ip_pkptr, dsr_tlv_ptr);
261
262     //compute C1*Wi/2
263     delay_estimate = DSR_EXT_C1 * dsr_ext_get_cw_from_priority(up) / 2;
264
265     //compute sum(lamda/x) for existing flows and the anticipated path
266     lambda_sum = 0.0;
267     lambda_list_size = op_prg_list_size(lambda_data);
268     list_itr = prg_list_head_cell_get (lambda_data);
269     for(i=0; i<lambda_list_size; i++)
270     {
271         lambda_data_elem = (Dsr_Ext_Yangs_Eta_Data*) prg_list_cell_data_get(list_itr);
272         lambda_sum += lambda_data_elem->pk_rate;
273
274         if( i < (lambda_list_size-1) )
275             list_itr = prg_list_cell_next_get(list_itr);
276     }
277     //multiply lambda_sum(packets/sec) by transmission time for one packet(seconds/packet)
278     //to get a ratio of airtime comsumed by the neighborhood
279     lambda_sum *= DSR_EXT_Td;
280
281     //multiply airtime ratio into estimate calculation
282     delay_estimate *= lambda_sum;
283
284     //compute C2 - PI(1- 2*lambda*Td/Wj)
285     temp_pi = 1.0;
286     list_itr = prg_list_head_cell_get (lambda_data);
287     for(i=0; i<lambda_list_size; i++)
288     {
289         lambda_data_elem = (Dsr_Ext_Yangs_Eta_Data*) prg_list_cell_data_get(list_itr);
290         temp_pi *= (1 - (2 * lambda_data_elem->pk_rate * DSR_EXT_Td / lambda_data_elem->cw_size));
291         if( i < (lambda_list_size-1) )
292             list_itr = prg_list_cell_next_get(list_itr);
293     }
294     temp_pi = DSR_EXT_C2 - temp_pi;
295
296     //add C3
297     delay_estimate += DSR_EXT_C3;
298
299     //free up labda_data list data structure
300     dsr_ext_list_mem_free(lambda_data);
301     FRET(delay_estimate);
302 }
303 #endif
304
305 static Boolean dsr_ext_path_admit(Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr, Boolean partial)
306 {
307     int                      alpha,priority; //# of nodes in the route within this interference range
308     double                  local_available_bandwidth, neighborhood_available_bandwidth;
309     double                  comp_bandwidth_required; //stores the bandwidth requirment for the flow
310     Boolean             accept;
311     Packet*             dsr_pkptr;
312     Dsr_Ext_Route_Metrics*      dsr_metrics;
313     IpT_Dgram_Fields* ip_dgram_fd_ptr = OPC_NIL;
314 #ifdef ENABLE_DSR_YANG_DELAY
315     double                  hop_cost,delay_estimate; //stores delay estimate up to this node
316 #endif
317
318     FIN(dsr_ext_partial_path_admit(Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr));
319
320     op_pk_nfd_access( ip_pkptr, "fields", &ip_dgram_fd_ptr);
321     priority = dsr_ext_map_ip_to_mac_priority(ip_dgram_fd_ptr->tos >> 5);
322
323     op_pk_nfd_get(ip_pkptr, "data", &dsr_pkptr);
324     op_pk_nfd_access(dsr_pkptr, "Metrics", &dsr_metrics);
```

```
325     op_pk_nfd_set(ip_pkptr, "data", dsr_pkptr);
326
327     comp_bandwidth_required = dsr_metrics->pk_size * dsr_metrics->pk_rate;
328     accept = OPC_TRUE;
329
330 #ifdef ENABLE_DSR_YANG_BANDWIDTH
331     if(dsr_tlv_ptr == OPC_NIL)
332         alpha = 1;
333     else
334         alpha = dsr_ext_compute_yangs_alpha(ip_pkptr, dsr_tlv_ptr, partial);
335
336     local_available_bandwidth = dsr_ext_compute_yangs_local_available_bandwidth(alpha, priority);
337
338     if(local_available_bandwidth < comp_bandwidth_required)
339         accept = OPC_FALSE;
340
341     neighborhood_available_bandwidth = dsr_ext_compute_yangs_neighborhood_available_bandwidth(alpha, priority);
342
343     if(neighborhood_available_bandwidth < comp_bandwidth_required)
344         accept = OPC_FALSE;
345
346     //SPEED METRIC
347     hop_cost = DSR_EXT_METRIC_SPEED * curr_speed / MAX_SPEED;
348
349     //BANDWIDTH METRIC
350     if(local_available_bandwidth < neighborhood_available_bandwidth)
351         neighborhood_available_bandwidth = local_available_bandwidth;//select the lowest of the two
352     hop_cost += (CHANNEL_BANDWIDTH - neighborhood_available_bandwidth) / CHANNEL_BANDWIDTH ;
353 #endif
354 #ifdef ENABLE_DSR_YANG_DELAY
355     delay_estimate = dsr_ext_compute_yangs_delay(ip_pkptr, dsr_tlv_ptr);
356     dsr_metrics->delay_est += delay_estimate;
357     if( dsr_metrics->delay_req > 0.0 &&  dsr_metrics->delay_req  < dsr_metrics->delay_est)
358     {
359         accept = OPC_FALSE;
360     }
361
362     //DELAY METRIC
363     hop_cost += DSR_EXT_METRIC_DELAY * dsr_metrics->delay_est / 0.02;//normalize for "peak delay" of 20ms
364 #endif
365     dsr_metrics->cost += hop_cost;
366     FRET(accept);
367 }
368
369
370 static float dsr_ext_resolve_metric(Dsr_Ext_Route_Metrics*    dsr_metrics)
371 {
372     double result;
373     FIN(dsr_ext_resolve_metric(Dsr_Ext_Route_Metrics*    dsr_metrics));
374     result = dsr_metrics->hop_count;
375 #ifdef ENABLE_DSR_YANG_BANDWIDTH
376     //cost computed in path_admit (directly above)
377     result = dsr_metrics->cost;
378 #endif
379     FRET((float)result);
380 }
381
382 /* RREQ of local origin will need to have a fresh route metric data structure initialized */
383 static Dsr_Ext_Route_Metrics* dsr_ext_create_route_request_metrics(InetT_Address dest_address, int flow_id)
384 {
385     Dsr_Ext_Route_Metrics*     dsr_metrics;
386     Dsr_Ext_Connection*        connection_ptr;
387     FIN(dsr_ext_create_route_request_metrics())
388
389     // expiration function for a connection pointer is different than the acceptance window.
390     //     on RRPLY, delete this connection pointer
391     connection_ptr = dsr_ext_get_connection_ptr(dest_address, flow_id, DSRC_ROUTE_REQUEST);
392     connection_ptr->is_new_connection = OPC_FALSE;
393
394     /* create a newroute metric data structure */
395     dsr_metrics = op_prg_mem_alloc (sizeof(Dsr_Ext_Route_Metrics));
396
397     dsr_metrics->pk_rate = connection_ptr->pk_rate;
398     dsr_metrics->pk_size = connection_ptr->pk_size;
399     dsr_metrics->delay_req = connection_ptr->delay_req;
400     dsr_metrics->delay_est = 0.0;
401     dsr_metrics->hop_count = 1;
402     dsr_metrics->max_bandwidth = 1e20;//near infinite bandwidth for the the first hop to itself
403     dsr_metrics->cost = 0;
404
405     FRET(dsr_metrics);
406 }
407
408 /* update the multihop route metric contained in the dsr packet with local data */
409 static double dsr_ext_update_route_request_metrics(InetT_Address src_addr, Packet* dsr_pkptr, int up)
410 {
411     double                 evaluation, last_hop_ETT;
```

```
412     double                 average_last_hop_bandwidth;
413     Dsr_Ext_Route_Metrics*     dsr_metrics;
414     Dsr_Ext_Neighbor_Data*     neighbor_ptr;
415     int                    i, prio_class;
416
417     FIN(dsr_ext_update_route_request_metrics(Dsr_Ext_Route_Metrics* dsr_metrics));
418
419     prio_class = dsr_ext_map_ip_to_mac_priority(up);
420
421     op_pk_nfd_access(dsr_pkptr, "Metrics", &dsr_metrics);
422     neighbor_ptr = dsr_ext_get_neighbor_ptr(src_addr);
423     dsr_metrics->hop_count++;
424
425     //compare the metric bandwidth cap with the last-hop bandwith, use the smaller of the two
426
427     //average the ring buffer
428     average_last_hop_bandwidth = 0.0;
429     for(i=0; i < DSR_EXT_NUM_BW_MEASUREMENTS; i++)
430         average_last_hop_bandwidth += neighbor_ptr->ETT_data[prio_class].rv_bw_measurement[i];
431     average_last_hop_bandwidth /= DSR_EXT_NUM_BW_MEASUREMENTS;
432
433     if(    dsr_metrics->max_bandwidth > average_last_hop_bandwidth)
434         dsr_metrics->max_bandwidth = average_last_hop_bandwidth;
435
436     if(average_last_hop_bandwidth < 1e-6 ||
437         neighbor_ptr->probability_forward < 0.1 ||
438         neighbor_ptr->probability_reverse < 0.1 )
439         last_hop_ETT = 1e10;
440     else
441         last_hop_ETT = 1.0/(neighbor_ptr->probability_forward  * neighbor_ptr->probability_reverse) * dsr_metrics->pk_size /
average_last_hop_bandwidth;
442
443     dsr_metrics->CETT += last_hop_ETT;
444     evaluation = dsr_ext_resolve_metric(dsr_metrics);
445     FRET(evaluation);
446 }
447
448 /* A route accept window expires a short time after the first route request packet from a node arrives.
449  * At this point the acceptance window is being closed and we want to select the best route.
450  */
451 static void dsr_ext_route_accept_window_expiry(void* v_connection_ptr, int code)
452 {
453     Dsr_Ext_Connection*     connection_ptr;
454     PrgT_List_Cell *        list_itr;
455     Packet*                    source_ip_pkptr;
456     Packet*                    dsr_pkptr;
457     Packet*                 best_route_pk_ptr;
458     List*                    tlv_options_lptr = OPC_NIL;
459     int                     i, num_elems;
460     double                     best_route_metric_value = 1e99;
461     double                   temp_metric_value;
462
463     Dsr_Ext_Route_Metrics*    dsr_metrics;
464     DsrT_Packet_Option* dsr_tlv_ptr;
465     FIN(dsr_ext_route_accept_window_expiry(void* ptr_flags, int code));
466     connection_ptr = (Dsr_Ext_Connection*) v_connection_ptr;
467
468
469     //sanity check the element list for impossible size
470     num_elems = prg_list_size (connection_ptr->source_pk_list_ptr);
471     if(num_elems < 1)
472     {
473         dsr_ext_destroy_connection_ptr(connection_ptr);
474         FOUT;
475     }
476
477     //search the list for the single best packet, according to our metric
478     list_itr = prg_list_head_cell_get (connection_ptr->source_pk_list_ptr);
479     source_ip_pkptr = (Packet*) prg_list_cell_data_get(list_itr);
480     best_route_pk_ptr = source_ip_pkptr;
481     for (i = 0; i < num_elems; i++)
482     {
483         op_pk_nfd_get(source_ip_pkptr, "data", &dsr_pkptr);
484         op_pk_nfd_get(dsr_pkptr, "Metrics", &dsr_metrics);
485         temp_metric_value = dsr_ext_resolve_metric(dsr_metrics);
486         op_pk_nfd_set_ptr(dsr_pkptr, "Metrics", dsr_metrics, op_prg_mem_copy_create,
487                           op_prg_mem_free, sizeof (Dsr_Ext_Route_Metrics));
488         if( temp_metric_value < best_route_metric_value )
489         {
490             best_route_metric_value = temp_metric_value;
491             best_route_pk_ptr = source_ip_pkptr;
492         }
493         op_pk_nfd_set(source_ip_pkptr, "data", dsr_pkptr);
494
495         if(i < num_elems-1)
496         {
497             list_itr = prg_list_cell_next_get(list_itr);
```

```
498             source_ip_pkptr = (Packet*) prg_list_cell_data_get(list_itr);
499         }
500     }
501 
502     //respond to a REQ by replying to a subset of best routes
503     //react to a REP by accepting the best route
504     if(code == DSRC_ROUTE_REQUEST)
505     {
506 #ifdef ENABLE_DSR_YANG_BANDWIDTH
507         //send a RREPLY using the best source route
508         dsr_ext_route_reply_send(best_route_pk_ptr);
509         op_intrpt_schedule_call (op_sim_time() + DSR_EXT_DELAYED_ACCEPT_DELETE , 0, dsr_ext_delayed_connection_delete,
(void*)connection_ptr);
510 #else
511         //partial admission control is all that's being done, just use the single best route
512         dsr_ext_route_reply_send (best_route_pk_ptr);
513         //the connection pointer can be destroyed immediatly
514         dsr_ext_destroy_connection_ptr(connection_ptr);
515 #endif
516         //schedule the destruction of the RREQ connection data
517     }
518     else if(code == DSRC_ROUTE_REPLY)
519     {
520 
521 #ifdef ENABLE_DSR_YANG_DELAY
522         int flow_id;
523         //stored in the dsr_metrics should be the delay_estimate from the dest node
524         //store the value to the flow_id slot in the global_delay_estimate
525         op_pk_nfd_get(best_route_pk_ptr, "data", &dsr_pkptr);
526         op_pk_nfd_get(dsr_pkptr, "flow_id", &flow_id);
527         op_pk_nfd_set(dsr_pkptr, "flow_id", flow_id);
528         op_pk_nfd_access(dsr_pkptr, "Metrics", &dsr_metrics);
529 
530         global_delay_estimate[flow_id] = dsr_metrics->delay_est;
531 
532         op_pk_nfd_set(best_route_pk_ptr, "data", dsr_pkptr);
533 
534 #endif
535         /* Record the successfull route discover in the global statistic*/
536         total_routes_discovered += 1.0;
537         op_stat_write (total_routes_discovered_shandle, 1.0);
538         //display selected route
539         dsr_tlv_ptr = get_tlv_from_ip_pkptr(best_route_pk_ptr, DSRC_ROUTE_REPLY);
540         print_route( dsr_tlv_ptr);
541         //accept (cache) the best route reply recieved during source accept window
542         dsr_ext_route_cache_update (best_route_pk_ptr);
543 
544         //the connection pointer can be destroyed immediatly
545         dsr_ext_destroy_connection_ptr(connection_ptr);
546     }
547     FOUT;
548 }
549 
550 #ifdef ENABLE_DSR_YANG_BANDWIDTH
551 
552 static int dsr_ext_map_ip_to_mac_priority(int up)
553 {
554     int priority;
555     if(up == 0 || up == 3)        priority = DSR_PRIORITY_IMPORTANT;
556     else if(up == 1 || up == 2)    priority = DSR_PRIORITY_BESTEFFORT;
557     else if(up == 4 || up == 5)    priority = DSR_PRIORITY_REALTIME;
558     else if(up == 6 || up == 7)    priority = DSR_PRIORITY_CRITICAL;
559     else dsr_rte_error("Invalid flow priority.","Type of Service field must be [0,7].","");
560     return priority;
561 }
562 
563 /* updates the time stamp for packets from a particular flow */
564 static void dsr_ext_update_flow_reservation(Packet* ip_pkptr)
565 {
566     int                             i, num_elems, flow_id, priority;
567     Boolean                         flow_is_in_list;
568     Dsr_Ext_Flow_Status_Element*     elem_ptr;
569     IpT_Dgram_Fields*                ip_dgram_fd_ptr = OPC_NIL;
570     IpT_Rte_Ind_Ici_Fields*         intf_ici_fdstruct_ptr = OPC_NIL;
571     PrgT_List_Cell *                list_itr;
572     Packet*                            dsr_pkptr;
573     Packet*                            qos_pkptr;
574     int                             pk_size;
575     double                           pk_rate;
576     char                             pk_format[128];
577 
578     FIN(dsr_ext_update_flow_reservation(Packet* qos_packet));
579 
580     manet_rte_ip_pkt_info_access (ip_pkptr, &ip_dgram_fd_ptr, &intf_ici_fdstruct_ptr);
581 
582     //extract the priority from the tos fields
583     priority = dsr_ext_map_ip_to_mac_priority(ip_dgram_fd_ptr->tos >> 5);
```

