INCREASING BROADCAST RELIABILITY IN VEHICULAR AD HOC NETWORKS

by

Nathan Balon

A master's project submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer and Information Science)
in The University of Michigan
2006

Committee:

Jinhua Guo
Kiumi Akingbehin

© Nathan Balon 2006
All Rights Reserved

ABSTRACT

INCREASING BROADCAST RELIABILITY IN VEHICULAR AD HOC NETWORKS

by
Nathan Balon

Advisor: Jinhua Guo

Broadcast transmissions are the predominate form of network traffic within, an 802.11-based, vehicular ad hoc network (VANET). However, since there is no MAC-layer recovery on broadcast frames the reception rates of broadcast messages can become very low, especially under saturated conditions. In this paper, an adaptive broadcast protocol is presented. The goal of this protocol is to improve the reception rate of broadcast messages. We rely on the observation that a node in a VANET is able to detect network congestion by simply analyzing the sequence numbers of packets it has recently received. Based on the percentage of packets that are successfully received in the last few seconds, a node dynamically adjusts the size of the contention window. As a result of dynamically adjusting the contention window, the performance of the broadcast protocol improves.

I dedicate my work to my Grandmother, Mary Olexick, for providing with a place to live, so I could focus on my education.

ACKNOWLEDGEMENTS

I would like to thank Dr. Jinhua Guo for being my advisor for this research project. Professor Guo presented me the opportunity of learning the steps involved in conducting network research. As a result, this master's project has taught me many valuable skills that I will be able to apply in future ventures. In addition, I appreciate Dr. Guo's willingness to offer feedback on means to improve the project. Even when I became temporarily baffled on a portion of the project, Dr. Guo's constant encouragement helped me move forward. Also, I would like to thank Dr. Kiumi Akingbehin for agreeing to be on the project's committee. In addition, I would like to thank all of the professors from the Computer Science Department at University of Michgan at Dearborn. I am also grateful for Kathie Deluge who proofread this paper for me.

TABLE OF CONTENTS

DEDICATION . . . . **i**

ACKNOWLEDGEMENTS . . . . **ii**

LIST OF FIGURES . . . . **v**

LIST OF TABLES . . . . **vii**

LIST OF APPENDICES . . . . **viii**

CHAPTER

I. Introduction 1

1.1 VANET Characteristics 4
1.2 Motivation 7
1.3 Contributions 9

II. Related Work 11

2.1 DSRC Physical Layer 11
2.1.1 Channel Assignment 12
2.1.2 Control Channel Access 13
2.2 IEEE 802.11 MAC 14
2.2.1 Contention Window 15
2.2.2 Inter-Frame Spacing 16
2.2.3 Hidden Terminal Problem 18
2.3 Priority Access 20
2.4 Broadcast 22
2.4.1 Location Based Broadcast 24
2.4.2 Urban Multi-Hop Broadcast Protocol 25
2.4.3 VCWC Protocol for Cooperative Collision Warning 28
2.4.4 Multi-Hop Broadcast 31

III. Methodology 34

3.1 Passive Monitoring 37
3.1.1 Sequence Numbers 38
3.1.2 Monitoring Network Traffic 39
3.2 Broadcast Table 40
3.2.1 Hash Table 40
3.2.2 Invalidating Old Hash Table Entries 41
3.3 Adaptive Adjustment of the Contention Window 42
3.3.1 Estimated Reception Rate 42

3.3.2 Update Timer . . . 43
3.3.3 Local Reception Rate . . . 44
3.3.4 Contention Window Adjustment . . . 44
3.3.5 Contention Window Sizes . . . 45
3.4 Prioritized Access (QoS) . . . 46
3.4.1 Access Categories . . . 47
3.4.2 Control Channel Applications . . . 47
3.4.3 Arbitration Inter-Frame Space . . . 48
3.4.4 Access Category Contention Window Sizes . . . 49

IV. Simulation and Results . . . 52

4.1 The Network Simulator ns-2 . . . 52
4.1.1 NS-2 Architecture . . . 53
4.1.2 Simulation Scripts . . . 55
4.1.3 NAM: Network Animator . . . 64
4.2 Mobility Model . . . 65
4.2.1 Freeway Mobility Model . . . 66
4.2.2 Map File . . . 68
4.2.3 Mobility Trace . . . 72
4.3 Network Traffic . . . 73
4.3.1 QoS . . . 74
4.3.2 Creating Network Traffic . . . 74
4.3.3 The Network Traffic Program: traffic-w-jitter.pl . . . 77
4.4 Performance Metrics . . . 80
4.4.1 NS-2 Simulation Trace Format . . . 81
4.4.2 Reception Rate . . . 82
4.4.3 Access Delay . . . 84
4.5 Adaptive Broadcast Implementation . . . 86
4.5.1 Broadcast Table Implementation . . . 87
4.5.2 Changes to the 802.11e Source Code . . . 92
4.6 Simulation Results . . . 92
4.6.1 Dynamic Window Simulation . . . 94
4.6.2 Static CW Simulation . . . 99

V. Conclusion and Further Research . . . 101

5.1 Conclusion . . . 101
5.2 Future Work . . . 104
5.2.1 Adaptive Transmission Rate Control . . . 105
5.2.2 Dynamic Transmission Power Control . . . 106

APPENDICES . . . **107**

BIBLIOGRAPHY . . . **205**

LIST OF FIGURES

Figure

2.1 DSRC Channels . . . 13

2.2 802.11 Backoff Procedure . . . 17

2.3 802.11 Inter-Frame Spaces . . . 18

2.4 Hidden Terminal Problem . . . 19

2.5 RTS/CTS Handshake . . . 20

2.6 802.11e Backoff . . . 21

2.7 802.11e Queues . . . 23

2.8 Urban Multi-Hop Broadcast Protocol . . . 27

3.1 802.11 MAC Header . . . 38

3.2 Broadcast Frames Received . . . 40

3.3 Contention Window Sizes . . . 51

4.1 Mirrored NS-2 Objects . . . 54

4.2 NS-2 Architecture . . . 56

4.3 Steps Involved in an NS-2 Simulation . . . 56

4.4 NAM Controls . . . 65

4.5 Simulation Topography . . . 71

4.6 Broadcast Table Entry Class Diagram . . . 88

4.7 Broadcast Table Class Diagram . . . 90

4.8 Simulation Reception Rate 400 Nodes . . . 96

4.9 Simulation Reception Rate 600 Nodes . . . 98

4.10 Static CW Reception Rate for 800 Nodes . . . 100

4.11 Static CW Access Delay for 800 Nodes 100

LIST OF TABLES

Table

3.1 Data Maintained in a Broadcast Table 40

3.2 Contention Window Sizes 46

3.3 Access Categories 47

3.4 Access Categories 50

4.1 Command-Line Options for wireless-simulation.tcl 63

4.2 Data Contained in an NS-2 Trace 63

4.3 Command-Line Options for traffic-w-jitter.pl 78

4.4 Performance Metrics 80

4.5 Trace Events to Determine the Access Delay 86

4.6 Simulation Parameters for the Access Categories 94

4.7 Average Access Delay for 400 Nodes 96

4.8 Average Access Delay for 600 Nodes 97

LIST OF APPENDICES

Appendix

A. Simulation Script 108

A.1 wireless-simulation.tcl 108

B. Mobility 114

B.1 create-map.pl 114
B.2 map.txt 115

C. Network Traffic 120

C.1 traffic-w-jitter.pl 120

D. Performace Metrics 124

D.1 reception.pl 124
D.2 access-time.pl 129

E. Modifications to the Simulator 133

E.1 Broadcast Table 133
E.1.1 broadcast_table.h 133
E.1.2 broadcast_table.cc 135
E.2 802.11e Source Code 145
E.2.1 mac-802_11e.h 145
E.2.2 mac-802_11e.cc 152
E.2.3 mac-timers_802_11e.h 188
E.2.4 mac-timers_802_11e.cc 192

CHAPTER I

Introduction

The rapid advances in wireless technologies provide opportunities to utilize these technologies for advanced vehicle safety applications. In particular, the new Dedicated Short-Range Communication (DSRC) offers the potential to effectively support vehicle-to-vehicle and vehicle-to-roadside safety communications, known today as Vehicle Safety Communication (VSC) technologies. DSRC enables a new class of communication applications that will increase the overall safety and efficiency of the transportation system. The driving force behind the creation of DSRC is the need to improve the safety of the roads. In the United States, as in many of other countries, driving fatalities are one of leading causes of death. As an indication of the severity of the problem, in 1999 there were 6,279,000 motor vehicle accidents that accounted for 41,611 deaths in the United States[10]. One way to improve safety is to alert drivers of a dangerous situations before they are able to observe them.

Intelligent Transportation Systems (ITS) [18] are the future of transportation. As a result of emerging standards, such as 5.9 GHz dedicated short-range communication, vehicles will soon be able to talk to one another as well as their environment. A number of applications will be made available for vehicular networks that will improve the overall safety of the transportation infrastructure. For instance, the

system will be able to monitor traffic to coordinate traffic lights so that traffic flows smoothly. Sensors will use feedback from vehicles to detect traffic jams. Public safety vehicles will broadcast, via the wireless channel, to change traffic signals in order to respond quickly to emergencies. In order to avoid collisions and improve efficiency, cars will communicate with one another to drive cooperatively. These are some of the future applications that will become possible with the advent of the DSRC standard.

Considering the tremendous benefits expected from vehicular communications and the huge number of vehicles, it is clear that *Vehicular Ad Hoc Networks* (VANET) are likely to become the most relevant realization of mobile ad hoc networks. The appropriate integration of on-board computers, road maps, and GPS devices along with communication capabilities, opens tremendous opportunities, but also raises formidable research challenges.

DSRC [5], which is a candidate for use in a VANET, is a short to medium range communication service that supports both public safety and private communication. The communication environment of DSRC is both vehicle-to-vehicle and vehicle-to/from-roadside. DSRC aims to provide high data rates and at the same time minimize latency within relatively small communication zones.

A VANET is a specific type of mobile ad hoc network (MANET), and each node, in a MANET, acts as both a data terminal and a router. The nodes in the network use the wireless medium to communicate with other nodes within their radio range. The benefit of using ad hoc networks is it is possible to deploy these networks in areas where it isn't feasible to install the infrastructure. In the United States there are thousands upon thousands of miles of roads. It would be expensive and unrealistic to install access points to provide coverage for all the roads in the United States. For this reason, ad hoc networks are the promising solution. Vehicles in the VANET

communicate with each other using the DSRC protocols and algorithms. In reality, a VANET is not a pure ad hoc network since *roadside access units* (RSU) are available. A RSU has a transceiver, antenna, processor, and sensors. RSUs are strategically placed along the road in order to provide services to vehicles. For instance, a RSU may be placed near an intersection to improve the flow of traffic through that intersection and to reduce accidents. Also, a commercial entity can deploy a RSU to provide value-added services, such as the announcement of possible places of interest to the driver (e.g., restaurants, motels, and gas stations). Another benefit of ad hoc networks is they can be quickly deployed with no administrator involvement. The administration of a large scale vehicular network would be a difficult task.

A large array of applications are being developed for DSRC. The applications of DSRC are categorized into the following four classes.

- **Vehicle-to-Vehicle** applications transmit messages from one vehicle to another.
- **Vehicle-to/from-Infrastructure** are applications in which messages are sent either from a Road-Side Unit (RSU) to a vehicle or from a vehicle to a RSU.
- **Vehicle-to-Home** are a class of applications used when a vehicle is parked at the driver's residence, for purposes such as transferring data to a vehicle.
- **Routing Based** applications are used when the intended recipient is greater than one-hop away.

Based on these four application classes, the DSRC applications can be further categorized as safety and non-safety applications.

A number of novel problems are associated with VANETs because of the unique characteristics of the network. Also, one of the greatest challenges is the vehicles

in the network move at greater speeds than most other MANETs, leading to a network that can frequently become fragmented. Furthermore, research in the area of vehicular networks is relatively new. In addition, the DSRC standard is incomplete at the time of writing this paper. Because of the uniqueness of these networks and the relatively small amount of research in this area, there are many issues that must be addressed before VANETs are commercially deployable.

1.1 VANET Characteristics

The characteristics of a vehicular ad hoc network are unique compared to other mobile ad hoc networks. The distinguishing properties of a VANET offers opportunities to exploit these properties to increase network performance, and at the same time it presents considerable challenges. A VANET is fundamentally different [7] from other MANETs. First, a VANET is characterized by a rapidly changing but somewhat predictable topology. Second, fragmentation of the network frequently occurs. Third, the effective network diameter of a VANET is relatively small. Fourth, redundancy is limited both temporally and functionally. Last, a VANET poses a number of unique security challenges.

The topology of the VANET changes frequently because of the high mobility of vehicles. Due to the frequent topology changes, the time that a communication link exists between two vehicles is brief. The reason why the link in a VANET is short lived is because vehicles travel at high speeds (approaching speeds of up to 200 km/h). To increase the duration that a link is valid, one solution is to increase the transmission power. The problem associated with increasing a vehicle's transmission range to maintain a communication link is that the throughput of the network decreases. When vehicles travel in opposite directions, as can be expected, a

link exits for a brief period of time. Even when vehicles travel in the same direction, with each vehicle having a transmission range of 500 ft, the wireless link between vehicles exists on the average of a minute. Because vehicles exhibit a high degree of mobility, group membership is difficult to maintain. For example, maintaining an accurate list of neighboring vehicles is challenging. Protocols that rely on group membership are difficult to implement. Nevertheless, the topology of a VANET is also beneficial. Because a vehicle's movement is constrained by the road, the future movement of a vehicle is predictable.

When VANETs are initially deployed only a small percentage of vehicles will be equipped with a transceiver. The limited number of vehicles with a transceiver will lead to frequent fragmentation of the network, causing a portion of the network to become unreachable. Even when a VANET is fully deployed, fragmentation may exist in rural areas or during periods of light traffic, such as late at night. Since it could take years before the majority of cars are equipped with a transceiver, the VANET protocols should not assume that all vehicles can communicate.

As a result of the poor connectivity between nodes, the effective diameter of the network is small. For this reason, it is unrealistic for a node to maintain the complete global topology of the network. The limited effective diameter results in problems when applying existing routing algorithms to a VANET. Traditional routing protocols are either proactive or reactive. To begin, proactive routing algorithms maintain routes by using tables. Frequent exchanges between nodes are needed to keep the routing information valid. Because the topology changes so rapidly, the routes maintained in the routing tables quickly become invalid. Traditional table-based routing approaches (such as DSDV) consume a great deal of bandwidth. Subsequently, reactive routing aims at establishing a route only when one is needed.

The problem with the reactive approach is that a route must be discovered before the first packet is sent. As a result, the time needed to send a message increases. Neither of these two approaches performs particularly well in a VANET. The problem with the proactive approach is that it does not scale well. The problem with the reactive approach is that even when a route to a destination is found right before transmitting a message the route may also be very short-lived because of mobility. In addition, the expected path life of a route decreases as the number of hops increases. A path may cease to exist shortly after it is discovered. Using traditional ad hoc routing algorithm is likely to result in a routing error when a message is sent greater than three or four hops. Also, routing is not likely to play as large a role in a VANET as it does in other networks. In a VANET, it is more important to send a message towards a certain location.

Redundancy is crucial for services such as security. In a VANET, redundancy is limited both temporally and functionally. Since links between nodes fail to exist for a significant period of time, redundancy is difficult to achieve.

Privacy and security are other issues that must be addressed. In order for the adoption of a VANET to gain public support, a driver's anonymity must be preserved. For instance, the general public is unlikely to support a VANET if it is possible to track a driver's movement. If anonymity features are not included, it would be possible for third parties to monitor a driver's daily activities. For this reason, mechanisms are needed to ensure the driver's privacy. Second, it is crucial that a message's integrity is maintained. It should not be possible to tamper with the messages in the VANET. The tampering of safety messages could result in automobile accidents, defeating the VANET's purpose. If the location field in a packet is tampered with, a vehicle would receive incorrect information regarding the sur-

rounding vehicles. If strict security measures are not put in place, an attacker would be able inject false data into the network resulting in the flow of traffic being altered and causing chaos within the transportation system.

These are some of the unique challenges related to a VANET. These are not the only unique characteristics of a VANET, but they represent some of the basic issues that must be addressed in a VANET.

1.2 Motivation

Broadcast messages will play a larger role than the use of unicast messages in a VANET. The largest percentage of the messages sent in a vehicular network are broadcast in nature. The primary uses of broadcast messages are: broadcasting emergency warning messages and periodically broadcasting a vehicles state (e.g. a vehicles velocity, acceleration, location, and direction). The low-layer technology used for DSRC is a variant of IEEE 802.11a [10]. However, 802.11 technology is known for not being able to manage the medium resources very efficiently, especially in case of broadcast messages. Providing reliable delivery of broadcast messages in a VANET introduces several key technical challenges.

No retransmission is possible for failed broadcast transmissions because they are undetectable. A failed unicast transmission is detected by the lack of acknowledgment (ACK) from the receiver. However, it is not practical to receive an ACK from each node for a broadcast message. If acknowledgments were used, a problem known as the “ACK explosion problem” [19] would exist. Each receiving node would, at almost the same instance, send an ACK back to the transmitting node causing collisions and contention.

The contention window size, CW, fails to change because of the lack of MAC-level

recovery for broadcast frames. In order to control congestion, the contention window (CW) is exponentially increased each time a failed transmission is detected. Since there is no detection of failed broadcast transmissions, the size of the CW fails to change for broadcast traffic as it does for unicast traffic. If a large number of nodes are contending for access, excessive collisions result.