```
584
585     if(priority == DSR_PRIORITY_BESTEFFORT)
586     {
587         FOUT;//do not process best effor packets
588     }
589     //extract QoS data from packet
590     op_pk_nfd_get(ip_pkptr, "data", &dsr_pkptr);
591
592     if(op_pk_nfd_is_set (dsr_pkptr, "data") == OPC_FALSE)
593     {
594         op_pk_format(dsr_pkptr, pk_format);
595         if (strcmp (pk_format, "manet_qos_packet") == 0)
596         {
597             //sometimes the DSR protocol will send one-hop neighbors packets without DSR headers
598             printf("QoS where a DSR packet should be, line %d.\n", __LINE__-315);
599             FOUT;
600         }
601         else
602         {
603             //printf("not a QoS packet:%s\n", pk_format);
604             op_pk_nfd_set(ip_pkptr, "data", dsr_pkptr);
605             FOUT;
606         }
607     }
608     else
609     {
610         op_pk_nfd_get(dsr_pkptr, "data", &qos_pkptr);
611     }
612
613     op_pk_nfd_access(qos_pkptr, "flow_id", &flow_id);
614     op_pk_nfd_access(qos_pkptr, "pk_rate", &pk_rate);
615     op_pk_nfd_access(qos_pkptr, "pk_size", &pk_size);
616     op_pk_nfd_set(dsr_pkptr, "data", qos_pkptr);
617     op_pk_nfd_set(ip_pkptr, "data", dsr_pkptr);
618
619     //search for the flow in the associated priority bin, and update the timestamp for the flow
620     flow_is_in_list = OPC_FALSE;
621     num_elems = prg_list_size (flow_reservations[priority]);
622     list_itr = prg_list_head_cell_get (flow_reservations[priority]);
623     for(i=0; i<num_elems; i++)
624     {
625         elem_ptr = (Dsr_Ext_Flow_Status_Element*) prg_list_cell_data_get(list_itr);
626         if (inet_address_equal (elem_ptr->src_addr, ip_dgram_fd_ptr->src_addr) == OPC_TRUE  &&
627             inet_address_equal (elem_ptr->src_addr, ip_dgram_fd_ptr->src_addr) == OPC_TRUE  &&
628             elem_ptr->flow_id == flow_id )
629         {
630             flow_is_in_list = OPC_TRUE;
631             elem_ptr->timestamp = op_sim_time();
632             break;
633         }
634
635         if( i < (num_elems-1) )
636             list_itr = prg_list_cell_next_get(list_itr);
637     }
638
639     //there is no entry for this flow, so create one
640     if(flow_is_in_list == OPC_FALSE)
641     {
642     //create a new entry for this flow
643         elem_ptr = (Dsr_Ext_Flow_Status_Element*) op_prg_mem_alloc (sizeof(Dsr_Ext_Flow_Status_Element));
644         elem_ptr->src_addr  = inet_address_copy (ip_dgram_fd_ptr->src_addr);
645         elem_ptr->dest_addr = inet_address_copy (ip_dgram_fd_ptr->dest_addr);
646         elem_ptr->flow_id = flow_id;
647         elem_ptr->pk_rate = pk_rate;
648         elem_ptr->pk_size = pk_size;
649         elem_ptr->timestamp = op_sim_time();
650
651         //insert this new entry into the appropriate flow reservation bin
652         op_prg_list_insert (flow_reservations[priority], elem_ptr, OPC_LISTPOS_HEAD);
653     }
654
655     FOUT;
656 }
657
658 /* searches the flow state tables for old entries and removes them */
659 static void dsr_ext_cleanup_flow_reservations(void* ptr_flags, int code)
660 {
661     int                             i, j, num_elems;
662     Dsr_Ext_Flow_Status_Element*     elem_ptr;
663     PrgT_List_Cell                    *list_itr, *temp_list_itr;
664
665     FIN(dsr_ext_cleanup_flow_reservations(void* ptr_flags, int code));
666
667     //scan each reservation list for expired flows, and remove them
668     for(i=0; i<DSR_EXT_NUM_RATES;i++)
669     {
670         num_elems = prg_list_size (flow_reservations[i]);
```

```
671         list_itr = prg_list_head_cell_get (flow_reservations[i]);
672         for(j=0; j<num_elems; j++)
673         {
674             elem_ptr = (Dsr_Ext_Flow_Status_Element*) prg_list_cell_data_get(list_itr);
675             if ( op_sim_time() - elem_ptr->timestamp >= DSR_EXT_FLOW_RESERVATION_CLEANUP)
676             {
677                 op_prg_mem_free(elem_ptr);
678                 temp_list_itr = list_itr;
679             }
680             else
681                 temp_list_itr = OPC_NIL;
682 
683             if( j < (num_elems-1) )
684                 list_itr = prg_list_cell_next_get(list_itr);
685 
686             //after moving the iterator to the next cell, deallocate the old cell
687             if(temp_list_itr != OPC_NIL)
688                 prg_list_cell_remove (flow_reservations[i], temp_list_itr);
689         }
690     }
691     op_intrpt_schedule_call (op_sim_time() + DSR_EXT_FLOW_RESERVATION_CLEANUP, 0, dsr_ext_cleanup_flow_reservations, ptr_flags);
692 
693     FOUT;
694 }
695 static double dsr_ext_calc_distance(double* a, double* b)
696 {
697     double temp;
698     FIN(dsr_ext_calc_distance(double* a, double* b));
699     temp =      (a[0] - b[0]) * (a[0] - b[0]) +
700             (a[1] - b[1]) * (a[1] - b[1]) +
701             (a[2] - b[2]) * (a[2] - b[2]);
702     if(temp < 0.0001)
703         FRET(0.0);
704     temp = sqrt(temp);
705     FRET(temp);
706 }
707 
708 static Boolean dsr_ext_is_interference_neighbor(double loc[], double range)
709 {
710     double temp,here[3];
711     FIN(dsr_ext_is_interference_neighbor(float loc[], float range));
712     op_ima_obj_pos_get(op_topo_parent(op_id_self()), &temp,&temp,&temp,&here[0],&here[1],&here[2]);
713 
714     temp = dsr_ext_calc_distance(here, loc);
715     if( temp > range )
716     {
717         FRET(OPC_FALSE);
718     }
719     FRET(OPC_TRUE);
720 }
721 
722 //counts how many transmiting nodes are in this nodes interference range
723 static int dsr_ext_num_transmitters_in_interference_range()
724 {
725     int                         i,num_elems,count;
726     Dsr_Ext_Neighbor_Data*        neighbor_ptr;
727     DsrT_Route_Request_Option*    route_request_option_ptr = OPC_NIL;
728     IpT_Dgram_Fields*             ip_dgram_fd_ptr = OPC_NIL;
729     PrgT_List_Cell                *list_itr;
730 
731     FIN(dsr_ext_num_neighbors_in_interference_range());
732 
733     count = 1; //include self
734     num_elems = op_prg_list_size(neighbor_list);
735     list_itr = prg_list_head_cell_get (neighbor_list);
736     for(i=0; i<num_elems; i++)
737     {
738         neighbor_ptr = (Dsr_Ext_Neighbor_Data*) prg_list_cell_data_get(list_itr);
739         if(neighbor_ptr->interference_range == OPC_TRUE)
740         {
741             count++;
742         }
743         if( i < (num_elems-1) )
744             list_itr = prg_list_cell_next_get(list_itr);
745 
746     }
747     FRET( count);
748 }
749 
750 /* fills the rates array */
751 static void dsr_ext_fill_flow_reservation_data(double rates[])
752 {
753     int                             i,j, num_elems;
754     double                            sum;
755     Dsr_Ext_Flow_Status_Element*     elem_ptr;
756     PrgT_List_Cell                    *list_itr;
757 
```

```
758     FIN(dsr_ext_fill_flow_reservation_data(double rates[]));
759
760     //aggregate flow data from flow reservation table
761     for(i=0; i<DSR_EXT_NUM_RATES; i++)
762     {
763         /*account for protocol overhead created by this node*/
764         double n = dsr_ext_num_transmitters_in_interference_range();
765
766         //model sharing the rebroadcast load, as well as generating some load
767         if(n < DSR_EXT_OVERHEAD_CONST)
768             n = n*n;
769         else
770             n = (((DSR_EXT_OVERHEAD_CONST + 1)*n)-DSR_EXT_OVERHEAD_CONST)/n;
771
772         if(i==DSR_PRIORITY_IMPORTANT)
773             sum = n/DSR_EXT_FLOW_STATE_BCAST_INTERVAL;
774         else
775             sum = 0.0;
776
777         num_elems = prg_list_size (flow_reservations[i]);
778         list_itr = prg_list_head_cell_get (flow_reservations[i]);
779         for(j=0; j<num_elems; j++)
780         {
781             elem_ptr = (Dsr_Ext_Flow_Status_Element*) prg_list_cell_data_get(list_itr);
782
783             //calculate the rate associated each flow, only if the values are valid
784             sum += elem_ptr->pk_rate;
785
786             if( j < (num_elems-1) )
787                 list_itr = prg_list_cell_next_get(list_itr);
788
789         }
790
791         //store the bandwidth consumed by this node for this priority class
792         rates[i] = sum;
793     }
794     FOUT;
795 }
796
797 static void dsr_ext_broadcast_flow_state(void* ptr_flags, int code)
798 {
799     Dsr_Ext_Rate_Reservations* rate_ptr;
800     Packet* flow_pkptr;
801     double temp, delay;
802     int i;
803
804     FIN(dsr_ext_broadcast_flow_state(void* ptr_flags, int code));
805
806     rate_ptr = op_prg_mem_alloc (sizeof(Dsr_Ext_Rate_Reservations));
807
808     //fille the rate info
809     dsr_ext_fill_flow_reservation_data(rate_ptr->rates);
810
811     /* get physical location of this node */
812     op_ima_obj_pos_get(op_topo_parent(op_id_self()), &temp,&temp,&temp,
813                        &(rate_ptr->loc[0]),&(rate_ptr->loc[1]),&(rate_ptr->loc[2]));
814
815     //use the rate_ptr as a temporary storage to compute velocity
816     if(op_sim_time() == 0.0)
817         curr_speed = 0.0;
818     else
819         curr_speed = dsr_ext_calc_distance(rate_ptr->loc, prev_location) / (op_sim_time() - prev_location_timestamp);
820     for(i=0; i<3; i++)
821         prev_location[i] = rate_ptr->loc[i];
822     prev_location_timestamp = op_sim_time();
823
824     //build a flow state packet
825     flow_pkptr = op_pk_create_fmt("dsr_ext_flow_state_packet");
826
827     //set the data in the packet
828     op_pk_nfd_set_ptr(flow_pkptr, "flow_data",rate_ptr, op_prg_mem_copy_create,
829                       op_prg_mem_free, sizeof (Dsr_Ext_Rate_Reservations));
830
831     op_stat_write (broadcast_overhead_pkts_shandle, 1.0);
832     op_stat_write (broadcast_overhead_bits_shandle, op_pk_total_size_get (flow_pkptr));
833
834     //broadcast the packet
835     dsr_ext_send_packet(flow_pkptr, InetI_Broadcast_v4_Addr, 0);
836
837     //apply a +/- 10% jitter on the broadcast interval
838     delay = DSR_EXT_FLOW_STATE_BCAST_INTERVAL * 0.9 + op_dist_uniform (DSR_EXT_FLOW_STATE_BCAST_INTERVAL * 0.2);
839
840     op_intrpt_schedule_call (op_sim_time() + delay, 0, dsr_ext_broadcast_flow_state, ptr_flags);
841
842     FOUT;
843 }
844
```

```
845 // store_neightbor_flow_state  checks to see if recording a flow state is terriby premature, if so, FALSE is returned,
846 // otherwise the data is recorded, and TRUE is returned */
847 static Boolean dsr_ext_store_neighbor_flow_state(InetT_Address src_addr, Dsr_Ext_Rate_Reservations* info_ptr)
848 {
849     int i;
850     Dsr_Ext_Neighbor_Data*        neighbor_ptr;
851 
852     FIN(dsr_ext_store_neighbor_flow_state());
853 
854     neighbor_ptr = dsr_ext_get_neighbor_ptr(src_addr);
855 
856     /* echo filter, make sure at least %80 of the normal interval has passed before accepting a flow state broadcast
857         This identifies unique packet reception, and is nessesary to prevent reverberation of flow broadcasts.    */
858     if( (op_sim_time() - neighbor_ptr->last_flow_update) < (DSR_EXT_FLOW_STATE_BCAST_INTERVAL * 0.8))
859     {
860         FRET(OPC_FALSE);
861     }
862 
863     neighbor_ptr->last_flow_update = op_sim_time();
864     neighbor_ptr->interference_range = OPC_TRUE;
865     for(i=0; i<DSR_EXT_NUM_RATES; i++)
866     {
867         neighbor_ptr->rates[i] = info_ptr->rates[i];
868     }
869 
870     FRET(OPC_TRUE);
871 }
872 
873 //counts the nodes in the source list of the packet to the nodes recorded as being in the interference range
874 //includes Tang's next hop estimation
875 static int dsr_ext_compute_yangs_alpha(Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr, Boolean estimate)
876 {
877     int                          i,count,alpha;
878     Dsr_Ext_Neighbor_Data*        neighbor_ptr;
879     DsrT_Route_Request_Option*    route_request_option_ptr = OPC_NIL;
880     IpT_Dgram_Fields*             ip_dgram_fd_ptr = OPC_NIL;
881     InetT_Address*                hop_address_ptr;
882     PrgT_List_Cell                *list_itr;
883 
884     FIN(dsr_ext_compute_yangs_alpha(Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr));
885 
886     alpha = 0;
887     route_request_option_ptr = (DsrT_Route_Request_Option*) dsr_tlv_ptr->dsr_option_ptr;
888     count = op_prg_list_size(route_request_option_ptr->route_lptr);
889     list_itr = prg_list_head_cell_get (route_request_option_ptr->route_lptr);
890     for(i=0; i<count; i++)
891     {
892         hop_address_ptr = (InetT_Address*) prg_list_cell_data_get(list_itr);
893         neighbor_ptr = dsr_ext_get_neighbor_ptr(*hop_address_ptr);
894         if(neighbor_ptr->interference_range == OPC_TRUE)
895         {
896             alpha++;
897         }
898         if( i < (count-1) )
899             list_itr = prg_list_cell_next_get(list_itr);
900     }
901 
902     //include the source node as a transmitting node
903     op_pk_nfd_access(ip_pkptr, "fields", &ip_dgram_fd_ptr);
904 
905     neighbor_ptr = dsr_ext_get_neighbor_ptr(ip_dgram_fd_ptr->src_addr);
906     if(neighbor_ptr->interference_range == OPC_TRUE)
907     {
908         alpha++;
909     }
910 
911     //Tangs estimate
912     if(estimate)
913     {
914         //if the destination is not a one hop neighbor, increment alpha one more time
915         neighbor_ptr = dsr_ext_get_neighbor_ptr(route_request_option_ptr->target_address);
916 
917         if(neighbor_ptr->interference_range == OPC_FALSE)
918         {
919             alpha++;
920         }
921     }
922     FRET(alpha);
923 }
924 
925 int dsr_ext_yangs_eta_data_sort_proc(const void* aptr, const void* bptr)
926 {
927     Dsr_Ext_Yangs_Eta_Data* beta = (Dsr_Ext_Yangs_Eta_Data*) aptr;
928     Dsr_Ext_Yangs_Eta_Data* gamma = (Dsr_Ext_Yangs_Eta_Data*) bptr;
929 
930     if(beta->eta > gamma->eta)
931         return -1;
```