The hidden terminal problem exists because the RTS/CTS exchange cannot be used. The hidden terminal problem [17] is one of the main causes of collisions in a wireless network. The IEEE 802.11 protocols use an optional RTS/CTS handshake followed by an acknowledgment to guarantee the delivery of a unicast packet. Broadcast messages, on the other hand, do not use the RTS/CTS exchange because it would flood the network with traffic.

The vehicular network should support the ability to prioritize messages. When emergency warning messages are broadcast, they should be given a higher access priority than common data messages.

The collision rate of broadcast frames increases as the distance from the sender increases. Under saturated conditions, the probability of the reception of a broadcast frame sharply decreases at distance greater than 66% of the transmission range[22] . The primary reason for the decreased reception rate is the hidden terminal problem. One solution to increase the probability of reception is to implement a repetition strategy where a message is broadcast multiple times. The main drawback of repetition is that it generates excessive traffic wasting network bandwidth.

Multi-hop broadcasts are another challenge. A naïve approach, such a flooding a broadcast frame, results in a broadcast storm [19] leading to a significant number of frames colliding and resulting in poor use of the network resources. A flooding algorithm works by each node receiving a broadcast message for the first time re-

broadcasting the message. A message sent to n nodes results in the message being rebroadcast n times. The problem is characterized by redundant rebroadcasts, contention, and collisions. Creating an efficient multi-hop broadcast is an open problem.

The problems mention above result in an unreliable broadcast protocol for DSRC. Because of this, broadcasting in a VANET is best effort service. When a message is broadcast, the transmitter of a broadcast can only hope that the message is successfully received.

1.3 Contributions

The goal of this research paper is to develop an adaptive broadcast protocol that improves the reliability of delivering broadcast messages in a VANET. An adaptive broadcast protocol is proposed. The adaptive protocol relies on the observation that a node is able to detect collisions and congestion by simply analyzing the sequence numbers of packets it has recently received.

In a VANET, each node broadcasts its status to its neighbors approximately 10 times per second. While a node does not know if the packets it sent are correctly delivered or not, it knows the exact percentage of packets sent from neighboring nodes that are successfully received. Based on the percentage of packets that are successfully received in the last few seconds, a node is able to determine the current local conditions of the network and roughly estimate the number of neighbors in its communication range. Therefore, a node is able to dynamically adjust the parameters it uses, such as contention window size, transmission rate, and transmission power, to improve the delivery rate of broadcast messages. The focus of this research is to use the network feedback to dynamically adjust the contention window for broadcast transmissions in a VANET.

The novelty of this approach is that no communication control overhead is involved. In addition, the proposed technique does not require changing the existing IEEE 802.11 standard instead it focuses on optimizing the parameters used by 802.11. For that reason, we believe that this approach has very good chance of being commercially deployed.

CHAPTER II

Related Work

A number of technologies are used in vehicular networks that are necessary for broadcast transmissions. First, Section 2.1 contains a description of the DSRC physical layer. The focus of this section is channel assignment and control channel access for DSRC. Next, Section 2.2 describes the IEEE 802.11 MAC protocol. The focal points of this section are the distributed coordination function (DCF) along with some of the problems associated with DCF. Subsequently, Section 2.3 explains IEEE 802.11e, which is used to provide quality of service (QoS) for the different traffic classes. Finally, Section 2.4 contains an overview of some of the broadcast algorithms that have been proposed for VANETs. Section 2.4 also briefly describes the open problem of creating a reliable multi-hop broadcast. These are some of the related works that are relevant to improving the reliability of broadcasts in a VANET.

2.1 DSRC Physical Layer

The physical layer of the network protocol stack is responsible for placing raw bits on to channel. This section describes the physical layer of DSRC. To begin, the channel assignment of DSRC is described. Next, control channel access is discussed. In addition, the problem of coordinating access to multiple channels is also examined.

2.1.1 Channel Assignment

The FCC allocated 75 MHz of the radio spectrum for DSRC. The 5.9 GHz DSRC spectrum is composed of six service channels that are 10 MHz each. Also, one 10 MHz control channel exists. The FCC recommends no unlicensed use of the DSRC band. Figure 2.1 provides the channel layout for DSRC. The data rates possible for the 10 MHz channels are 6, 9, 12, 18, 24, and 27 Mb/s. In addition, the physical layer preamble is transmitted at 3 Mb/s. The modulation scheme used by DSRC is Orthogonal Frequency Division Multiplexing (OFDM). Also, the subcarrier frequency spacing of 802.11a is double that of DSRC. In addition, DSRC doubles the guard period in comparison to 802.11a. The modified version of 802.11a, used for DSRC, is known as 802.11p. The following list contains the channels of DSRC and the type of applications that are supported by the channel.

- Channel 172 is reserved for medium power safety applications.
- Channel 174 is reserved for medium power applications that are shared by all.
- Channel 175 is a combination of channels 174 and 176.
- Channel 176 is reserved for medium power applications that are shared by all.
- Channel 178 is the control channel it support all power levels, safety application broadcasts, service announcements, and vehicle-to-vehicle broadcasts messages.
- Channel 180 is reserved for low power configurations and provides little interference when units are separated by 50 ft or more.
- Channel 181 is a combination of channels 180 and 182.
- Channel 182 is reserved for low power configurations and provides little interference when units are separated by 50 ft or more.

5.850 GHz		CH. 175			CH. 181		5.925 GHz
Reserved 5 MHz	CH. 172 Service 10 MHz	CH. 174 Service 10 MHz	CH. 176 Service 10 MHz	CH. 178 Control 10 MHz	CH. 180 Service 10 MHz	CH. 182 Service 10 MHz	CH. 184 Service 10 MHz

Figure 2.1: DSRC Channels

- Channel 184 is reserved for a high power service channel that is used to coordinate intersection applications.

2.1.2 Control Channel Access

Current wireless technologies are only able to monitor a single channel at a time. As a result, when DSRC is initial deployed, it is expected, each vehicle will have a single transceiver that can only listen to single channel at any instance. To overcome this problem, it is possible to equip either an OBU or RSU with multiple transceivers allowing them access to multiple channels simultaneously. The drawback to having multiple transceivers is it increases the complexity and the cost. For the initial roll out of DSRC, it is envisioned that vehicles will have only a single transceiver. As a result of only being able to listen to single channel at time is channel coordination is needed.

Channel 178 is reserved for the control channel. The control channel is the most important channel of DSRC, and the efficient use of this channel is critical. Each OBU monitors the control channel for both safety messages and service channel announcements. The control is monitored by each vehicle and RSU. Since there is a limited amount of bandwidth available, communication on the control channel is

brief. The FCC recommends that the control channel is used for messages that take less than 200 μs to transmit. If the communication last longer than 200 μs another channel must be used.

Vehicles must periodically switch to the control channel to receive safety messages. A requirement of DSRC is that all vehicles must switch to the control channel every 100 ms and remain on the channel for a minimum amount of time. The reason vehicles switch to the control channel, every 100 ms, is to receive safety messages from the surrounding vehicles. To guarantee that safety messages are not sent before a vehicle switches to the control channel, the time that a vehicle switches to the channel must be synchronized. One possible way to synchronize control channel access is with the time received from a GPS receiver. There are a number of proposals, for the DSRC standard, for how to implement the synchronization with GPS.

Also, the control channel is used for service announcements. When a service of interest is discovered, the OBU switches from the control channel to a service channel to use the service. For instance, a RSU may provide a map update. An updated map is then transferred to a vehicle. The OBU of a vehicle discovers a map update and switches to a service channel to begin the transfer of the new digital map. If the transfer takes too long to complete, the vehicle must switch to the control channel to receive safety messages and then switch back to the service channel to resume the file transfer. The control channel coordination allows a vehicle to correctly receive safety messages and also use the available services in the network.

2.2 IEEE 802.11 MAC

The physical layer is primary difference between the IEEE 802.11 variants. For instance, the modulation schemes are different for 802.11a and 802.11b. The 802.11b

protocol uses High-Rate Direct Sequence Spread Spectrum (HR-DSSS) as opposed to the 802.11a protocol that uses Orthogonal Frequency Division Multiplexing (OFDM). Also, the amount of bandwidth available and the allocated frequency spectrum varies between the different 802.11 physical layers. DSRC, that is used for VANETs, uses a variant of the 802.11a protocol. While the physical layer of the different 802.11 protocols vary, the media access control (MAC) of 802.11 remains the same for the different IEEE 802.11 protocols.

The 802.11 standard defines two MAC protocols, the Distributed Coordination Function (DCF) and the Point Coordination Function (PCF). The DCF is an asynchronous contention based access protocol. In a contention based protocol, all nodes that have data to send contend for access to the channel. On the other hand, PCF is a contention free protocol that provides access to the medium by scheduling when a node can transmit. Contention free protocols, such as PCF, enable the use of real-time services. Although there are benefits to using the PCF, it is not applicable for a VANET in most cases because it relies on a central node to support the real-time delivery of packets. For this reason, the majority of communication that takes place in DSRC uses the DCF.