```
932     if(beta->eta < gamma->eta)
933         return  1;
934     else
935         return  0;
936 }
937
938 static void dsr_ext_fill_yangs_eta_data(List* eta_star, int prio)
939 {
940     int                             i,j,count;
941     double                           rates[DSR_EXT_NUM_RATES];
942     PrgT_List_Cell*                   list_itr;
943     Dsr_Ext_Yangs_Eta_Data*           eta_star_elem_ptr;
944     Dsr_Ext_Neighbor_Data*           neighbor_ptr;
945
946     FIN(dsr_ext_fill_yangs_eta_data(List* eta_star, int prio));
947
948     dsr_ext_fill_flow_reservation_data(rates);
949
950     //add the rates for this node
951     //iterate through the priority classes
952     //override prio with DSR_EXT_NUM_RATES to model ALL traffic
953     for(j=DSR_EXT_NUM_RATES; j>=0; j--)
954     {
955
956         //create a new element to add to the eta_data list
957         eta_star_elem_ptr = (Dsr_Ext_Yangs_Eta_Data*) op_prg_mem_alloc (sizeof(Dsr_Ext_Yangs_Eta_Data));
958         eta_star_elem_ptr->pk_size = dsr_ext_get_pk_size_from_priority(j);
959         eta_star_elem_ptr->cw_size = dsr_ext_get_cw_from_priority(j);
960         eta_star_elem_ptr->pk_rate = rates[j];
961
962         if(eta_star_elem_ptr->pk_rate <= 0.0)
963             eta_star_elem_ptr->pk_rate = 1e-6;
964
965         eta_star_elem_ptr->eta = CHANNEL_BANDWIDTH / (eta_star_elem_ptr->cw_size * eta_star_elem_ptr->pk_rate);
966
967         op_prg_list_insert (eta_star, eta_star_elem_ptr, OPC_LISTPOS_TAIL);
968     }
969
970     //add rates for neighboring nodes
971     count = op_prg_list_size(neighbor_list);
972     list_itr = prg_list_head_cell_get (neighbor_list);
973     for(i=0; i<count; i++)
974     {
975         // for each neighbor, add the virtual nodes within it to the eta_data list
976         neighbor_ptr = (Dsr_Ext_Neighbor_Data*) prg_list_cell_data_get(list_itr);
977         //calculate the bandwidth used by each flow, only if the values are valid
978         if( neighbor_ptr->interference_range == OPC_TRUE)
979         {
980             //iterate through the priority classes
981             //override prio with DSR_EXT_NUM_RATES to model ALL traffic
982             for(j=DSR_EXT_NUM_RATES; j>=0; j--)
983             {
984                 if(neighbor_ptr->rates[j] <= 0.0)
985                     continue;
986
987                 //create a new element to add to the eta_data list
988                 eta_star_elem_ptr = (Dsr_Ext_Yangs_Eta_Data*) op_prg_mem_alloc (sizeof(Dsr_Ext_Yangs_Eta_Data));
989                 eta_star_elem_ptr->pk_size = dsr_ext_get_pk_size_from_priority(j);
990                 eta_star_elem_ptr->cw_size = dsr_ext_get_cw_from_priority(j);
991                 eta_star_elem_ptr->pk_rate = neighbor_ptr->rates[j];
992
993                 eta_star_elem_ptr->eta = CHANNEL_BANDWIDTH / (eta_star_elem_ptr->cw_size * eta_star_elem_ptr->pk_rate);
994
995                 op_prg_list_insert (eta_star, eta_star_elem_ptr, OPC_LISTPOS_TAIL);
996             }
997         }
998         if( i < (count-1) )
999             list_itr = prg_list_cell_next_get(list_itr);
1000    }
1001    //sort the list by the eta term
1002    prg_list_sort (eta_star, dsr_ext_yangs_eta_data_sort_proc);
1003    FOUT;
1004 }
1005
1006 //computes local available bandwidth in accordance with algorithm 1 of Yang05
1007 //the priority parameter indicates the priority of the flow being requested, so as to
1008 //identify which 'virtual node' this computation will take place from
1009 static double dsr_ext_compute_yangs_local_available_bandwidth(int alpha, int priority)
1010 {
1011     int                         i,N;            //number of nodes in the interference neighborhood
1012     double                       Vf, V_star[2],X[2],Y[2], eta, result;
1013     List*                        eta_star; //list of eta_data pointers
1014     Dsr_Ext_Yangs_Eta_Data*     eta_star_elem_ptr;
1015
1016
1017     FIN(dsr_ext_compute_yangs_local_available_bandwidth(<args>));
1018     eta_star = op_prg_list_create();
```

```
1019 /*1,2*/
1020     dsr_ext_fill_yangs_eta_data( eta_star, priority);
1021     N = op_prg_list_size(eta_star);
1022     eta = 1e20;
1023
1024     //begin computing the local available bandwidth
1025 /*3*/Vf = alpha*dsr_ext_get_pk_size_from_priority(priority)/dsr_ext_get_cw_from_priority(priority);
1026 /*?*/V_star[0] = 0.0;
1027 /*4*/X[0] = 0.0;
1028     Y[0] = 0.0;
1029     for(i=0; i<N; i++)
1030     {
1031         eta_star_elem_ptr = (Dsr_Ext_Yangs_Eta_Data*) op_prg_list_access(eta_star, i);
1032 /*5*/     Y[0] += eta_star_elem_ptr->pk_rate * eta_star_elem_ptr->pk_size /CHANNEL_BANDWIDTH;
1033     }
1034
1035     //search for eta
1036     for(i=0; i<N; i++)
1037     {
1038         eta_star_elem_ptr = (Dsr_Ext_Yangs_Eta_Data*) op_prg_list_access(eta_star, i);
1039 /*6*/     X[1] = X[0] + (eta_star_elem_ptr->pk_size / eta_star_elem_ptr->cw_size);
1040 /*7*/     Y[1] = Y[0] - (eta_star_elem_ptr->pk_rate * eta_star_elem_ptr->pk_size / CHANNEL_BANDWIDTH);
1041 /*8*/     V_star[1] = eta_star_elem_ptr->eta * (1 - Y[1]) - X[1];
1042 /*9*/     if(V_star[0] <= Vf && Vf < V_star[1])
1043         {
1044 /*10*/        eta = (X[0] + Vf) / (1- Y[0]);
1045 //printf("*assigning eta:%e, X[0]:%e, Vf:%e, Y[0]:%e\n", eta, X[0], Vf, Y[0]);
1046             break;
1047         }
1048
1049         //shift the elements left
1050         X[0] = X[1];
1051         Y[0] = Y[1];
1052         V_star[0] = V_star[1];
1053     }
1054
1055 /*11*/result = (Vf * CHANNEL_BANDWIDTH) / (alpha * eta);
1056
1057     //clean up data structures used
1058     dsr_ext_list_mem_free(eta_star);
1059
1060     FRET(result);
1061 }
1062
1063 static double dsr_ext_compute_yangs_neighborhood_available_bandwidth(int alpha, int priority)
1064 {
1065     int                         i,N;            //number of nodes in the interference neighborhood
1066     double                        X, Y, eta_star_gamma, result;
1067     List*                         eta_star; //list of eta_data pointers
1068     Dsr_Ext_Yangs_Eta_Data*     eta_star_elem_ptr;
1069
1070     FIN(dsr_ext_compute_yangs_neighborhood_available_bandwidth(int alpha, int priority));
1071
1072     eta_star_gamma = 0.0;
1073     if(alpha < 1)//prevent div by zero exception
1074         alpha = 1;
1075
1076     eta_star = op_prg_list_create();
1077     dsr_ext_fill_yangs_eta_data(eta_star, priority);
1078     N = op_prg_list_size(eta_star);
1079
1080     //find the eta-star-gamma value, which is the smallets eta_star with
1081     // priority less important than or equal to current priority
1082     for(i=0; i<N; i++)
1083     {
1084         eta_star_elem_ptr = (Dsr_Ext_Yangs_Eta_Data*) op_prg_list_access(eta_star, i);
1085         if(eta_star_elem_ptr->cw_size <= dsr_ext_get_cw_from_priority(priority))
1086         {
1087             eta_star_gamma = eta_star_elem_ptr->eta;
1088             break;
1089         }
1090     }
1091
1092     //for all the nodes in the interference range with eta_star less than or equal to eta_star_gamma:
1093     //  X = sum the ratio of packet size to contention window
1094     //for all the nodes with eta_star greater than gamma:
1095     //  Y = sum the ratio of bandwidth to channel capacity
1096     X = 0.0; Y = 0.0;
1097     for(i=0; i<N; i++)
1098     {
1099         eta_star_elem_ptr = (Dsr_Ext_Yangs_Eta_Data*) op_prg_list_access(eta_star, i);
1100         if(eta_star_elem_ptr->eta <= eta_star_gamma)
1101             X += eta_star_elem_ptr->pk_size / eta_star_elem_ptr->cw_size;
1102         else
1103             Y += eta_star_elem_ptr->pk_rate * eta_star_elem_ptr->pk_size / CHANNEL_BANDWIDTH;
1104     }
1105
```

```
1106     //compute neighborhood available bandwidth
1107     result = CHANNEL_BANDWIDTH / alpha * ((1 - Y) - (X / eta_star_gamma));
1108     dsr_ext_list_mem_free(eta_star);
1109
1110     FRET(result);
1111 }
1112
1113 #endif //YANG_BANDWIDTH
1114

1115 #endif //DSR_EXT
```

## A.2 Code Modifications

This section details the various changes made to existing DSR code to implement QASR. All dsr_rte functions which contain modifications are listed in full.

```
001
002 static void
002 dsr_rte_received_pkt_handle (void)
003     {
004     Packet*                         ip_pkptr = OPC_NIL;
005     Packet*                            copy_pkptr = OPC_NIL;
006     Packet*                            return_pkptr = OPC_NIL;
007     IpT_Rte_Ind_Ici_Fields*         intf_ici_fdstruct_ptr = OPC_NIL;
008     IpT_Dgram_Fields*                 ip_dgram_fd_ptr = OPC_NIL;
009     Packet*                            dsr_pkptr = OPC_NIL;
010     List*                           tlv_options_lptr = OPC_NIL;
011     int                                num_options, count;
012     DsrT_Packet_Option*                dsr_tlv_ptr = OPC_NIL;
013     DsrT_Packet_Type                packet_type = DsrC_Undef_Packet;
014     char                            addr_str [INETC_ADDR_STR_LEN];
015     char                            node_name [OMSC_HNAME_MAX_LEN];
016     char                            temp_str [256];
017     Compcode                        status;
018     Boolean                            app_pkt_set = OPC_FALSE;
019
020     char                            pk_format [128];
021 #ifdef ENABLE_DSR_EXTENSIONS
022     Dsr_Ext_Connection*                connection_ptr;
023     Dsr_Ext_Route_Metrics*     dsr_metrics;
024     double delay, ratio;
025 #endif
026     /* A packet has arrived. Handle the packet    */
027     /* appropriately based on its various TLV    */
028     /* options set in the DSR header            */
029     FIN (dsr_rte_received_pkt_handle (void));
030
031     /* The process was invoked by the parent    */
032     /* MANET process indicating the arrival        */
033     /* of a packet. The packet can either be    */
034     /* 1. A higher layer application packet        */
035     /*    waiting to be transmitted when a         */
036     /*    route is found.                         */
037     /* 2. A MANET signaling/routing packet         */
038     /*    arrival which may or may not be a        */
039     /*    broadcast packet.                        */
040
041     /* Access the argument memory to get the    */
042     /* packet pointer.                         */
043     ip_pkptr = (Packet*) op_pro_argmem_access ();
044
045     if (ip_pkptr == OPC_NIL)
046         dsr_rte_error ("Could not obtain the packet from the argument memory", OPC_NIL, OPC_NIL);
047
048         /* Access the information from the incoming IP packet    */
049     manet_rte_ip_pkt_info_access (ip_pkptr, &ip_dgram_fd_ptr, &intf_ici_fdstruct_ptr);
050
051     /* Determine the packet type    */
052     packet_type = dsr_rte_packet_type_determine (ip_dgram_fd_ptr, intf_ici_fdstruct_ptr);
053
054     /* Check if this IP packet is carrying a DSR header    */
055     if (packet_type == DsrC_Higher_Layer_Packet)
056         {
057         if (LTRACE_ACTIVE)
058             {
059             inet_address_print (addr_str, ip_dgram_fd_ptr->dest_addr);
060             inet_address_to_hname (ip_dgram_fd_ptr->dest_addr, node_name);
061             sprintf (temp_str, "to destination %s (%s)", addr_str, node_name);
062             op_prg_odb_print_major (pid_string, "An application packet has arrived at this node", temp_str, OPC_NIL);
063             }
064         /* This IP datagram does not have a DSR header    */
065         /* It should be a higher layer packet           */
066         dsr_rte_app_pkt_arrival_handle (ip_pkptr, intf_ici_fdstruct_ptr, ip_dgram_fd_ptr, OPC_FALSE);
067
068         FOUT;
069         }
```