2.2.1 Contention Window

The 802.11 family of protocols use Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) with an acknowledgment (ACK). The goal of the CSMA/CA protocol is preventing excessive collisions from occurring. In addition, the ACK signals, to the transmitter, that a packet was successfully received.

DCF achieves collision avoidance with a random back-off procedure. The IEEE 802.11 standard uses the concept of time-slots. Each time-slot for 802.11a is 9 μs, but in general the length of a time-slot will vary based on the physical layer characteristics

of the IEEE 802.11 protocol. For example, the length of a time-slot for DSRC is 13 μs as opposed to the 9 μs for 802.11a.

When a node begins a transmission, it randomly selects the number of time-slots it must wait before transmitting, which is known as the back-off process. One clock tick of the back-off timer expires, when the medium remains free from transmission for one time-slot. Collisions are avoided by nodes randomly selecting different values for their back-off timers. The value of the back-off timer is chosen randomly in the range of $[0, CW)$, where CW is the size of the contention window.

If two nodes wish to transmit a frame at the same time and they select different values for their back-off timers then a collision is avoided, because the nodes will transmit at different times. On the other hand, if two nodes both decrement their back-off timers to zero at the same time, a collision occurs. The back-off time ($T_{backoff}$) is calculated with Equation 2.1 where j is the number of retransmissions:

$$T_{backoff} = Rand(0, 2^j * CW_{min}) * T_{slot} \quad (2.1)$$

The contention window continues to increase, as illustrated in Figure 2.2, after each failed transmission until CW_{max} is reached. If the transmission of a frame does not succeed after a predefined numbers of attempts the frame is discarded. On the other hand, after a frame is successfully received the CW is reset to CW_{min}. In the case of broadcast transmissions, a node always transmits with its CW set to CW_{min} since there is no way to determine if a packet is successfully received or not.

2.2.2 Inter-Frame Spacing

In addition, DCF uses a number of different inter-frame spaces. When a node wishes to transmit a frame, it must wait for the *distributed inter-frame spacing* (DIFS) of 34 μs to expire, for 802.11a. (As previously discussed, time of the inter-

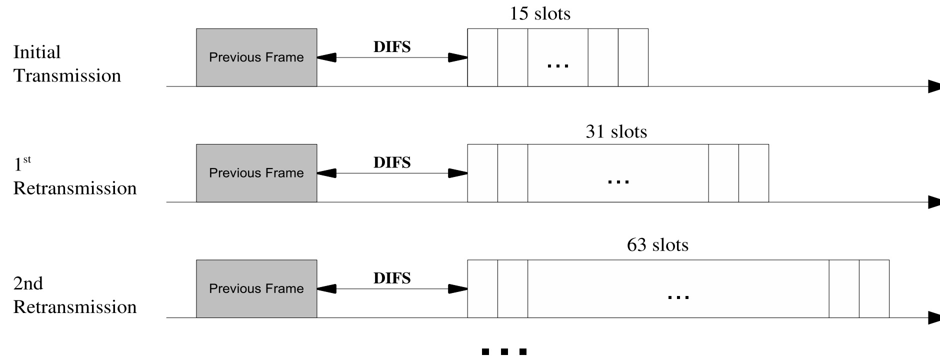

Figure 2.2: 802.11 Backoff Procedure

frame spacing will vary based on the specific 802.11 protocol that is used.) During this time the wireless medium must remain free. If a transmission is overhead while a node is waiting for its DIFS timer to expire, the node then defers its attempted access to the medium until the medium becomes free. When the overheard transmission is complete the node will then begin to listen to the medium until the DIFS has expired.

Once the DIFS is complete, the node will begin to count down its back-off timer. If a transmission occurs before the back-off timer reaches zero, the node will then pause its back-off timer. The node must then wait for the medium to remain free for the length of the DIFS timer to resume the back-off timer. Finally, when a node back-off timer reaches zero it will begin its transmission.

The wireless transmission is made reliable with the introduction of an explicit acknowledgment mechanism. The intended receiver of a frame transmits an acknowledgment (ACK) which alerts the sender that the frame has been successfully received. If an ACK is not received by the sender of a frame, it is assumed that the frame was not successfully received, and another attempt is made to transmit the frame.

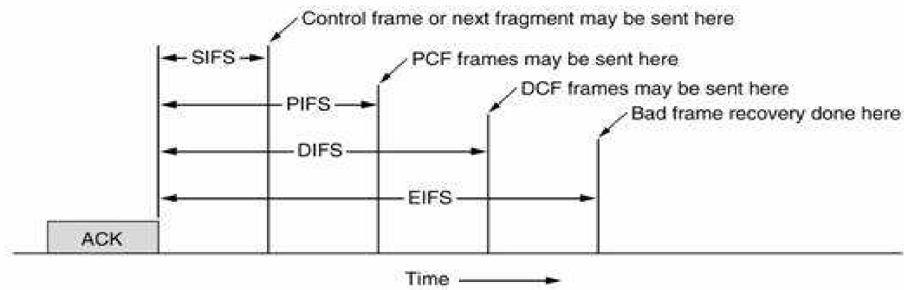

Figure 2.3: 802.11 Inter-Frame Spaces

A *short inter-frame space* (SIFS) is used before the transmission of an ACK. For 802.11a the value of SIFS is 16 μs. A smaller value is used to prevent other nodes from gain access to the medium (i.e., the other nodes must wait 34 μs).

The *PCF inter-frame space* (PIFS) is set to a value between SIFS and DIFS. The value of PIFS is set so that an access point can poll a station, so the station will responded before other nodes are able to gain access to the channel using DIFS. It results in nodes that are using the DIFS from deferring access the wireless medium while another station is being polled by an access point. In addition, there is a extended inter-frame space, which is the longest, that is used if an error occurs. Figure 2.3 contains the inter-frame spacing used for 802.11.

2.2.3 Hidden Terminal Problem

The "hidden terminal problem" is one of the primary factors that affects the reliability of DCF. The hidden terminal problem results in excessive collisions in a wireless network. Consequently, hidden terminal problem occurs when there are two nodes that are outside the transmission range of each other but each transmits to a node that is shared between them. In Figure 2.4, nodes $S1$ and $S2$ cannot sense each others transmissions. Therefore, the medium appears free to both $S1$ and $S2$. If both $S1$ and $S2$ were to transmit to $R1$ at the same time, a collision would occur

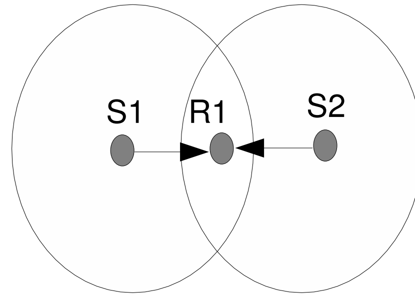

Figure 2.4: Hidden Terminal Problem

at $R1$ and neither of the frames would be successfully received.

The 802.11 protocol addresses the hidden terminal problem with an optional Request-to-Send/Clear-to-Send (RTS/CTS) exchange. The RTS/CTS exchange occurs before any data is transferred[16].

Figure 2.5 illustrates the RTS/CTS sequence. When node $S1$ has data to send, once it is able to gain access to the medium (e.g., after the nodes back-off timer expires), node $S1$ first transmits a RTS to the intended receiver $R1$. When $R1$ receives the RTS after a Short Inter-Frame Spacing (SIFS) has expired, node $R1$ will respond with a CTS. When the CTS is received at $R1$ it will signal to node $R1$ that $S1$ is ready to receive a data frame. The hidden node problem is mostly eliminated, when $S2$ overhears the CTS transmitted from $R1$ it then sets the network allocation vector (NAV) for the amount of time it takes to complete the communication. Node $S2$ will then defer from accessing the wireless medium until the NAV expires and the transmission between $S1$ and $R1$ is complete. When $S2$ overhears the ACK sent from $R1$ it knows the transmission is complete. After the DCF Inter-Frame Spacing (DIFS) has elapsed nodes in the network can then begin to contend for access to the channel.

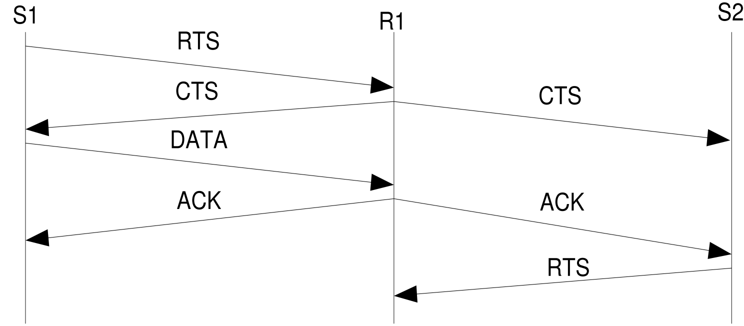

Figure 2.5: RTS/CTS Handshake

2.3 Priority Access

Prioritized access is achieved by either a *reservation-based scheme* or a *priority-based scheme*. First, a reservation scheme is based on a schedule being assigned to flows. Each flow has a time-slot during which it is scheduled to transmit. The problem with this method is when an accident occurs it is important that a warning is transmitted as quickly as possible. There may not be enough time to establish a schedule before transmitting an emergency warning. Also, message scheduling is difficult to achieve in a highly dynamic environment, such as the one that exist in a VANET. Second, a priority scheme, such as the one proposed by IEEE 802.11e, varies the MAC parameters for each traffic class. Enhanced Distributed Coordination Access (EDCA) is a priority scheme based on DCF. Nodes are given prioritized access with EDCA by shortening the inter-frame space and shortening the back-off times. Because of the problem associated with scheduling in a VANET, EDCA is a likely candidate for use with DSRC.

A requirement of DSRC is that safety messages must have priority access over non-safety messages. In order to timely deliver high priority messages, such as those used by collision warning applications, DSRC adopts the Enhanced Distributed Channel Access [13, 15] (EDCA) of 802.11e.

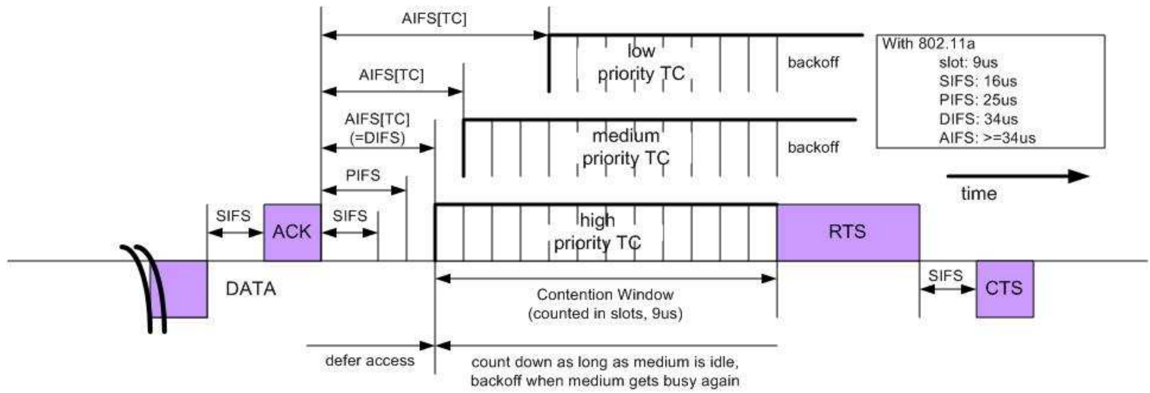

Figure 2.6: 802.11e Backoff

EDCA provides differential access to the wireless medium by assigning eight priority classes, referred to as Access Categories (AC). The AC's are labeled 0 to 7, with $AC0$ having the highest priority. EDCA functions similarly to DCF. The difference between DCF and EDCA is that EDCA uses a different set of access parameters for each AC. The AC parameters are used to set $CW_{min}[AC]$, $CW_{max}[AC]$, and $AIFS[AC]$. The parameters $CW_{min}[AC]$ and $CW_{max}[AC]$ control the minimum and maximum size of the contention window. Assigning larger values to the $CW_{min}[AC]$ and $CW_{max}[AC]$ for a low priority class increases the average amount of time that a class has to wait before transmitting, as shown in Figure 2.6. On the other hand, the inter-frame spacing is used for the duration of time that a station must wait before it can begin the back-off process. EDCA makes use of the Arbitration Inter-Frame Space (AIFS) to vary the amount of time a station must remain idle before it can decrement it back-off timer. Equation 2.2 is used to calculate the AIFS value.

$$AIFS[i] = SIFS + AIFSN[i] * slottime \tag{2.2}$$

Choosing a smaller value for $AIFSN[i]$ means that the station will be able to back-off sooner and will be able to access the channel faster. As a result of using priority access, contention is mainly between the same AC.

There is also the question of how to implement the priority queues. The DSRC standard recommends the use of four priority queues for the access categories. One thing that must be addressed is how to implement them for the different channels for DSRC. One possible implementation is to have a separate set of priority queues for each channel. Another possible implementation is to have one set of queues for the services channels and another set of queues for the control channel, simplifying the implementation. In either case, the control channel is always given priority over a service channel. For example, if a frame in a service queue and a frame in a control queue back-off timers expire at the same time; the frame from the control queue is transmitted. The control channel frame is always given priority over a service channel frame. Using EDCA ensures that high priority control channel traffic is given preferential treatment. Figure 2.7 shows a comparison of the MAC layer queues for 802.11 and 802.11e.

2.4 Broadcast

A few of authors have addressed the problem broadcasting in a VANET. Torrent-Moreno, Jiang, and Hartenstein show that the probability of reception of a broadcast message decreases as the distance from the sender increases and under saturated conditions the probability of reception messages can be as low as 20% at distances of 100 meters from the sender [22]. The primary reason that the reception rate decreases is because of the hidden terminal problem. The authors implement a priority access mechanism that improves the reception rate of broadcast messages, but still fails to

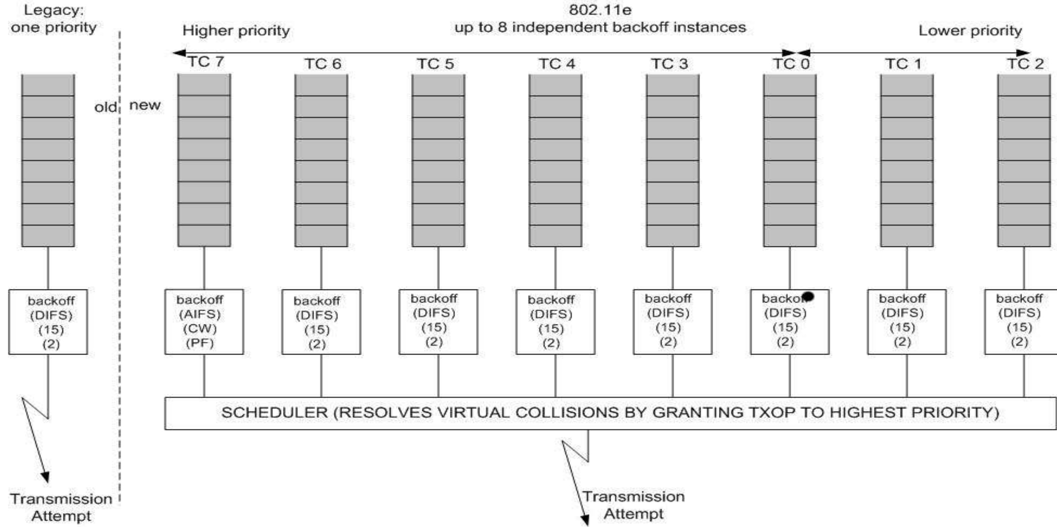

Figure 2.7: 802.11e Queues

achieve reliability anywhere near 100%. In all likelihood, it may be unrealistic to expect every node in an 802.11 based network to successfully receive a broadcast because of the hidden terminal problem.

Xu et al. [25] propose a single-hop broadcast protocol that increases the probability of a message's reception by sending the message multiple times. Yang et al. [27] propose the VCWC protocol to transmit emergency warning messages (EWM), which is based on a state machine and a multiplicative rate decrease algorithm. When an accident first occurs, the vehicle starts transmitting EMWs at the maximum rate. Over time, the transmission rate decreases for EMWs. The work does not address the of problem of increasing the reliability of low priority traffic. Another, problem with this scheme is it does not scale well if used for a multi-hop broadcast. Also, the protocol increases the amount of network traffic.

Both [25] and [27] aim at increasing the probability of reception by broadcasting a message multiple times, which increases the load on the network. Other solutions aim at assigning time slots to nodes for them to transmit during, these solution

are likely inapplicable to VANETs because it requires the synchronization of nodes which is hard to achieve because of the high mobility of nodes. Also, many of the algorithms that need to maintain sets or clusters may not perform well in VANETs because of the high mobility and the large amount of overhead that is necessary to maintain the sets.

2.4.1 Location Based Broadcast

A repetition strategy is employed for the Location Based Broadcast [25] (LBB) that transmits a frame to all vehicles within communication range of the sender. When packets arrive, it is the receiver's responsibility to determine what action to take in terms of processing the packet based on the location from the sender and the type of message. In order to reliably deliver broadcast frames, a repetition strategy is used. Each frame has a time for which the message is useful denoted by τ. The time it takes to transmit a packet is denoted by t_{trans}. The lifetime of the message is divided into $m = \lfloor \tau / t_{trans} \rfloor$ slots. The concept of flipping an unfair coin is used to determine if a node should transmit during a time slot with $p(H) = n/m$ and $p(T) = 1 - n/m$. The packet is transmitted if a head, $p(H)$, is obtained for the time slot. If one or more packets are transmitted without a collision, then the packet is successfully received. The value n is a parameter of the protocol, and it is selected so that $n < m$. The selection of the value of n is key in the implementation of the protocol. If n is selected so that a node transmits too often, then a significant amount of bandwidth is wasted. The LBB increases the probability that a frame is successfully received but, at the same time, consumes additional resources.