```
070
071     /* Get the DSR packet from the IP datagram    */
072     op_pk_nfd_get (ip_pkptr, "data", &dsr_pkptr);//using op_pk_nfd_access broke the world (null DSR packet error)
073
074     if(dsr_pkptr == 0)
075     {
076         dsr_rte_error("NULL DSR packet found in IP packet.",0,0);
077         FOUT;
078     }
079
080 #ifdef ENABLE_DSR_EXTENSIONS
081     op_pk_format (dsr_pkptr, pk_format);
082
083 #ifdef ENABLE_DSR_YANG_BANDWIDTH
084     else if(strcmp (pk_format, "dsr_ext_flow_state_packet") == 0 )
085     {
086         Boolean rebroadcast;
087         double prob;
088         if (manet_rte_address_belongs_to_node (module_data_ptr, ip_dgram_fd_ptr->dest_addr) == OPC_TRUE ||
089             manet_rte_address_belongs_to_node (module_data_ptr, ip_dgram_fd_ptr->src_addr)  == OPC_TRUE )
090         {
091             //print_local_ip(); printf(" recieved a response packet\r\n");
092             op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
093             manet_rte_ip_pkt_destroy (ip_pkptr);
094         }
095         else
096         {
097             Dsr_Ext_Rate_Reservations* info_ptr;
098             op_pk_nfd_access(dsr_pkptr, "flow_data",&info_ptr);
099             if(dsr_ext_is_interference_neighbor( info_ptr->loc, DSR_EXT_INTERFERENCE_RANGE) == OPC_FALSE)
100             {
101                 op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
102                 manet_rte_ip_pkt_destroy (ip_pkptr);
103                 FOUT;
104             }
105
106             //limit protocol bandwidth to a constant overhead cost
107             prob = (double)dsr_ext_num_transmitters_in_interference_range();
108             prob = DSR_EXT_OVERHEAD_CONST / prob;
109             if( prob > 1.0 || (rand()/(double)RAND_MAX) < prob)
110                 rebroadcast = OPC_TRUE;
110             else
111                 rebroadcast = OPC_FALSE;
112
113             //call the store_interference_neighbor_data function, if succesfull, retransmit, otherwise, destroy the packet
114             if(dsr_ext_store_neighbor_flow_state(ip_dgram_fd_ptr->src_addr, info_ptr) == OPC_TRUE && rebroadcast == OPC_TRUE)
115             {
116                 op_stat_write (broadcast_overhead_pkts_shandle, 1.0);
117                 op_stat_write (broadcast_overhead_bits_shandle, op_pk_total_size_get (dsr_pkptr));
118
119                 op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
120                 dsr_rte_jitter_schedule (ip_pkptr, DSRC_ROUTE_REQUEST);
121             }
122             else
123             {
124                 op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
125                 manet_rte_ip_pkt_destroy (ip_pkptr);
126                 FOUT;
127             }
128         }
129         FOUT;
130     }
131 #endif //Yang
132 #else
133     //It's not likely that flow_state packets will arrive if QASR features are not compiled,
134     //but just in case, we delete them before they can upset the DSR code
135     op_pk_format (dsr_pkptr, pk_format);
136     if (strcmp (pk_format, "dsr_ext_flow_state_packet") == 0 )
137         {
138         manet_rte_ip_pkt_destroy (ip_pkptr);
139         FOUT;
140         }
141 #endif
142
143     /* This packet is received from the MAC layer    */
144     /* Update the statistic for the total traffic    */
145     dsr_support_total_traffic_received_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
146
147     /* Check if this is an application packet    */
148     /* or just a DSR routing packet                */
149     app_pkt_set = op_pk_nfd_is_set (dsr_pkptr, "data");
150
151     /* Get the list of options    */
152     op_pk_nfd_access (dsr_pkptr, "Options", &tlv_options_lptr);
153
154     /* Set the DSR packet back into the IP datagram    */
155     op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
```

```
154
155     if (app_pkt_set == OPC_FALSE)
156         {
157         /* This is a DSR routing packet. Update    */
158         /* the statistic for routing traffic    */
159         dsr_support_routing_traffic_received_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
160         }
161     else
162         {
163         /* This is an application packet. Decrease the TTL     */
164         /* if i am not the destination node                */
165         if (manet_rte_address_belongs_to_node (module_data_ptr, ip_dgram_fd_ptr->dest_addr) == OPC_FALSE)
166             ip_dgram_fd_ptr->ttl--;
167         }
168
169         /* Get the number of options    */
170     num_options = op_prg_list_size (tlv_options_lptr);
171     for (count = 0; count < num_options; count++)
172         {
173         /* Make a copy of the incoming packet for each option    */
174         copy_pkptr = manet_rte_ip_pkt_copy (ip_pkptr);
175
176         /* Get the DSR packet from the IP datagram    */
177         op_pk_nfd_get (copy_pkptr, "data", &dsr_pkptr);
178
179         /* Get the list of options    */
180         op_pk_nfd_access (dsr_pkptr, "Options", &tlv_options_lptr);
181
182
183         /* Set the DSR packet into the IP datagram    */
184         op_pk_nfd_set (copy_pkptr, "data", dsr_pkptr);
185
186         /* Get each option    */
187         dsr_tlv_ptr = (DsrT_Packet_Option*) op_prg_list_access (tlv_options_lptr, count);
188         /* Process the option based on the type    */
189         switch (dsr_tlv_ptr->option_type)
190             {
191             case (DSRC_ROUTE_REQUEST):
192                 {
193                 /* The packet contains a route request option    */
194                 /* Insert this node into the route request list    */
195                 dsr_pkt_support_route_request_hop_insert (copy_pkptr, intf_ici_fdstruct_ptr->interface_received);
196
197 #ifndef ENABLE_DSR_EXTENSIONS
198                 /* Insert the route in the route cache based on            */
199                 /* the requirement for caching overheard information    */
200                 dsr_rte_route_cache_update (dsr_tlv_ptr, ip_dgram_fd_ptr);
201 #endif
202
203                 /* After possibly inserting the route in the route cache    */
204                 /* process the received route request option                */
205                 dsr_rte_received_route_request_process (copy_pkptr, dsr_tlv_ptr);
206                 break;
207                 }
208
209             case (DSRC_ROUTE_REPLY):
210                 {
211                 /* The packet contains a route reply option    */
212                 /* Insert the route in the route cache only if this node is the source node    */
213                 if ( manet_rte_address_belongs_to_node (module_data_ptr, ip_dgram_fd_ptr->dest_addr) == OPC_TRUE)
214                 {
215 #ifdef ENABLE_DSR_YANG_BANDWIDTH
216                     int flow_id;
217
218                     op_pk_nfd_get (copy_pkptr, "data", &dsr_pkptr);
219
220                     op_pk_nfd_get (dsr_pkptr, "flow_id", &flow_id);
221                     op_pk_nfd_set (dsr_pkptr, "flow_id", flow_id);
222
223                     //the get-set pair needs to be used here, not access,
224                     //in order to trigger a copy from the original packet
225                     op_pk_nfd_get (dsr_pkptr, "Metrics", &dsr_metrics);
226                     op_pk_nfd_set_ptr(dsr_pkptr, "Metrics", dsr_metrics, op_prg_mem_copy_create,
227                                       op_prg_mem_free, sizeof (Dsr_Ext_Route_Metrics));
228
229                     op_pk_nfd_set (copy_pkptr, "data", dsr_pkptr);
230
231                     //aquire a connection structure, creating a new one if nessesary
232                     connection_ptr = dsr_ext_get_connection_ptr(ip_dgram_fd_ptr->src_addr, flow_id, DSRC_ROUTE_REQUEST);
233
234                     //add this copy of the ip packet to the list
235                     op_prg_list_insert (connection_ptr->source_pk_list_ptr, copy_pkptr, OPC_LISTPOS_TAIL);
236                     if(    connection_ptr->reply_recieved == OPC_FALSE)
237                     {
238                         //schedule the route reply for after the route accept window closes
                        op_intrpt_schedule_call (op_sim_time() + 0.005 , DSRC_ROUTE_REPLY, dsr_ext_route_accept_window_expiry,
(void*)connection_ptr);
```

```
239                         connection_ptr->reply_recieved = OPC_TRUE;
240                     }
241 #else
242                     //print the addresses of the hops in the route
243                     print_route( dsr_tlv_ptr);
244 
245                     /* store the route reply */
246                     dsr_rte_route_cache_update (dsr_tlv_ptr, ip_dgram_fd_ptr);
247 #endif
248                 }
249                 /* After possibly inserting the route in the route cache    */
250                 /* process the received route reply option                  */
251                 dsr_rte_received_route_reply_process (copy_pkptr, dsr_tlv_ptr);
252 
253                 break;
254                 }
255 
256             case (DSRC_ROUTE_ERROR):
257                 {
258                 /* The packet contains a route error option    */
259                 /* Process the received route error            */
260                 dsr_rte_received_route_error_process (copy_pkptr, dsr_tlv_ptr);
261                 break;
262                 }
263 
264             case (DSRC_ACK_REQUEST):
265                 {
266                 /* The packet contains an acknowledgement    */
267                 /* request option. Process the option        */
268                 status = dsr_rte_received_ack_request_process (copy_pkptr);
269 
270                 if (status == OPC_COMPCODE_SUCCESS)
271                     {
272                     /* An acknowledgement was sent out for the    */
273                     /* received acknowledgement request. Remove    */
274                     /* the acknowledgement request option from    */
275                     /* the packet received                        */
276                     op_pk_nfd_get (ip_pkptr, "data", &dsr_pkptr);
277                     dsr_pkt_support_option_remove (dsr_pkptr, DSRC_ACK_REQUEST);
278                     op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
279                     num_options--;
280                     count--;
281                     }
282 
283                 break;
284                 }
285 
286             case (DSRC_ACKNOWLEDGEMENT):
287                 {
288                 /* The packet contains an acknowledgement option    */
289                 /* The node should add to its route cache the        */
290                 /* single link from the node identified by the ACK    */
291                 /* source address to the node identified by the ACK    */
292                 /* destination address.                               */
293 #ifndef ENABLE_DSR_EXTENSIONS
294                 /* Insert the route in the route cache based on            */
295                 /* the requirement for caching overheard information    */
296                 dsr_rte_route_cache_update (dsr_tlv_ptr, ip_dgram_fd_ptr);
297 #endif
298                 /* After possibly inserting the route in the route cache    */
299                 /* process the received acknowledgement option               */
300                 dsr_rte_received_acknowledgement_option_process (copy_pkptr, dsr_tlv_ptr);
301 
302                 break;
303                 }
304 
305             case (DSRC_SOURCE_ROUTE):
306                 {
307                 /* The packet contains a DSR source route option    */
308 #ifndef ENABLE_DSR_EXTENSIONS //passive route cacheing is disabled entirely in QASR
309                 /* Insert the route in the route cache based on            */
310                 /* the requirement for caching overheard information    */
311                 dsr_rte_route_cache_update (dsr_tlv_ptr, ip_dgram_fd_ptr);
312 #endif
313                 /* After possibly inserting the route in the route cache    */
314                 /* process the received DSR source route option              */
315                 dsr_rte_received_dsr_source_route_option_process (copy_pkptr);
316                 break;
317                 }
318 
319             default:
320                 {
321                 /* Invalid option in packet    */
322                 dsr_rte_error ("Invalid Option Type in DSR packet", OPC_NIL, OPC_NIL);
323                 }
324             }
325         }
```

```
326
327     /* If the destination address in the IP packet        */
328     /* matches one of the receiving node's own IP        */
329     /* addresses and this is a application packet,         */
330     /* remove the DSR header and all DSR options and     */
331     /* pass the packet to the higher layer                */
332     if (manet_rte_address_belongs_to_node (module_data_ptr, ip_dgram_fd_ptr->dest_addr) == OPC_TRUE)
333         {
334 #ifdef ENABLE_DSR_YANG_DELAY
335         int flow_id;
336         op_pk_nfd_get (ip_pkptr, "data", &dsr_pkptr);//using op_pk_nfd_access broke the world (null DSR packet error)
337         op_pk_nfd_get(dsr_pkptr, "flow_id", &flow_id);
338         op_pk_nfd_set(dsr_pkptr, "flow_id", flow_id);
339         op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
340
341         delay = op_sim_time () - op_pk_creation_time_get (ip_pkptr);
342
343         if((int)flow_id >= 0 && flow_id < 2000 && global_delay_estimate[flow_id])
344         {
345             ratio = delay / global_delay_estimate[flow_id];
346
347             op_stat_write (delay_ratio_metric_shandle, ratio);
348         }
349 #endif
350
351
352         /* Decapsulate the DSR packet    */
353         return_pkptr = dsr_rte_ip_datagram_decapsulate (ip_pkptr);
354
355         if (return_pkptr != OPC_NIL)
356             {
357             /* Send the IP packet to the higher layer    */
358             manet_rte_to_higher_layer_pkt_send_schedule (module_data_ptr, parent_prohandle, return_pkptr);
359             }
360         else
361             {
362             /* Destroy the packet    */
363             manet_rte_ip_pkt_destroy (ip_pkptr);
364             }
365         }
366     else
367         {
368         /* Destroy the packet    */
369         manet_rte_ip_pkt_destroy (ip_pkptr);
370         }
371     FOUT;
372     }
373
374
375 static void
376 dsr_rte_app_pkt_arrival_handle (Packet* ip_pkptr, IpT_Rte_Ind_Ici_Fields* intf_ici_fdstruct_ptr,
377                             IpT_Dgram_Fields* ip_dgram_fd_ptr, Boolean discovery_performed)
378     {
379     DsrT_Path_Info*            path_ptr = OPC_NIL;
380     InetT_Address*            next_hop_addr_ptr;
381     DsrT_Packet_Option*        dsr_tlv_ptr = OPC_NIL;
382     Packet*                    dsr_pkptr = OPC_NIL;
383     Packet*                    qos_pkptr = OPC_NIL;
384     Boolean                    maint_req_added = OPC_FALSE;
385     Boolean                    source_route_added = OPC_FALSE;
386     List*                     temp_lptr;
387     char                      dest_node_name [OMSC_HNAME_MAX_LEN];
388     char                      dest_hop_addr_str [INETC_ADDR_STR_LEN];
389     char                      temp_str [2048];
390     char*                      route_str;
391     InetT_Address*            copy_address_ptr;
392     int                        num_nodes;
393     int flow_id = 0;
394
395
396 #ifdef ENABLE_DSR_QOS_EXTENSIONS
397     Dsr_Ext_Route_Metrics*     dsr_metrics;
398     double    pk_rate, delay_req, bandwidth_required;
399     int        pk_size;
400     int        up = ip_dgram_fd_ptr->tos>>5;
401
402 #endif
403     /* An application packet needs to be sent to     */
404     /* its destination via the DSR network.         */
405     /* Process the packet                            */
406     FIN (dsr_rte_app_pkt_arrival_handle (<args>));
407
408 #ifdef ENABLE_DSR_QOS_EXTENSIONS
409     op_pk_nfd_get(ip_pkptr, "data", &qos_pkptr);
410     op_pk_format (qos_pkptr, temp_str);
411     if (strcmp (temp_str, "manet_qos_packet") == 0)
412     {
```