The protocol supports transmitting broadcast messages containing 250 bytes of data every 200 ms to 140 nodes. GPS devices typically are updated at 5 Hz, so sending a message every 200 ms should be acceptable.

One of the problems with this approach is that it is only a single-hop broadcast; it is unlikely that the protocol would support the multi-hop relaying of broadcast messages. If multi-hop relaying is used with this protocol, it would consume a waste a large portion of the network bandwidth. One approach the authors recommend to improve their protocol is to implement an adaptive control at the MAC layer.

2.4.2 Urban Multi-Hop Broadcast Protocol

The urban multi-hop broadcast (UMB) protocol [14] addresses the problem of transmitting multi-hop broadcast messages in areas where there is shadowing caused by large buildings. UMB protocol selects the furthest node from the transmitter to rebroadcast a message and places repeaters at intersections that rebroadcast the message in order to overcome the problem of large buildings obstructing a message's path. The goals of the protocol are to avoid collisions caused by hidden nodes, use the channel efficiently, make broadcast communication reliable, and disseminate messages in all directions at an intersection. The protocol assumes that all vehicles are equipped with a GPS device and an electronic map.

The first part of the protocol is the *directional broadcast* that is used to select the farthest node from the transmitter to rebroadcasts the frame. The RTS/CTS sequence of 802.11 helps alleviate the hidden terminal problem. In the case of broadcast messages, if the RTS/CTS sequence is used, as previously mentioned, a storm around the transmitter would ensue. UMB introduces an alternative to the RTS/CTS, the Request to Broadcast (RTB) and the Clear to Broadcast (CTB). Only the transmitter and farthest node from the transmitter exchange the RTB/CTB messages. When a node has a broadcast message to send, it transmits a RTB.

The network is iteratively divided into segments to determine the farthest node from the broadcaster. To determine the furthest node, each node transmits a black-

burst. When a node receives a RTB, each node computes the length of the black-burst based on their distance from the sender. The length of the black-burst is computed as follows:

$$L_1 = \left\lfloor \frac{\hat{d}}{Range} * N_{max} \right\rfloor * SlotTime \tag{2.3}$$

where L_1 is the black-burst length of the first iteration, $\hat{d}$ is the distance between the source and receiver, N_{max} is the number of segments, and $SlotTime$ is the length of a time-slot. Each node will then simultaneously start transmitting a black-burst. If a node finishes transmitting the black-burst and hears no others sending the black-burst on the medium, it affirms that it is the farthest node. A CTB is then transmitted to complete the reservation of the channel. On the other hand, if two or more nodes determine that they are the farthest away, a collision will occur when the CTB is sent. In this case, the RTB is retransmitted to the furthest non-empty segment and that segment is divided into N_{max} sub-segments. This process continues until one of the nodes CTB succeeds. The iterative black-burst is calculated as follows:

$$\begin{aligned} L_1 &= \left\lfloor \frac{\hat{d} - Llongest_{i-1} * W_{i-1}}{W_{i-1}} * N_{max} \right\rfloor * SlotTime \\ i &= 2, 3, \ldots, D_{max} \\ W_i &= \frac{Range}{N_{max}^i} \end{aligned} \tag{2.4}$$

where $lLongest$ is the longest black-burst and W_i is the segment width for the ith iteration. After a node to forward the broadcast is selected, the sender transmits the frame to the receiver. Collisions are avoided because the surrounding nodes overhear the RTB/CTB exchange and defer from accessing the channel. The receiver of the broadcast then sends back an ACK to indicate that the frame was successfully received. The receiver continues the process of relaying the broadcast message.

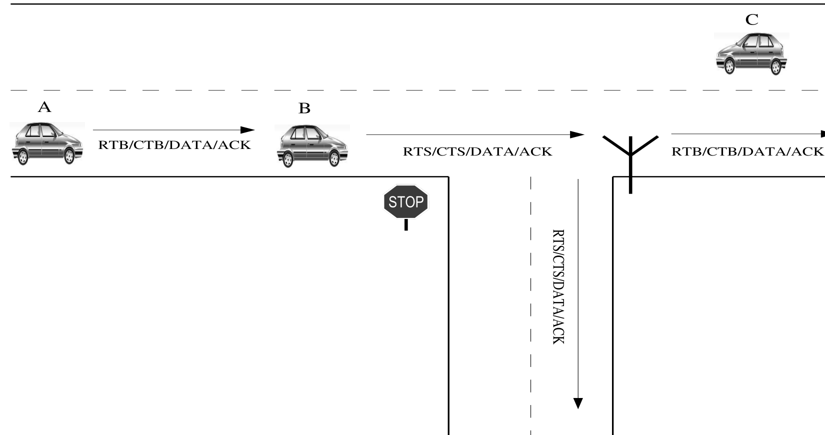

Figure 2.8: Urban Multi-Hop Broadcast Protocol

The second part of the protocol is the *intersection broadcast* that involves relaying frames by placing repeaters at intersections. If a vehicle is within range of an intersection, as determined by the vehicle's electronic map, it sends an 802.11 unicast packet to the repeater (i.e.,the RSU). Instead of using the RTC/CTB, the RTS/CTS exchange is used to transmit the packet to a RSU. The RSU will in turn relay the message in all directions except the direction from which the message was received. The sender includes directional information in the packet which prevents a RSU from rebroadcasting a message in the same direction from which it was received. The protocol also addresses the problem of loops by using a cache to determine if a packet has already been seen. Figure 2.8 shows the process of relaying a broadcast frame.

The one drawback of the protocol is it requires repeaters. It is unlikely that repeaters will be installed at every intersection. Other papers have proposed other distance based broadcast protocols, but the UMB protocol incorporates the RTB/CTB exchange to help eliminate the hidden terminal problem when multi-hop relaying is used.

2.4.3 VCWC Protocol for Cooperative Collision Warning

The Vehicular Collision Warning Communication (VCWC) protocol provides congestion control, service differentiation, and a method for propagating emergency message warnings [27].

A communication collision warning protocol is either passive or active. First, a passive protocol requires each vehicle to frequently broadcast its state to other neighboring vehicles. Each vehicle then uses the collected state information, from the surrounding vehicles, to determine dangerous conditions. The drawback of using a passive protocol is the network is always saturated with safety messages. Second, an active protocol only sends messages when an emergency event occurs. For instance, an emergency warning message (EWM) is sent if a vehicle decelerates abruptly. VCWC uses an active approach to send cooperative collision warnings.

A number of problems arise in vehicular communication systems. To begin, the wireless links are unreliable. Next, a vehicle becomes an abnormal vehicle (AV) when an event such as abrupt deceleration occurs. When a vehicle transitions to the AV state it should send EWMs. The surrounding vehicles should receive emergency messages as quickly as possible, so the driver has time to react. Also, a communication protocol must share the channel with other applications. The channel will not be used just to send emergency warnings. When an emergency event occurs, the surrounding vehicles can also become abnormal vehicles and generate their own emergency message warnings (as a result of reacting to the situation). The network should support many simultaneous emergency messages. For instance, when an emergency event occurs a chain reaction may occur, where vehicles behind the vehicle also generate their own EWMs. In this case, the initial vehicle should stop sending EWM if the vehicles behind it are also generating their own EWMs.

Message differentiation is used because both time-sensitive and non-time-sensitive messages contend for the channel. The authors distinguish three classes of messages: class 1 emergency warning messages (EWM), class 2 forwarded EWM, and class 3 non-time-sensitive messages. Differentiation between the message classes is achieved by using different inter-frame spacing and contention window sizes. Service differentiation allows high priority messages faster channel access.

The focus of VCWC is providing congestion control for emergency warning messages. In order for the network to remain stable congestion control is needed. When an accident or an emergency arises a large number of emergency messages are generated. To overcome this problem, the rate EWMs are generated at decreases over time. Also, the protocol uses state transitions for abnormal vehicles.

A multiplicative rate decrease algorithm is used to limit the number of messages sent by an AV. The emergency warning rate after the k_{th} transmitted EWM is calculated with the formula below. The parameters a and L are fixed parameters; there values from the results of their simulations. The parameter λ_0 is the initial transmission rate for emergency messages.

$$f(\lambda_0, k) = max\left(\lambda_{min}, \frac{\lambda_0}{a^{\lfloor k/L \rfloor}}\right) \tag{2.5}$$

When an emergency event first occurs, the AV transmits warning messages at the greatest rate. Over time, the timing between consecutive warnings sent by a vehicle is lengthened, based on the assumption that surrounding vehicles have already received the message. The minimum transmission rate should be set so that when a vehicle enters the transmission range of the abnormal vehicle it will have time to react. The time needed to react by approaching vehicles is much longer than the time needed to initially react to the situation. By reducing the rate that emergency warnings are

being sent, more warnings can be sent from larger number of vehicles concurrently.

Congestion control is also achieved by state transitions of the abnormal vehicles. There are three states that a vehicle can be in: the initial state, flagger AV and non-flagger AV. A vehicle transitions to the initial AV state when emergency occurs. In the initial AV state the vehicle will begin broadcasting EWMs at the maximum rate and then start decreasing the EWMs with the multiplicative decrease algorithm. A vehicle will transition from the initial AV state to the non-flagger AV state if Talert time period has expired and an EWM is overheard from a vehicle directly behind it. Talert is the initial amount of time that a message is broadcast, so that there is a high probability that the message is heard by others. In the non-flagger state the vehicle refrains from sending EWMs. The vehicle then keeps a timer FT and resets the timer each time it hears an EWM. If the timer expires and no EWM is heard the vehicle transitions to the flagger AV state. In the flagger AV state, the vehicle transmits EWMs at a minimum rate. When a vehicle in the flagger AV state overhears messages from following vehicles it transitions to a non-flagger state.

When an event occurs that triggers the sending of EWMs, the messages must be disseminated to as many vehicles as possible in order to avoid the dangerous situation. Messages are disseminated in two ways. First, a vehicle reacts to the situation and generates its own EWMs. Second, a forwarding protocol is used. The forwarding mechanism should limit how far messages are transmitted.

The VCWC protocol uses an active approach to detect abnormal driving situations. In reality the active approach wont help to prevent all accidents. If a vehicle is gradually decelerating while the vehicle behind it is gradually accelerating, neither of these vehicles may reach a threshold to send an EWM. It may be possible, that vehicles could collide with each other and never receive an alert. The active

approach would prevent some accidents from taking place, but it may not prevent all accidents from occurring. The protocol also doesn't address the hidden terminal problem. The VCWC relies on an abnormal vehicle sending warning messages at a high rate to compensate for collisions and the hidden terminal problem.

2.4.4 Multi-Hop Broadcast

In a wireless ad hoc network, the dissemination of information to an area greater than the transmission range of a node requires multi-hop relaying. The simplest way to perform multi-hop relaying is by flooding a packet. When flooding is used, a node that receives a broadcast message for the first time re-transmits the message. The node then ignores all subsequent rebroadcasts coming from other nodes. There are three problems associated with flooding. First, a number of redundant rebroadcasts occur because of flooding. To illustrate how serious this problem is when a message is sent to n hosts, the packet is transmitted n times. Second, contention occurs resulting in the high probability that a message is rebroadcasted by many hosts in a close proximity. Each node tries to retransmit the message at the same time. Each node severely contends with each other for access to the medium. Third, a large number of collisions occur because of the lack of RTS/CTS and because of the absence of collision detection. The authors of the paper term the combination of these problems the "broadcast storm problem".

The significance of the three problems related to broadcast storm problem were evaluated, and to determine the seriousness of the problem. To begin, a rebroadcast of a message provides only $0 \sim 61\%$ additional coverage. On the average, a rebroadcast covers only an additional 41% of area. The additional coverage dramatically decreases as the number of times k that a message is rebroadcasted. When $k \geq 4$, the additional expect coverage is less than 0.05%. Also, the contention is expected

to be higher as the number of nodes n increases. The probability of all n nodes experiencing contention increases rapidly to 80% when $n \geq 6$. The results show, the denser a network is the less likely a node is able to access the medium without experiencing contention. Last, the number of collisions that occur is high for a number of reasons, such as, the PCF is not available, the RTS/CTS exchange cannot be used, and collision detection is not used.

There are two possible solutions to reduce the effects of the broadcast storm problem. First is to reduce the probability of rebroadcasting a message. Second, is to differentiate the timing of rebroadcasts. There are five possible schemes that are proposed to alleviate the broadcast storm problem. First, a probabilistic scheme aims to limit the number of rebroadcasts. When a node receives a broadcast for the first time, the message is rebroadcast with a probability p. Second, counter-based broadcast is used to prevent a rebroadcast when the expected additional coverage (EAC) is low. The authors show that when $k \leq 4$ the additional coverage of a rebroadcast drastically decreases. A counter is used to keep track of the number of times that a message is overheard before a node gets a chance to broadcast the message. The counter base scheme prohibits the rebroadcast when $c \geq C$, with c the being the number of times a broadcast has been overheard and C is the counter threshold. Third, the distance-based scheme determines whether to rebroadcast a message based on a receiving node's distance from the sender. The variable D_{min} is used to record the distance between the sender and receiver of a broadcast. If D_{min} is less than the D threshold value, the broadcast is prohibited from being transmitted. The distance between the sender and receiver is determined by the transmitted and received power. Fourth, a location-based scheme allows the coverage area to be calculated with more precision than the previous schemes. A GPS device records

the points used in the broadcast. If the additional coverage of a broadcast is greater than a predetermined threshold the message is rebroadcast. One possible solution to calculate the additional coverage area is based on convex polygons. The last scheme is the cluster-based scheme, where the network is partitioned into clusters.

The authors introduce five schemes that improve on simple flooding. Some of the schemes' performances rely on the topology of the network; some of the schemes perform poorly in sparse networks. A simple counter-based scheme offers a tremendous improvement over flooding. The location-based scheme performs the best under all situations.

CHAPTER III

Methodology

The objective of the media access control layer is to arbitrate access to the shared medium, the wireless channel. If no coordination method is used, excessive collisions may result. The ideal MAC prevents nodes within transmission range of each other from accessing the channel at the same time and, therefore, collisions on the channel are avoided. In addition, the media access control must be fair, efficient, and provide the ability to prioritize traffic.

To reduce the problem associated with unreliable broadcasts in a VANET, modifications to the 802.11 broadcast protocol are necessary to improve the reception rate of broadcast traffic. As shown in [22], the reception rate of broadcast traffic is affected by both the size of the inter-frame space and the contention window. For instance, by selecting a smaller value for the time needed to wait before placing a frame onto the channel (IFS), a node is able to improve its reception rate. Using this approach, high-priority traffic is given a higher probability of being received than low-priority traffic. Furthermore, broadcast messages, such as emergency warnings, require that all vehicles receive them with the smallest delay possible. By assigning high priorities and shorter inter-frame space to emergency warnings, the likelihood of emergency warnings being received increases and the delay decreases.

The network conditions of a VANET favor a distributed QoS scheme such as EDCA of 802.11e. As a result of using 802.11e and using different $AIFS[i]$ values, collisions are likely between flows of the same access category (AC). If the number of contending flows of equal priority is substantial, the chance of a collision occurring drastically increases. On a crowded highway, the number of vehicles contending to access the wireless medium can become very large. For instance, on a gridlocked 4 lane highway, with vehicles placed 15 m apart, approximately 300 cars or more could be attempting to transmit (e.g., 600 m diameter / 15 m between vehicles * 4 *lanes* * 2 *directions* $\approx$ 320 vehicles).

One drawback of using EDCA is the MAC parameters do not adapt to the changing conditions of the network. In other words, the number of contending vehicles attempting to access the channel varies over time, but the MAC parameters do not change. In situations where a large number of vehicles are contending for access to the medium, it is beneficial to increase the initial size of the contention window to reduce the probability of a collision. For example, if the initial size of the CW is 15 time-slots (i.e., $CW_{min} = 15$), as a result, the back-off timer is randomly selected between 0 and 15 time-slots. Consequently, a node will always wait [0,15] time-slots before transmitting.

As the network traffic increases, the size of the contention window should also increase to adapt to an increased number of nodes trying to access the channel. For example, increasing the initial size of the CW from 15 to 31 slots can be used to reduce the probability of a collision on the channel as the amount of network traffic increases. If the channel traffic continues to increase, the CW can be increased from 31 to 63 time-slots. In a similar manner, the size of the CW can continue to increase until the upper bound of the contention window is reached (i.e., $CW = CW_{max}$).

Vehicles can also benefit from the opposing situation where the size of the contention window is reduced to account for light traffic.

DSRC use an algorithm that exponentially increases the CW, after each failed transmission, to adapt to the changing conditions of the network for unicast transmissions. However, in the case of broadcast transmissions, the CW does not change. Because broadcast transmissions suffer from the "ACK explosion problem", it is impossible to determine if broadcast frames are successfully received or not. As such, it is not possible to adjust the transmission parameter based upon the previous success or failure of a transmission. Also, an additional obstacle is the fact that most of the previous work aiming at dynamically adjusting the initial value of the contention window is based on the previous success or failure of unicast transmissions. The nodes within a VANET must use some other means in order to determine the amount of congestion/contention in the network since they cannot rely on receiving acknowledgments to confirm the successful delivery of a broadcast frame.