```
410         op_pk_nfd_get(qos_pkptr, "flow_id", &flow_id);
411         op_pk_nfd_get(qos_pkptr, "pk_rate", &pk_rate);
412         op_pk_nfd_get(qos_pkptr, "pk_size", &pk_size);
413         op_pk_nfd_get(qos_pkptr, "delay_req", &delay_req);
414         bandwidth_required = pk_rate * pk_size;
415     }
416     op_pk_nfd_set(ip_pkptr, "data", qos_pkptr);
417 #endif
418 
419     /* Section 6.1.1    */
420     /* Determine if there is a route to the destination    */
421     /* of the packet in the route cache                    */
422     path_ptr = dsr_route_cache_entry_access (route_cache_ptr, ip_dgram_fd_ptr->dest_addr, discovery_performed);
423     if (path_ptr != OPC_NIL)
424         {
425         if (LTRACE_ACTIVE)
426             {
427             route_str = dsr_support_route_print (path_ptr->path_hops_lptr);
428             op_prg_odb_print_major ("Found a route to the destination", route_str, OPC_NIL);
429             op_prg_mem_free (route_str);
430             }
431 
432         /* A route exists to the destination node in    */
433         /* this node's route cache. Get the next hop    */
434         next_hop_addr_ptr = (InetT_Address*) op_prg_list_access (path_ptr->path_hops_lptr, 1);
435 
436         /* There is no maintenance scheduled for the next    */
437         /* hop. Check if maintenance is needed against the    */
438         /* maintenance holdoff time                            */
439         if (dsr_maintenance_buffer_maint_needed (maint_buffer_ptr, *next_hop_addr_ptr) == OPC_TRUE)
440             {
441             if (LTRACE_ACTIVE)
442                 {
443                 inet_address_print (dest_hop_addr_str, *next_hop_addr_ptr);
444                 inet_address_to_hname (*next_hop_addr_ptr, dest_node_name);
445                 sprintf (temp_str, "to next hop node %s (%s) with ID (%d)", dest_hop_addr_str, dest_node_name,
ack_request_identifier);
446                 op_prg_odb_print_major ("Adding a maintenance request option in packet", temp_str, OPC_NIL);
447                 }
448 
449             /* Create a IP datagram with a maintenance    */
450             /* request option in the DSR header            */
451             dsr_tlv_ptr = dsr_pkt_support_ack_request_tlv_create (ack_request_identifier);
452 
453             /* Create the DSR packet    */
454             dsr_pkptr = dsr_pkt_support_pkt_create (ip_dgram_fd_ptr->protocol);
455 
456             /* Set the maintenance request option in the DSR packet header    */
457             dsr_pkt_support_option_add (dsr_pkptr, dsr_tlv_ptr);
458 
459             /* Update the statistic for the number of maintenance requests sent    */
460             dsr_support_maintenace_stats_update (stat_handle_ptr, global_stathandle_ptr, OPC_TRUE);
461 
462             /* Set the flag to indicate that a maintenance request    */
463             /* option has been added to the DSR header                */
464             maint_req_added = OPC_TRUE;
465             }
466 
467         /* If the next hop address is the destination    */
468         /* ,ie, only one hop to the destination, then    */
469         /* send the packet out directly without adding    */
470         /* a source route option to the IP datagram        */
471 #ifdef ENABLE_DSR_EXTENSIONS
472         if(1)//never cheat by not sending a dsr header, it breaks things
473 #else
474         if (inet_address_equal (ip_dgram_fd_ptr->dest_addr, *next_hop_addr_ptr) == OPC_FALSE)
475 #endif
476             {
477             /* The next hop is not the destination, ie,    */
478             /* there is more than one hop to reach the    */
479             /* destination. Add a source route option    */
480             /* along with the DSR header to the IP        */
481             /* datagram and then send out the packet    */
482             dsr_tlv_ptr = dsr_pkt_support_source_route_tlv_create (path_ptr->path_hops_lptr, path_ptr->first_hop_external,
483                                                 path_ptr->last_hop_external, routes_export, OPC_FALSE);
484 
485             if (maint_req_added == OPC_FALSE)
486                 {
487                 /* Create the DSR packet if not already created    */
488                 dsr_pkptr = dsr_pkt_support_pkt_create (ip_dgram_fd_ptr->protocol);
489                 }
490 
491             /* Set the source route option in the DSR packet header    */
492             dsr_pkt_support_option_add (dsr_pkptr, dsr_tlv_ptr);
493 
494             /* Set the flag to indicate that a source route option    */
495             /* has been added to the DSR header                        */
```

```
496             source_route_added = OPC_TRUE;
497             }
498         else
498             {
499
500             if (routes_export)
501                 {
502                 /* Print the single hop route    */
503                 temp_lptr = op_prg_list_create ();
504                 dsr_support_route_print_to_ot (ip_dgram_fd_ptr, temp_lptr);
505                 dsr_temp_list_clear (temp_lptr);
506                 }
507             if (routes_dump)
508                 {
509                 if (inet_address_equal (ip_dgram_fd_ptr->src_addr, INETC_ADDRESS_INVALID) == OPC_FALSE)
510                     {
511                     /* Print the single hop route    */
512                     temp_lptr = op_prg_list_create ();
513
514                     /* Read the source node    */
515                     copy_address_ptr = inet_address_create_dynamic (ip_dgram_fd_ptr->src_addr);
516                     op_prg_list_insert (temp_lptr, copy_address_ptr, OPC_LISTPOS_TAIL);
517
518                     /* Read the destination node    */
519                     copy_address_ptr = inet_address_create_dynamic (ip_dgram_fd_ptr->dest_addr);
520                     op_prg_list_insert (temp_lptr, copy_address_ptr, OPC_LISTPOS_TAIL);
521
522                     /* Dump the route    */
523                     manet_rte_path_display_dump (temp_lptr);
524
525                     /* Free the contents of the list    */
526                     num_nodes = op_prg_list_size (temp_lptr);
527
528                     while (num_nodes > 0)
529                         {
530                         copy_address_ptr = (InetT_Address*) op_prg_list_remove (temp_lptr, OPC_LISTPOS_HEAD);
531                         inet_address_destroy_dynamic (copy_address_ptr);
532                         num_nodes--;
533                         }
534
535                     /* Free the list    */
536                     dsr_temp_list_clear (temp_lptr);
537                     }
538                 }
539             }
540
541         if ((maint_req_added == OPC_TRUE) || (source_route_added == OPC_TRUE))
542             {
543 #ifdef ENABLE_DSR_EXTENSIONS
544             dsr_metrics = dsr_ext_create_route_request_metrics(ip_dgram_fd_ptr->dest_addr, flow_id);
545             op_pk_nfd_set_ptr(dsr_pkptr, "Metrics", dsr_metrics, op_prg_mem_copy_create,
546                 op_prg_mem_free, sizeof (Dsr_Ext_Route_Metrics));
547             op_pk_nfd_set(dsr_pkptr, "flow_id", flow_id);
548 #endif
549             /* Encapsulate the DSR packet in the received IP datagram    */
550             dsr_rte_ip_datagram_encapsulate (ip_pkptr, dsr_pkptr, *next_hop_addr_ptr);
551             }
552
553         if (maint_req_added == OPC_TRUE)
554             {
555             /* A maintenance request has been added    */
556             /* Place a copy of the packet in the    */
557             /* maintenance buffer for retranmission    */
558             dsr_maintenance_buffer_pkt_enqueue (maint_buffer_ptr, ip_pkptr, *next_hop_addr_ptr, ack_request_identifier);
559
560             /* Increment the ACK Request identifier    */
561             ack_request_identifier++;
562             }
563 #ifdef ENABLE_DSR_YANG_BANDWIDTH
564         //update the flow reservations only for packets routed (transmitted) by this node
565         //this is a higher layer packet going out, so it must be transmitted by this node
566         dsr_ext_update_flow_reservation(ip_pkptr);
567 #endif
568
569
570         /* Update the statistic for the total traffic sent    */
571         dsr_support_total_traffic_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
572
573         //Send part of the packet deliver ratio gets recorded here
574         op_stat_write(total_data_sent_shandle, 1.0);
575
576         /* Send the packet out to the MAC    */
577         manet_rte_to_mac_pkt_send (module_data_ptr, ip_pkptr, *next_hop_addr_ptr, ip_dgram_fd_ptr, intf_ici_fdstruct_ptr);
578         }
579     else
579         {
580         /* No route exists to the destination    */
```

```
581         /* Perform route discovery by             */
582         /* originating a route request as in    */
583         /* section 6.2.1                        */
584         if (LTRACE_ACTIVE)
585             {
586             op_prg_odb_print_major ("No route exists to destination", "Perform route discovery", OPC_NIL);
587             }
588 
589         /* Do not originate the route request    */
590         /* if there is already a request sent     */
591         /* to the same destination, simply         */
592         /* enqueue the packet in send buffer.     */
593 
594         /* Convert dest_addr to string */
595         inet_address_print (temp_str, ip_dgram_fd_ptr->dest_addr);
596 
597         /* Check if there is no route discovery already in process     */
598         /* Start one by sending Route request                        */
599         if (prg_string_hash_table_item_get (route_request_table_ptr->route_request_send_table, temp_str) == OPC_NIL)
600         {
601 #ifdef ENABLE_DSR_QOS_EXTENSIONS
602             //get and fill a connection pointer with QoS requirements from the MANET packet, if it exists...
603             if(qos_pkptr != OPC_NIL)
604             {
605                 op_pk_format (qos_pkptr, temp_str);
606                 if (strcmp (temp_str, "manet_qos_packet") == 0)
607                 {
608                     //run path_admit before starting a discovery
609                     Dsr_Ext_Connection* connection_ptr;
610 
611                     //the normal action would be to call dsr_ext_path_admit, but the  packet is incomplete at this point
612                     //therefore the flow acceptance is hacked in right here
613                     //if(dsr_ext_path_admit( ip_pkptr, OPC_NIL, OPC_TRUE) == OPC_FALSE)
614                     if(up < 1 || up >2)//ignore background trafic
615                     {
616                         float local_available_bandwidth, neighborhood_available_banwidth;
617                         local_available_bandwidth = dsr_ext_compute_yangs_local_available_bandwidth(1,
dsr_ext_map_ip_to_mac_priority(up));
618                         neighborhood_available_banwidth = dsr_ext_compute_yangs_neighborhood_available_bandwidth(1,
dsr_ext_map_ip_to_mac_priority(up));
619                         if(    bandwidth_required  > local_available_bandwidth ||
620                             bandwidth_required > neighborhood_available_banwidth)
621                         {
622                             manet_rte_ip_pkt_destroy (ip_pkptr);
623                             FOUT;
624                         }
625                     }
626                     connection_ptr = dsr_ext_get_connection_ptr(ip_dgram_fd_ptr->dest_addr, flow_id,DSRC_ROUTE_REQUEST);
627                     if(connection_ptr->is_new_connection == OPC_TRUE)
628                     {
629                         //printf("%f, ", op_sim_time());print_local_ip();
630                         //printf(": flow:%d qos:%d rate=%5.3f, size=%d, delay=%f\n",  flow_id, up, pk_rate, pk_size, delay_req);
631 
632                         total_routes_requested += 1.0; //used within this module to establish when the network is saturated
633                         op_stat_write (total_routes_requested_shandle, 1.0);
634                         connection_ptr->is_new_connection = OPC_FALSE;
635                     }
636                     connection_ptr->tos = ip_dgram_fd_ptr->tos;
637                     connection_ptr->pk_rate = pk_rate;
638                     connection_ptr->pk_size = pk_size;
639                     connection_ptr->delay_req = delay_req;
640                 }
641                 else
642                     printf("no qos requirments\n");
643             }
644             else
645             {
646                 printf("%f, ");printf(" null QoS packet on line:%d\n", op_sim_time(), __LINE__-314);
647             }
648 #endif
649             dsr_rte_route_request_send (ip_dgram_fd_ptr->dest_addr, non_propagating_request_function, ip_dgram_fd_ptr->tos,
flow_id);
650         }
651 
652         /* Place the packet in the send buffer    */
653         dsr_send_buffer_packet_enqueue (send_buffer_ptr, ip_pkptr, ip_dgram_fd_ptr->dest_addr);
654         }
655 
656     FOUT;
657     }
658 
659 
659 static void
660     dsr_rte_received_route_request_process (Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr)
661     {
662 
663     DsrT_Route_Request_Option*        route_request_option_ptr = OPC_NIL;
```

```
663     IpT_Dgram_Fields*                ip_dgram_fd_ptr = OPC_NIL;
664     IpT_Rte_Ind_Ici_Fields*         intf_ici_fdstruct_ptr = OPC_NIL;
665     int                                num_hops, count;
666     InetT_Address*                   hop_address_ptr = OPC_NIL;
667
668     DsrT_Path_Info*                    path_ptr = OPC_NIL;
669     char                            src_node_name [OMSC_HNAME_MAX_LEN];
670     char                            src_hop_addr_str [INETC_ADDR_STR_LEN];
671     char                            dest_node_name [OMSC_HNAME_MAX_LEN];
672     char                            dest_hop_addr_str [INETC_ADDR_STR_LEN];
673     char                            temp_str [2048];
674     char*                            route_str;
675 #ifdef ENABLE_DSR_QOS_EXTENSIONS
676     int                             flow_id;
677     double                             result;
678     Packet*                            dsr_pkptr;
679     Dsr_Ext_Connection*                connection_ptr;
680     Dsr_Ext_Route_Metrics*             dsr_metrics;
681 #endif
682
683 #ifdef ENABLE_DSR_YANG_DELAY
684     double                           delay_est;
685 #endif
686
687     /* Process the received route request option    */
688     /* Section 6.2.2 of the draft                    */
689     FIN (dsr_rte_received_route_request_process (<args>));
690 #ifdef ENABLE_DSR_QOS_EXTENSIONS
691     //extract the flow ide from this route request
692     op_pk_nfd_get(ip_pkptr, "data", &dsr_pkptr);
693     op_pk_nfd_access(dsr_pkptr,"Metrics",&dsr_metrics);
694     op_pk_nfd_get(dsr_pkptr,"flow_id", &flow_id);
695     op_pk_nfd_set(ip_pkptr, "data", dsr_pkptr);
696 #endif
697     /* Access the information from the incoming IP packet    */
698     manet_rte_ip_pkt_info_access (ip_pkptr, &ip_dgram_fd_ptr, &intf_ici_fdstruct_ptr);
699
700     /* Get the route request option    */
701     route_request_option_ptr = (DsrT_Route_Request_Option*) dsr_tlv_ptr->dsr_option_ptr;
702
703     if (LTRACE_ACTIVE)
704         {
705         route_str = dsr_support_option_route_print (dsr_tlv_ptr);
706         inet_address_print (src_hop_addr_str, ip_dgram_fd_ptr->src_addr);
707         inet_address_to_hname (ip_dgram_fd_ptr->src_addr, src_node_name);
708         inet_address_print (dest_hop_addr_str, route_request_option_ptr->target_address);
709         inet_address_to_hname (route_request_option_ptr->target_address, dest_node_name);
710         sprintf (temp_str, "from node %s (%s) destined to node %s (%s) with route",
711             src_hop_addr_str, src_node_name, dest_hop_addr_str, dest_node_name);
712         op_prg_odb_print_major ("Received a route request option in packet", temp_str, route_str, OPC_NIL);
713         op_prg_mem_free (route_str);
714         }
715
716     /* If the source address of the IP datagram        */
717     /* belongs to this node, then discard the IP    */
718     /* datagram as it has received its own packet    */
719     if (manet_rte_address_belongs_to_node (module_data_ptr, ip_dgram_fd_ptr->src_addr) == OPC_TRUE)
720         {
721         if (LTRACE_ACTIVE)
722             {
723             op_prg_odb_print_major ("Destroying the route request packet",
724                 "as the source node received its own packet", OPC_NIL);
725             }
726
727         /* The originator of the route request has    */
728         /* received its own packet again. Discard    */
729         /* ths IP datagram                            */
730         manet_rte_ip_pkt_destroy (ip_pkptr);
731
732         FOUT;
733         }
734
735     /* If the node's own IP address appears in the    */
736     /* list of recorded addresses, then discard the    */
737     /* entire packet. Get the list of addresses        */
738     num_hops = op_prg_list_size (route_request_option_ptr->route_lptr);
739     for (count = 0; count < (num_hops - 1); count++)
740         {
741         /* Access each hop and check if it belongs    */
742         /* to this node.                            */
743         hop_address_ptr = (InetT_Address*) op_prg_list_access (route_request_option_ptr->route_lptr, count);
744
745         /* Check if the hop belongs to the node        */
746         if (manet_rte_address_belongs_to_node (module_data_ptr, *hop_address_ptr) == OPC_TRUE)
747             {
748             if (LTRACE_ACTIVE)
749                 {
```

```
750                 op_prg_odb_print_major ("Destroying the route request packet",
751                     "as the node's own IP address appears in the list of recorded addresses", OPC_NIL);
752                 }
753
754             /* The hop does belong to the node    */
755             /* Destroy the IP packet           */
756             manet_rte_ip_pkt_destroy (ip_pkptr);
757
758             FOUT;
759             }
760         }
761
762     /* If the target address of the route request    */
763     /* matches one of the node's own IP addresses    */
764     /* then the node should return a route reply    */
765     /* to the initiator of this route request        */
766     if (manet_rte_address_belongs_to_node (module_data_ptr, route_request_option_ptr->target_address) == OPC_TRUE)
767         {
768         /* This node is the target of the route        */
769         /* request. Send a route reply                */
770
771 #ifdef ENABLE_DSR_YANG_DELAY
772         //TODO:: Validate end-to-end delay estimate against requirment
773         delay_est = dsr_ext_compute_yangs_delay(ip_pkptr, dsr_tlv_ptr);
774         dsr_metrics->delay_est += delay_est;
775 #endif
776
777 #ifdef ENABLE_DSR_QOS_EXTENSIONS
778
779 //print_local_ip();printf(" recieved flow request:%d from ", flow_id);ip_print(ip_dgram_fd_ptr->src_addr);printf("\n");
780         // store route requests from a particular source for a short window.
781         //When the window expires, a route reply should be sent using the best route found.
782         connection_ptr = dsr_ext_get_connection_ptr(ip_dgram_fd_ptr->src_addr, flow_id, DSRC_ROUTE_REPLY);
783         if(    connection_ptr->is_new_connection == OPC_TRUE)
784         {
785             //schedule the route reply for after the route accept window closes
786             op_intrpt_schedule_call (op_sim_time() + DSR_EXT_ACCEPT_WINDOW , DSRC_ROUTE_REQUEST, dsr_ext_route_accept_window_expiry,
(void*)connection_ptr);
787             connection_ptr->is_new_connection = OPC_FALSE;
788         }
789
790         op_prg_list_insert (connection_ptr->source_pk_list_ptr, ip_pkptr, OPC_LISTPOS_TAIL);
791 #else
792         //normal DSR sends a reply immediatly
793         dsr_rte_route_reply_send (ip_pkptr, dsr_tlv_ptr);
794         /* Destroy the IP packet            */
795         manet_rte_ip_pkt_destroy (ip_pkptr);
796 #endif
797         FOUT;
798         }
799
800
801 #ifdef ENABLE_DSR_QOS_EXTENSIONS
802     //Successive route requests may have an improved metric evaluation, if they do not, drop them
803     op_pk_nfd_get(ip_pkptr, "data", &dsr_pkptr);
804
805     //use the last hop address, which is hidding in the source route, or use the source address if there is none
806     if(hop_address_ptr == OPC_NIL)
807         result = dsr_ext_update_route_request_metrics(ip_dgram_fd_ptr->src_addr, dsr_pkptr, ip_dgram_fd_ptr->tos>>5);
808     else
809         result = dsr_ext_update_route_request_metrics(*hop_address_ptr, dsr_pkptr, ip_dgram_fd_ptr->tos>>5);
810
811     op_pk_nfd_set(ip_pkptr, "data", dsr_pkptr);
812
813     //distrubuted filter for inferior paths during search
814     if(dsr_ext_is_better_route(ip_dgram_fd_ptr->src_addr, route_request_option_ptr->target_address, flow_id,result) == OPC_FALSE)
815     {
816         manet_rte_ip_pkt_destroy (ip_pkptr);
817         FOUT;
818     }
819
820     //this is where partial path admission control is done
821     if(dsr_ext_path_admit(ip_pkptr, dsr_tlv_ptr, OPC_TRUE) == OPC_FALSE)
822     {
823         manet_rte_ip_pkt_destroy (ip_pkptr);
824         FOUT;
825     }
826 #else
827
828     /* Search the route request table for an     */
829     /* entry from the initiator of this route    */
830     /* request with the same identification        */
831     if (dsr_route_request_forwarding_table_entry_exists (route_request_table_ptr, ip_dgram_fd_ptr->src_addr,
832                                                 route_request_option_ptr->identification) == OPC_TRUE)
833         {
834         if (LTRACE_ACTIVE)
835             {
```

```
835             op_prg_odb_print_major ("Destroying the route request packet",
836                 "as an entry already exists in the route request table for this identification value", OPC_NIL);
837             }
838
839         /* An entry already exists in the route    */
840         /* request table for this originating    */
841         /* node and the identification value    */
842         /* Destroy the IP datagram                */
843         manet_rte_ip_pkt_destroy (ip_pkptr);
844         FOUT;
845         }
846 #endif
847
848     /* Check the TTL field of the IP datagram    */
849     if ((ip_dgram_fd_ptr->ttl - 1) == 0)
850         {
851         /* This may be either a non-propagating    */
852         /* request that was set to one hop, or     */
853         /* the TTL field value of the packet     */
854         /* has reached the maximum number        */
855         if (LTRACE_ACTIVE)
856             {
857             op_prg_odb_print_major ("Destroying the route request packet",
858                 "as the TTL value of the IP datagram is 0", OPC_NIL);
859             }
860
861         /* Destroy the IP datagram                */
862         manet_rte_ip_pkt_destroy (ip_pkptr);
863         FOUT;
864         }
865
866     /* None of the above criteria match. Process    */
867     /* the received route request                    */
868
869     /* Add an entry for the route request in the    */
870     /* route request table                            */
871     dsr_route_request_forwarding_table_entry_insert (route_request_table_ptr, ip_dgram_fd_ptr->src_addr,
872                 route_request_option_ptr->target_address, route_request_option_ptr->identification);
873
874 #ifndef ENABLE_DSR_QOS_EXTENSIONS
875     if (cached_route_replies_function)
876         {
877         /* Check if there exists a route from this node    */
878         /* to the destination if the cached route reply */
879         /* functionality has been enabled on this node    */
880         path_ptr = dsr_route_cache_entry_access (route_cache_ptr, route_request_option_ptr->target_address, OPC_FALSE);
881
882         if (path_ptr != OPC_NIL)
883             {
884             /* A route exists to the target address from    */
885             /* this node. Send a "cached" route reply        */
886             /* based on certain restrictions                */
887             dsr_rte_cached_route_reply_send (dsr_tlv_ptr, path_ptr, ip_dgram_fd_ptr);
888
889             /* Destroy the route request packet       */
890             manet_rte_ip_pkt_destroy (ip_pkptr);
891
892             FOUT;
893             }
894         }
895 #endif
896
897     /* No route exists to the destination from this node    */
898     /* Re-broadcast this packet with a short jitter.        */
899     dsr_rte_jitter_schedule (ip_pkptr, dsr_tlv_ptr->option_type);
900
901     FOUT;
902     }
903
904
905 static void
905 dsr_rte_received_route_reply_process (Packet* ip_pkptr, DsrT_Packet_Option* dsr_tlv_ptr)
906     {
907     DsrT_Route_Reply_Option*        route_reply_option_ptr = OPC_NIL;
908     IpT_Dgram_Fields*                 ip_dgram_fd_ptr = OPC_NIL;
909     IpT_Rte_Ind_Ici_Fields*         intf_ici_fdstruct_ptr = OPC_NIL;
910     int                               num_hops, count;
911     InetT_Address*                    hop_address_ptr;
912     InetT_Address*                    next_hop_addr_ptr;
913     char                            src_node_name [OMSC_HNAME_MAX_LEN];
914     char                            src_hop_addr_str [INETC_ADDR_STR_LEN];
915     char                            dest_node_name [OMSC_HNAME_MAX_LEN];
916     char                            dest_hop_addr_str [INETC_ADDR_STR_LEN];
917     char                            temp_str [2048];
918     char*                            route_str;
919
920     /* Processes the received route reply option    */
```

```
921     /* Section 6.2.5 of the draft                   */
922     FIN (dsr_rte_received_route_reply_process (<args>));
923
924     /* Access the information from the incoming IP packet    */
925     manet_rte_ip_pkt_info_access (ip_pkptr, &ip_dgram_fd_ptr, &intf_ici_fdstruct_ptr);
926
927     if (LTRACE_ACTIVE)
928         {
929         route_str = dsr_support_option_route_print (dsr_tlv_ptr);
930         inet_address_print (src_hop_addr_str, ip_dgram_fd_ptr->src_addr);
931         inet_address_to_hname (ip_dgram_fd_ptr->src_addr, src_node_name);
932         inet_address_print (dest_hop_addr_str, ip_dgram_fd_ptr->dest_addr);
933         inet_address_to_hname (ip_dgram_fd_ptr->dest_addr, dest_node_name);
934         sprintf (temp_str, "from node %s (%s) destined to node %s (%s) with route",
935             src_hop_addr_str, src_node_name, dest_hop_addr_str, dest_node_name);
936         op_prg_odb_print_major ("Received a route reply option in packet", temp_str, route_str, OPC_NIL);
937         op_prg_mem_free (route_str);
938         }
939
940     /* If this node is the destination of the route reply    */
941     /* then no more processing needs to be done               */
942     if (manet_rte_address_belongs_to_node (module_data_ptr, ip_dgram_fd_ptr->dest_addr) == OPC_TRUE)
943         {
944 #ifdef ENABLE_DSR_QOS_EXTENSIONS
945 #ifndef ENABLE_DSR_YANG_BANDWIDTH
946         int flow_id;
947         Dsr_Ext_Connection* conn_ptr;
948         conn_ptr = dsr_ext_get_connection_ptr(ip_dgram_fd_ptr->src_addr, -1, DSRC_ROUTE_REQUEST);
949         flow_id = conn_ptr->flow_id;
950
951         /* clear the history of the discovery process */
952         dsr_ext_destroy_connection_ptr(conn_ptr);
953         /* Record the successfull route discover in the global statistic*/
954 #endif
955 #else //not defined ENABLE_DSR_QOS_EXTENSIONS
956         /* Destroy the route reply packet    */
957         manet_rte_ip_pkt_destroy (ip_pkptr);
958 #endif
959         FOUT;
960         }
961     /* This node is not the destination of the route reply    */
962
963     /* Determine the next hop to which this route reply        */
964     /* needs to be sent.                                      */
965     route_reply_option_ptr = (DsrT_Route_Reply_Option*) dsr_tlv_ptr->dsr_option_ptr;
966
967     /* Get the number of hops    */
968     num_hops = op_prg_list_size (route_reply_option_ptr->route_lptr);
969     for (count = (num_hops - 1); count >= 0; count--)
970         {
971         /* Get each hop and determine if it belongs    */
972         /* to this node.                              */
973         hop_address_ptr = (InetT_Address*) op_prg_list_access (route_reply_option_ptr->route_lptr, count);
974
975         if (manet_rte_address_belongs_to_node (module_data_ptr, *hop_address_ptr) == OPC_TRUE)
976             {
977             if (count == 0)
978                 {
979                 /* The next hop is the destination    */
980                 next_hop_addr_ptr = &ip_dgram_fd_ptr->dest_addr;
981                 }
982             else
982                 {
983                 /* This hop belongs to this node    */
984                 /* Access the next hop address        */
985                 next_hop_addr_ptr = (InetT_Address*) op_prg_list_access (route_reply_option_ptr->route_lptr, (count - 1));
986                 }
987
988             break;
989             }
990         }
991
992     if (count < 0)
993         {
994         /* None of the hops in the route reply    */
995         /* belong to this node. This is an        */
996         /* overheard packet. Discard this         */
997         /* packet as this is only used to         */
998         /* update the node's route cache        */
999         manet_rte_ip_pkt_destroy (ip_pkptr);
1000
1001         FOUT;
1002         }
1003
1004     /* Update the statistic for the total traffic sent    */
1005     dsr_support_total_traffic_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
1006
```

```
1007     /* Update the statistics for the routing traffic sent    */
1008     dsr_support_routing_traffic_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
1009
1010     /* Forward the packet to the next hop address    */
1011     manet_rte_to_mac_pkt_send (module_data_ptr, ip_pkptr, *next_hop_addr_ptr, ip_dgram_fd_ptr, intf_ici_fdstruct_ptr);
1012     FOUT;
1013     }
1014
1015
1016 static void
1016 dsr_rte_received_dsr_source_route_option_process (Packet* ip_pkptr)
1017     {
1018     IpT_Dgram_Fields*                ip_dgram_fd_ptr = OPC_NIL;
1019     IpT_Rte_Ind_Ici_Fields*         intf_ici_fdstruct_ptr = OPC_NIL;
1020     Packet*                            dsr_pkptr = OPC_NIL;
1021     DsrT_Source_Route_Option*        source_route_option_ptr = OPC_NIL;
1022     InetT_Address                    next_hop_addr;
1023     DsrT_Packet_Option*                ack_request_dsr_tlv_ptr = OPC_NIL;
1024     DsrT_Packet_Option*                dsr_tlv_ptr = OPC_NIL;
1025     DsrT_Packet_Option*                route_error_tlv_ptr = OPC_NIL;
1026     InetT_Address                    rcvd_intf_address;
1027     InetT_Address                    current_hop_address;
1028     Boolean                            app_pkt_set = OPC_FALSE;
1029     char                             src_node_name [OMSC_HNAME_MAX_LEN];
1030     char                             src_hop_addr_str [INETC_ADDR_STR_LEN];
1031     char                             dest_node_name [OMSC_HNAME_MAX_LEN];
1032     char                             dest_hop_addr_str [INETC_ADDR_STR_LEN];
1033     char                             temp_str [2048];
1034     char*                             route_str;
1035     List*                             temp_lptr;
1036     InetT_Address*                    copy_address_ptr;
1037     InetT_Address*                    hop_address_ptr;
1038     int                               num_hops, count, num_nodes;
1039     InetT_Addr_Family                addr_family;
1040
1041     /* Processes the received source route    */
1042     /* option in the IP datagram            */
1043     FIN (dsr_rte_received_dsr_source_route_option_process (<args>));
1044
1045     /* Access the information from the incoming IP packet    */
1046     manet_rte_ip_pkt_info_access (ip_pkptr, &ip_dgram_fd_ptr, &intf_ici_fdstruct_ptr);
1047
1048     /* Figure out whether we are dealing with an*/
1049     /* IPv4 packet or an IPv6 packet.            */
1050     addr_family = inet_address_family_get (&(ip_dgram_fd_ptr->dest_addr));
1051
1052     /* Get the DSR packet from the IP datagram    */
1053     op_pk_nfd_get (ip_pkptr, "data", &dsr_pkptr);
1054
1055     /* Check if an application packet is set    */
1056     app_pkt_set = op_pk_nfd_is_set (dsr_pkptr, "data");
1057
1058     /* Get the source route option from the DSR packet    */
1059     dsr_tlv_ptr = dsr_rte_packet_option_get (dsr_pkptr, DSRC_SOURCE_ROUTE);
1060
1061     /* Get the route error option from the DSR packet if one exists    */
1062     route_error_tlv_ptr = dsr_rte_packet_option_get (dsr_pkptr, DSRC_ROUTE_ERROR);
1063
1064     /* Set the DSR packet into the IP datagram    */
1065     op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
1066
1067     /* Get the source route option    */
1068     source_route_option_ptr = (DsrT_Source_Route_Option*) dsr_tlv_ptr->dsr_option_ptr;
1069
1070     if (LTRACE_ACTIVE)
1071         {
1072         route_str = dsr_support_option_route_print (dsr_tlv_ptr);
1073         inet_address_print (src_hop_addr_str, ip_dgram_fd_ptr->src_addr);
1074         inet_address_to_hname (ip_dgram_fd_ptr->src_addr, src_node_name);
1075         inet_address_print (dest_hop_addr_str, ip_dgram_fd_ptr->dest_addr);
1076         inet_address_to_hname (ip_dgram_fd_ptr->dest_addr, dest_node_name);
1077         sprintf (temp_str, "from node %s (%s) destined to node %s (%s) with route",
1078             src_hop_addr_str, src_node_name, dest_hop_addr_str, dest_node_name);
1079         op_prg_odb_print_major ("Received a source route option in  packet", temp_str, route_str, OPC_NIL);
1080         op_prg_mem_free (route_str);
1081         }
1082
1083     /* Examine if there is an opportunity for automatic    */
1084     /* route shortening. If this node is not the         */
1085     /* intended next hop, but is named in the later     */
1086     /* unexpanded portion of the source route, there is    */
1087     /* an opportunity for automatic route shortening    */
1088
1089     /* Get the received interface address    */
1090     rcvd_intf_address = manet_rte_rcvd_interface_address_get (module_data_ptr, intf_ici_fdstruct_ptr, addr_family);
1091
1092     /* Get the current hop address in the source route    */
```

```
1093     current_hop_address = dsr_pkt_support_source_route_hop_obtain (source_route_option_ptr, DsrC_Current_Hop,
ip_dgram_fd_ptr->src_addr,
1094                                                                    ip_dgram_fd_ptr->dest_addr);
1095
1096     /* If the intended next hop is not this node, then    */
1097     /* check if there is an opportunity for automatic    */
1098     /* route shortening.                                 */
1099     if (inet_address_equal (rcvd_intf_address, current_hop_address) == OPC_FALSE)
1100         {
1101 #ifndef ENABLE_DSR_EXTENSIONS
1102         /* Check and perform for automatic route shortening        */
1103         /* Destroy the packet if route shortening is success     */
1104         /* or if it fails (i.e. overheard packet and this node     */
1105         /* not even in the source route)                         */
1106         dsr_rte_automatic_route_shortening_check (ip_dgram_fd_ptr,
1107             source_route_option_ptr, intf_ici_fdstruct_ptr);
1108 #endif
1109
1110         /* Free the memory allocated to rcvd_intf_address */
1111         inet_address_destroy (rcvd_intf_address);
1112
1113         /* Discard the overheard packet    */
1114         manet_rte_ip_pkt_destroy (ip_pkptr);
1115
1116         FOUT;
1117         }
1118     /* Free the memory allocated to rcvd_intf_address */
1119     inet_address_destroy (rcvd_intf_address);
1120
1121
1122     /* If there are no more hops in the source route    */
1123     /* this is the destination of the packet            */
1124     if (source_route_option_ptr->segments_left == 0)
1125         {
1126         /* Export the route taken if the    */
1127         /* source node has the attribute    */
1128         /* set or the simulation attribute    */
1129         /* has been enaabled                */
1130         if ((source_route_option_ptr->export_route) && (route_error_tlv_ptr == OPC_NIL))
1131             {
1132             dsr_support_route_print_to_ot (ip_dgram_fd_ptr, source_route_option_ptr->route_lptr);
1133             }
1134         if ((routes_dump) && (route_error_tlv_ptr == OPC_NIL))
1135             {
1136             if (inet_address_equal (ip_dgram_fd_ptr->src_addr, INETC_ADDRESS_INVALID) == OPC_FALSE)
1137                 {
1138                 /* Print the single hop route    */
1139                 temp_lptr = op_prg_list_create ();
1140
1141                 /* Read the source node    */
1142                 copy_address_ptr = inet_address_create_dynamic (ip_dgram_fd_ptr->src_addr);
1143                 op_prg_list_insert (temp_lptr, copy_address_ptr, OPC_LISTPOS_TAIL);
1144
1145                 /* Add the intermediate hops    */
1146                 num_hops = op_prg_list_size (source_route_option_ptr->route_lptr);
1147
1148                 for (count = 0; count < num_hops; count++)
1149                     {
1150                     hop_address_ptr = (InetT_Address*) op_prg_list_access (source_route_option_ptr->route_lptr, count);
1151                     copy_address_ptr = inet_address_copy_dynamic (hop_address_ptr);
1152                     op_prg_list_insert (temp_lptr, copy_address_ptr, OPC_LISTPOS_TAIL);
1153                     }
1154
1155                 /* Read the destination node    */
1156                 copy_address_ptr = inet_address_create_dynamic (ip_dgram_fd_ptr->dest_addr);
1157                 op_prg_list_insert (temp_lptr, copy_address_ptr, OPC_LISTPOS_TAIL);
1158
1159                 /* Dump the route    */
1160                 manet_rte_path_display_dump (temp_lptr);
1161
1162                 /* Free the contents of the list    */
1163                 num_nodes = op_prg_list_size (temp_lptr);
1164
1165                 while (num_nodes > 0)
1166                     {
1167                     hop_address_ptr = (InetT_Address*) op_prg_list_remove (temp_lptr, OPC_LISTPOS_HEAD);
1168                     inet_address_destroy_dynamic (hop_address_ptr);
1169                     num_nodes--;
1170                     }
1171
1172                 /* Free the list    */
1173                 dsr_temp_list_clear (temp_lptr);
1174                 }
1175             }
1176
1177         /* Destroy the IP packet    */
1178         manet_rte_ip_pkt_destroy (ip_pkptr);
```

```
1179
1180         FOUT;
1181         }
1182
1183 #ifdef ENABLE_DSR_YANG_BANDWIDTH
1184     //update the flow reservations only for packets routed (transmitted) by this node
1185     dsr_ext_update_flow_reservation(ip_pkptr);
1186 #endif
1187
1188     /* Get the next hop in the source route    */
1189     next_hop_addr = dsr_pkt_support_source_route_hop_obtain (source_route_option_ptr, DsrC_Next_Hop, ip_dgram_fd_ptr->src_addr,
1190                                                              ip_dgram_fd_ptr->dest_addr);
1191
1192     /* If the next address or the destination    */
1193     /* address is a multicast address, destroy    */
1194     /* the packet and do not process further    */
1195     if (inet_address_is_multicast (next_hop_addr) || inet_address_is_multicast (ip_dgram_fd_ptr->dest_addr))
1196         {
1197         /* The next hop or destination address is a     */
1198         /* multicast address. Destroy the packet        */
1199         manet_rte_ip_pkt_destroy (ip_pkptr);
1200
1201         FOUT;
1202         }
1203
1204     /* There is no maintenance scheduled for the next    */
1205     /* hop. Check if maintenance is needed against the    */
1206     /* maintenance holdoff time                           */
1207     if (dsr_maintenance_buffer_maint_needed (maint_buffer_ptr, next_hop_addr) == OPC_TRUE)
1208         {
1209         if (LTRACE_ACTIVE)
1210             {
1211             inet_address_print (dest_hop_addr_str, next_hop_addr);
1212             inet_address_to_hname (next_hop_addr, dest_node_name);
1213             sprintf (temp_str, "to next hop node %s (%s) with ID (%d)", dest_hop_addr_str, dest_node_name, ack_request_identifier);
1214             op_prg_odb_print_major ("Adding a maintenance request option in packet", temp_str, OPC_NIL);
1215             }
1216
1217         /* Create a IP datagram with a maintenance    */
1218         /* request option in the DSR header            */
1219         ack_request_dsr_tlv_ptr = dsr_pkt_support_ack_request_tlv_create (ack_request_identifier);
1220
1221         /* Update the statistic for the number of maintenance requests sent    */
1222         dsr_support_maintenace_stats_update (stat_handle_ptr, global_stathandle_ptr, OPC_TRUE);
1223
1224         /* Get the DSR packet from the IP datagram    */
1225         op_pk_nfd_get (ip_pkptr, "data", &dsr_pkptr);
1226
1227         /* Set the maintenance request option in the DSR packet header    */
1228         dsr_pkt_support_option_add (dsr_pkptr, ack_request_dsr_tlv_ptr);
1229
1230         /* Set the DSR packet into the IP datagram    */
1231         op_pk_nfd_set (ip_pkptr, "data", dsr_pkptr);
1232
1233         /* A maintenance request has been added    */
1234         /* Place a copy of the packet in the    */
1235         /* maintenance buffer for retranmission    */
1236         dsr_maintenance_buffer_pkt_enqueue (maint_buffer_ptr, ip_pkptr, next_hop_addr, ack_request_identifier);
1237
1238         /* Increment the ACK Request identifier    */
1239         ack_request_identifier++;
1240         }
1241
1242     /* Update the statistic for the total traffic sent    */
1243     dsr_support_total_traffic_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
1244
1245     if (app_pkt_set == OPC_FALSE)
1246         {
1247         /* Update the statistics for the routing traffic sent    */
1248         dsr_support_routing_traffic_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
1249         }
1250
1251     /* Send the packet out to the MAC    */
1252     manet_rte_to_mac_pkt_send (module_data_ptr, ip_pkptr, next_hop_addr, ip_dgram_fd_ptr, intf_ici_fdstruct_ptr);
1253
1254     FOUT;
1255     }
1256
1257
1258 static void
1258 dsr_rte_route_request_send (InetT_Address dest_address, Boolean non_prop_route_request, int tos, int flow_id)
1259     {
1260     DsrT_Packet_Option*            dsr_tlv_ptr = OPC_NIL;
1261     IpT_Dgram_Fields*             ip_dgram_fd_ptr = OPC_NIL;
1262     Packet*                        dsr_pkptr = OPC_NIL;
1263     Packet*                        ip_pkptr = OPC_NIL;
1264     char                          dest_node_name [OMSC_HNAME_MAX_LEN];
```

```
1265     char                        dest_hop_addr_str [INETC_ADDR_STR_LEN];
1266     char                        temp_str [2048];
1267     Ici*                        ip_iciptr;
1268     int                            mcast_major_port = IPC_MCAST_ALL_MAJOR_PORTS;
1269 #ifdef ENABLE_DSR_EXTENSIONS
1270     Dsr_Ext_Route_Metrics*         dsr_metrics;
1271 #endif
1272     /* Initiates a route request to a destination    */
1273     FIN (dsr_rte_route_request_send (<args>));
1274 
1275     /* Create a route request TLV option    */
1276     dsr_tlv_ptr = dsr_pkt_support_route_request_tlv_create (route_request_identifier, dest_address);
1277 
1278     /* Create the DSR packet    */
1279     dsr_pkptr = dsr_pkt_support_pkt_create (IpC_Protocol_Unspec);
1280 
1281     /* Set the route request option in the DSR packet header    */
1282     dsr_pkt_support_option_add (dsr_pkptr, dsr_tlv_ptr);
1283 
1284 #ifdef ENABLE_DSR_EXTENSIONS
1285         dsr_metrics = dsr_ext_create_route_request_metrics(dest_address, flow_id);
1286         op_pk_nfd_set_ptr(dsr_pkptr, "Metrics", dsr_metrics, op_prg_mem_copy_create,
1287             op_prg_mem_free, sizeof (Dsr_Ext_Route_Metrics));
1288         op_pk_nfd_set(dsr_pkptr, "flow_id", flow_id);
1289 #else
1290     /* Record the new route request to the global statistic */
1291     total_routes_requested += 1.0; //used within this module to establish when the network is saturated
1292     op_stat_write (total_routes_requested_shandle, 1.0);
1293 #endif
1294 
1295 
1296     /* Set the DSR packet in a newly created IP datagram    */
1297     /* The source address of the IP datagram is the node's    */
1298     /* own IP address and the destination address of the    */
1299     /* IP datagram is the limited broadcast address         */
1300     /* (255.255.255.255) for IPv4 or the all node link         */
1301     /* layer multicast address for IPv6                        */
1302     if (inet_address_family_get (&dest_address) == InetC_Addr_Family_v4)
1303         {
1304         ip_pkptr = dsr_rte_ip_datagram_create (dsr_pkptr, InetI_Broadcast_v4_Addr,
1305                             InetI_Broadcast_v4_Addr, OPC_NIL);
1306         }
1307     else
1307         {
1308         ip_pkptr = dsr_rte_ip_datagram_create (dsr_pkptr, InetI_Ipv6_All_Nodes_LL_Mcast_Addr,
1309                             InetI_Ipv6_All_Nodes_LL_Mcast_Addr, OPC_NIL);
1310 
1311         /* Install the ICI for IPv6 case */
1312         ip_iciptr = op_ici_create ("ip_rte_req_v4");
1313         op_ici_attr_set (ip_iciptr, "multicast_major_port", mcast_major_port);
1314         op_ici_install (ip_iciptr);
1315         }
1316 
1317     if (LTRACE_ACTIVE)
1318         {
1319         inet_address_print (dest_hop_addr_str, dest_address);
1320         inet_address_to_hname (dest_address, dest_node_name);
1321         sprintf (temp_str, "destined to node %s (%s) with ID (%d)", dest_hop_addr_str, dest_node_name, route_request_identifier);
1322         op_prg_odb_print_major ("Broadcasting a route request option in packet", temp_str, OPC_NIL);
1323         }
1324 
1325     /* Increment the route request identifier    */
1326     route_request_identifier++;
1327 
1328     /* Access the IP datagram fields    */
1329     op_pk_nfd_access (ip_pkptr, "fields", &ip_dgram_fd_ptr);
1330 
1331     ip_dgram_fd_ptr->tos = tos;
1332 
1333     /* If the non-propagating route request feature    */
1334     /* has been enabled, set the TTL field in the    */
1335     /* route request packet to one                    */
1336     if (non_prop_route_request)
1337         {
1338         /* Set the TTL to one    */
1339         ip_dgram_fd_ptr->ttl = 1;
1340         }
1341     else
1342         {
1343         /* Set the TTL to the default    */
1344         ip_dgram_fd_ptr->ttl = IPC_DEFAULT_TTL;
1345         }
1346 
1347     /* Insert the originating route request information in    */
1348     /* the originating route request table                    */
1349     dsr_route_request_originating_table_entry_insert (route_request_table_ptr, dest_address, ip_dgram_fd_ptr->ttl);
1350 
```

```
1350     /* Update the statistic for the total traffic sent    */
1351     dsr_support_total_traffic_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
1352
1353     /* Update the statistics for the routing traffic sent    */
1354     dsr_support_routing_traffic_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, ip_pkptr);
1355
1356     /* Update the statistic for the total number of route requests sent    */
1357     dsr_support_route_request_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, non_prop_route_request);
1358
1359     /* Send the packet to the CPU which will broadcast it    */
1360     /* after processing the packet                            */
1361     manet_rte_to_cpu_pkt_send_schedule (module_data_ptr, parent_prohandle, parent_pro_id, ip_pkptr);
1362
1363     /* Clear the ICI if installed */
1364     op_ici_install (OPC_NIL);
1365
1366     FOUT;
1367     }
1368
1369
1370 static void
1370 dsr_rte_route_reply_send (Packet* request_ip_pkptr, DsrT_Packet_Option* request_dsr_tlv_ptr)
1371     {
1372     IpT_Dgram_Fields*                ip_dgram_fd_ptr = OPC_NIL;
1373     DsrT_Route_Request_Option*       route_request_option_ptr = OPC_NIL;
1374     DsrT_Packet_Option*               reply_dsr_tlv_ptr = OPC_NIL;
1375     Packet*                           reply_ip_pkptr = OPC_NIL;
1376     Packet*                           dsr_pkptr = OPC_NIL;
1377     InetT_Address*                   next_hop_addr_ptr;
1378     InetT_Address*                   node_address_ptr;
1379     int                               num_hops;
1380     char                            src_node_name [OMSC_HNAME_MAX_LEN];
1381     char                            src_hop_addr_str [INETC_ADDR_STR_LEN];
1382     char                            dest_node_name [OMSC_HNAME_MAX_LEN];
1383     char                            dest_hop_addr_str [INETC_ADDR_STR_LEN];
1384     char                            next_node_name [OMSC_HNAME_MAX_LEN];
1385     char                            next_hop_addr_str [INETC_ADDR_STR_LEN];
1386     char                            temp_str [2048];
1387     char*                            route_str;
1388     ManetT_Nexthop_Info*            manet_nexthop_info_ptr = OPC_NIL;
1389
1390 #ifdef ENABLE_DSR_EXTENSIONS
1391     int flow_id;
1392     double delay_est;
1393     Dsr_Ext_Route_Metrics*    dsr_metrics;
1394 #endif
1395
1396     /* Sends out a route reply option on    */
1397     /* receipt of a route request packet    */
1398     /* to the source of the route request    */
1399     FIN (dsr_rte_route_reply_send (<args>));
1400
1401     /* Access the IP datagram fields    */
1402     op_pk_nfd_access (request_ip_pkptr, "fields", &ip_dgram_fd_ptr);
1403
1404 #ifdef ENABLE_DSR_EXTENSIONS
1405     op_pk_nfd_get(request_ip_pkptr, "data", &dsr_pkptr);
1406     op_pk_nfd_access(dsr_pkptr,"Metrics", &dsr_metrics);
1407     delay_est = dsr_metrics->delay_est;
1408     op_pk_nfd_get(dsr_pkptr,"flow_id", &flow_id);
1409     op_pk_nfd_set(dsr_pkptr,"flow_id", flow_id);
1410     op_pk_nfd_set(request_ip_pkptr, "data", dsr_pkptr);
1411 #endif
1412
1413     /* Access the route request option    */
1414     route_request_option_ptr = (DsrT_Route_Request_Option*) request_dsr_tlv_ptr->dsr_option_ptr;
1415
1416     /* Remove this node's address from the list    */
1417     node_address_ptr = (InetT_Address*) op_prg_list_remove (route_request_option_ptr->route_lptr, OPC_LISTPOS_TAIL);
1418     inet_address_destroy_dynamic (node_address_ptr);
1419
1420     /* The target address is this node. The route    */
1421     /* reply will be in the order of the route        */
1422     /* request. Hence, the next hop address will be    */
1423     /* the address before the target address        */
1424
1425     /* Get the size of the route list    */
1426     num_hops = op_prg_list_size (route_request_option_ptr->route_lptr);
1427
1428     /* If the number of hops is zero, then the next hop    */
1429     /* is the final destination address (the source    of     */
1430     /* the route request)                                  */
1431     if (num_hops == 0)
1432         {
1433         /* The next hop is the source of the request    */
1434         next_hop_addr_ptr = &ip_dgram_fd_ptr->src_addr;
1435         }
```

```
1436     else
1436         {
1437         next_hop_addr_ptr = (InetT_Address*) op_prg_list_access (route_request_option_ptr->route_lptr, OPC_LISTPOS_TAIL);
1438         }
1439
1440     /* Create the route reply TLV option    */
1441     reply_dsr_tlv_ptr = dsr_pkt_support_route_reply_tlv_create (ip_dgram_fd_ptr->src_addr,
route_request_option_ptr->target_address,
1442                                             route_request_option_ptr->route_lptr, OPC_FALSE);
1443
1444     /* Create the DSR packet    */
1445     dsr_pkptr = dsr_pkt_support_pkt_create (ip_dgram_fd_ptr->protocol);
1446
1447     /* Set the route reply option in the DSR packet header    */
1448     dsr_pkt_support_option_add (dsr_pkptr, reply_dsr_tlv_ptr);
1449
1450     /* Allocate memory for manet_nexthop_info_ptr */
1451     manet_nexthop_info_ptr = (ManetT_Nexthop_Info *) op_prg_mem_alloc (sizeof (ManetT_Nexthop_Info));
1452 #ifdef ENABLE_DSR_EXTENSIONS
1453         dsr_metrics = dsr_ext_create_route_request_metrics(ip_dgram_fd_ptr->src_addr, flow_id);
1454         dsr_metrics->delay_est = delay_est;
1455         op_pk_nfd_set_ptr(dsr_pkptr, "Metrics", dsr_metrics, op_prg_mem_copy_create,
1456             op_prg_mem_free, sizeof (Dsr_Ext_Route_Metrics));
1457         op_pk_nfd_set(dsr_pkptr, "flow_id", flow_id);
1458 #endif
1459
1460     /* Set the DSR packet in a newly created IP datagram    */
1461     /* The destination address of the IP datagram carrying    */
1462     /* the route reply option is the address of the         */
1463     /* initiator of the route request                        */
1464     reply_ip_pkptr = dsr_rte_ip_datagram_create (dsr_pkptr, ip_dgram_fd_ptr->src_addr,
1465                         *next_hop_addr_ptr, manet_nexthop_info_ptr);
1466
1467     /* Insert this route request received in the forwarding    */
1468     /* route request table                                     */
1469     dsr_route_request_forwarding_table_entry_insert (route_request_table_ptr, ip_dgram_fd_ptr->src_addr,
1470                 route_request_option_ptr->target_address, route_request_option_ptr->identification);
1471
1472     if (LTRACE_ACTIVE)
1473         {
1474         route_str = dsr_support_option_route_print (reply_dsr_tlv_ptr);
1475         inet_address_print (src_hop_addr_str, route_request_option_ptr->target_address);
1476         inet_address_to_hname (route_request_option_ptr->target_address, src_node_name);
1477         inet_address_print (dest_hop_addr_str, ip_dgram_fd_ptr->src_addr);
1478         inet_address_to_hname (ip_dgram_fd_ptr->src_addr, dest_node_name);
1479         inet_address_print (next_hop_addr_str, *next_hop_addr_ptr);
1480         inet_address_to_hname (*next_hop_addr_ptr, next_node_name);
1481         sprintf (temp_str, "from node %s (%s) destined to node %s (%s) with next hop %s (%s) for request ID (%ld) with route",
1482             src_hop_addr_str, src_node_name, dest_hop_addr_str, dest_node_name, next_hop_addr_str, next_node_name,
1483             route_request_option_ptr->identification);
1484         op_prg_odb_print_major ("Sending a route reply option in packet", temp_str, route_str, OPC_NIL);
1485         op_prg_mem_free (route_str);
1486         }
1487
1488     /* Update the statistic for the number of route replies sent from the destination    */
1489     dsr_support_route_reply_sent_stats_update (stat_handle_ptr, global_stathandle_ptr, OPC_FALSE);
1490
1491     /* Install the event state     */
1492     /* This event will be processed in ip_rte_support.ex.c while receiving     */
1493     /* DSR control packets. manet_nexthop_info_ptr will point to structure      */
1494     /* containing nexthop info, so IP table lookup is not again done for them. */
1495     op_ev_state_install (manet_nexthop_info_ptr, OPC_NIL);
1496
1497     /* Send the packet after a jitter    */
1498     /* to the CPU                        */
1499     dsr_rte_jitter_schedule (reply_ip_pkptr, DSRC_ROUTE_REPLY);
1500
1501     op_ev_state_install (OPC_NIL, OPC_NIL);
1502
1503     FOUT;
1504     }
1505
1506
1507 void dsr_rte_route_request_expiry_handle (InetT_Address* dest_address_ptr, int PRG_ARG_UNUSED (code))
1508     {
1509     List*                    pkt_lptr = OPC_NIL;
1510     int                        num_pkts;
1511     Packet*                    pkptr = OPC_NIL;
1512     char                     dest_node_name [OMSC_HNAME_MAX_LEN];
1513     char                     dest_hop_addr_str [INETC_ADDR_STR_LEN];
1514     char                     temp_str [2048];
1515 #ifdef ENABLE_DSR_EXTENSIONS
1516     Dsr_Ext_Connection*     connection_ptr;
1517 #endif
1518     /* Handles the route request expiry    */
1519     FIN (dsr_rte_route_request_expiry_handle (<args>));
1520
```

```
1521     /* Check if it is possible to schedule        */
1522     /* another route request. It may not be     */
1523     /* possible if the maximum number of        */
1524     /* retransmissions have been reached or if    */
1525     /* the request period is greater than the    */
1526     /* maximum request period                    */
1527     if (dsr_route_request_next_request_schedule_possible (route_request_table_ptr, *dest_address_ptr) == OPC_FALSE)
1528         {
1529         /* No more requests can be generated for    */
1530         /* this destination node. Delete all         */
1531         /* packets in the send buffer to this         */
1532         /* destination node that is unreachable        */
1533
1534         /* Remove all packets from the send buffer    */
1535         /* to this destination node                    */
1536         pkt_lptr = dsr_send_buffer_pkt_list_get (send_buffer_ptr, *dest_address_ptr, OPC_TRUE);
1537         num_pkts = op_prg_list_size (pkt_lptr);
1538
1539         while (op_prg_list_size (pkt_lptr) > 0)
1540             {
1541             /* Destroy all packets to this destination    */
1542             pkptr = (Packet*) op_prg_list_remove (pkt_lptr, OPC_LISTPOS_HEAD);
1543             manet_rte_ip_pkt_destroy (pkptr);
1544
1545             /* Update the number of data packets discarded    */
1546             op_stat_write (stat_handle_ptr->num_pkts_discard_shandle, 1.0);
1547             op_stat_write (global_stathandle_ptr->num_pkts_discard_global_shandle, 1.0);
1548             }
1549
1550         /* Remove this route request from the    */
1551         /* route request table                    */
1552         dsr_route_request_originating_table_entry_delete (route_request_table_ptr, *dest_address_ptr);
1553         inet_address_destroy_dynamic (dest_address_ptr);
1554
1555         FOUT;
1556         }
1557
1558     /* It is possible to schedule a new route request    */
1559     /* Check if there are any packets that are still    */
1560     /* queued to that destination                        */
1561     pkt_lptr = dsr_send_buffer_pkt_list_get (send_buffer_ptr, *dest_address_ptr, OPC_FALSE);
1562     num_pkts = op_prg_list_size (pkt_lptr);
1563
1564     if (num_pkts == 0)
1565         {
1566         /* There are no packets queued to be sent    */
1567         /* to this destination. Delete the request    */
1568         dsr_route_request_originating_table_entry_delete (route_request_table_ptr, *dest_address_ptr);
1569         inet_address_destroy_dynamic (dest_address_ptr);
1570
1571         FOUT;
1572         }
1573
1574     if (LTRACE_ACTIVE)
1575         {
1576         inet_address_ptr_print (dest_hop_addr_str, dest_address_ptr);
1577         inet_address_to_hname (*dest_address_ptr, dest_node_name);
1578         sprintf (temp_str, "to destination %s (%s)", dest_hop_addr_str, dest_node_name);
1579         op_prg_odb_print_major ("The route request timer has expired", "Rebroadcasting a route request packet", temp_str, OPC_NIL);
1580         }
1581
1582     /* There are packets queued to the destination    */
1583     /* Resend the route request                        */
1584 #ifdef ENABLE_DSR_EXTENSIONS
1585     connection_ptr = dsr_ext_get_connection_ptr( *dest_address_ptr, -1, DSRC_ROUTE_REQUEST);
1586     if(connection_ptr == OPC_NIL)
1587     {
1588     //printf(" gone wonky, NO CONNECTION POINTER FOUND\n");
1589     inet_address_destroy_dynamic (dest_address_ptr);
1590     dsr_temp_list_clear (pkt_lptr);
1591     FOUT;
1592     }
1593
1594     dsr_rte_route_request_send (*dest_address_ptr, OPC_FALSE, connection_ptr->tos, connection_ptr->flow_id);
1595 #else
1596     dsr_rte_route_request_send (*dest_address_ptr, OPC_FALSE, 0, -1);
1597 #endif
1598     inet_address_destroy_dynamic (dest_address_ptr);
1599     dsr_temp_list_clear (pkt_lptr);
1600
1601     FOUT;
1602     }
```