Two approaches can be used to receive feedback from the network. First, an active approach can be used where nodes monitor the network and exchange information with neighboring nodes. To illustrate, nodes may exchange a list of all their one-hop neighbors with other nodes in the immediate proximity. The drawback to the active approach is that it increases the network overhead. Consequently, the active approach results in the consumption of additional bandwidth. In a VANET, bandwidth is a scarce resource, so it is desirable to reduce the number of messages. The active approach increases the number of messages sent and leads to additional congestion. Second, a passive approach can be used to receive feedback. In a wireless network, a node overhears all transmitted messages within its communication range. For example, if all nodes have a transmissions range of 100 m, a node overhears all

messages sent from within 100 m of it. Nodes can receive feedback from the network simply by listening to the messages sent from other nodes. For example, nodes are able to record statistics concerning the condition of the network based on the overheard packets. The benefit of the passive approach is that it requires no additional network resources. As such, it is preferable to use a passive approach for feedback.

The rest of of Chapter III describes a protocol that passively monitors the network to dynamically adjust the MAC layer parameters. The goal of this dynamic protocol is to increase the reception rate of broadcast messages. To begin, Section 3.1 describes the technique used to monitor the network conditions. Section 3.2 contains a description of the data structure used to record the network conditions. Next, Section 3.3 explains the algorithm used to adjust the contention window. Finally, Section 3.4 describes how prioritized access is implemented for the algorithm.

3.1 Passive Monitoring

According to the DSRC standard, each vehicle broadcasts its status to its neighbors approximately 10 times every second [23]. As such, a node in a VANET is able to detect collisions and congestion by analyzing the sequence numbers of the packets it has recently received. While a node does not know if the packets it sent are correctly delivered, it can estimate the percentage of packets sent to it from its neighboring nodes that are successfully received. Based on the observation of packets that are recently received, a node is able to determine the current local conditions of the network. Therefore, a node is able to dynamically adjust its transmission parameters.

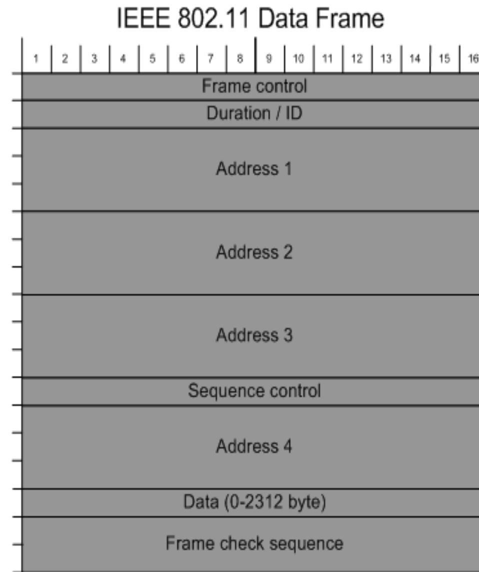

Figure 3.1: 802.11 MAC Header

3.1.1 Sequence Numbers

A two-byte sequence control field is contained in an 802.11 MAC header. Figure 3.1 displays the fields that make up the 802.11 MAC header. The sequence control field consists of two subfields, a 12-bit sequence number and a 4-bit fragment number. First, each frame passed down to the MAC is assigned a 12-bit sequence number. The sequence number was originally intended to detect duplicate frames. For example, a node can use the sequence number field to determine if a frame can safely be discarded because it is a duplicate. In effect, the sequence number acts as a modulo-4096 counter. The sequence numbers are incremented by one for each frame passed down to the MAC. Second, the fragment number is used for defragmentation. If a packet passed down from a higher layer must be fragmented, each frame will contain the same sequence number, but each fragment will be assigned its own fragment number.

Since the sequence control field already exists, it will be used to detect colli-

sions/congestion in the network. In other words, the modified broadcast protocol will analyze the overheard sequence numbers to determine the state of the network.

3.1.2 Monitoring Network Traffic

While it is not possible to detect the collision of a broadcast frame, it is possible to record the successful delivery of frames. In a VANET, each node will periodically broadcast its status to its neighbors every 100 ms. By simply analyzing the packets a node has recently received, a node is able to detect collisions and congestion. Thus, a node can determine the network conditions. For example, a node will be able to calculate the exact percentage of packets sent to it from neighboring nodes that are successfully received, and it can then roughly estimate the number of neighbors in its communication range.

A node is able to hear all transmissions placed on the wireless channel. As a result, each node records the overheard sequence numbers. As shown in Figure 3.2, node A records that it has overheard the frames coming from node B with the sequence numbers 32, 34, 35, 36, 37, 38, 40, 41. Based on the observed sequence numbers, node A can conclude that frame 33 and frame 39 were corrupted or lost and that 20% of the frames sent from node B were not received by node A. Also, node A is able to record all the sequence numbers that are overheard from node C, D, and E. Similarly, node A can conclude that three frames from node C, two frames from node D, and one frame from node E were corrupted or lost. Therefore, the percentage of packets sent from neighboring nodes that were corrupted in the time period is 20% (8 out of 40), and four nodes are currently in its communication range. This collision rate is used as an indication of congestion/contention in the distributed network.

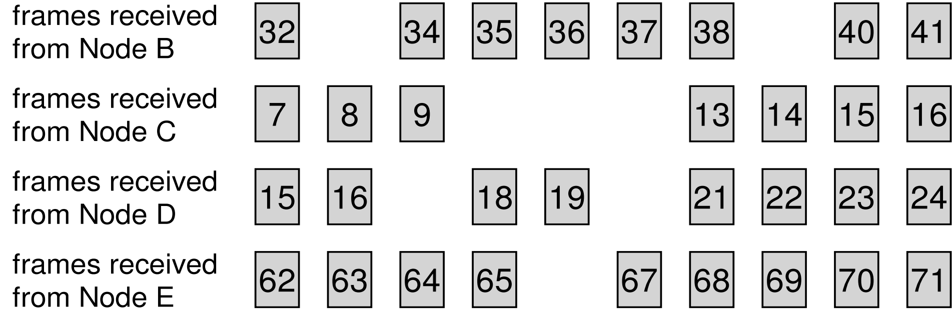

Figure 3.2: Broadcast Frames Received

Table 3.1: Data Maintained in a Broadcast Table

MAC Address	Sequence Number	Reception Rate	Time Stamp

3.2 Broadcast Table

As stated in Section 3.1, each node passively records the sequence numbers overheard from specific MAC addresses. Consequently, each node must maintain a data structure to record this information. One possible data structure that can be used is a hash table. The benefit of using a hash table is that lookups can be performed in near constant time (i.e., $O(1)$).

3.2.1 Hash Table

The condition of a VANET favors a data structure where lookups occur more frequently than insertions into the data structure. As a result, each node maintains a hash table that it uses to record broadcast messages. For each overheard message, a node will update the table entry for the specific source address.

A dynamic hash table is used so that an entry is updated in near constant time and so that the size of the table grows and shrinks according to the number of immediate nodes in the network. The MAC address is used as the key to the hash function. The entries maintained in the hash table are given in Table 3.1.

The hash table is used to store additional information in addition to the last overheard sequence number. To begin, the table contains the MAC address for each node. The MAC address is used to uniquely identify each node in the table. Next, the table records the sequence number of the last packet overheard from a node. Also, the reception rate is stored in the hash table. The reception rate is a weighted average and is used to determine the percentage of packets that are successfully received from a specific node. Section 3.3 contains a detailed description of how the reception rate is calculated. Finally, the table maintains a time stamp for each node in the table. The time stamp records the time that a transmission was overheard from a node.

3.2.2 Invalidating Old Hash Table Entries

In order to prevent stale data from affecting the calculation of the local network conditions, old entries are periodically removed from the table. If a broadcast has not been received from a node within a timeout threshold, the entry is removed from the table under the assumption that the node is no longer within the transmission range.

The timeout threshold should be carefully selected so that nodes that are still within the transmission range but are suffering from a high collision rate are not prematurely invalidated from the table. On the other hand, the timeout value should be short enough so that irrelevant data is not maintained in the table. For example, if the timeout threshold is set to 1 s and nodes exchange their status every 100 ms, than ten transmissions in a row would have to fail to falsely remove a node from the hash table. Periodically, when a timer expires, nodes are removed from the table after it is safe to say that they have moved out of the transmission range.

3.3 Adaptive Adjustment of the Contention Window

As discussed in Section 3.2 each node maintains a hash table. Within the hash table, an entry is used to record the reception rate for a node (i.e., the percentage of packets successfully received). The reception rate of a node is the critical component that determine whether the size of the contention window should be adjusted.

3.3.1 Estimated Reception Rate

To determine the reception rate, a weighted moving average is used to calculate the value of the $EstimatedReceptionRate$. The sequence numbers are the decisive factor in determining the reception rate. To determine the $EstimatedReceptionRate$, the difference between the received sequence numbers are examined. If, for instance, the previous sequence number overheard was 132 and the node just received sequence number 135, then the node could conclude that frames with the sequence numbers 133 and 134 were not received. The difference between the sequence numbers, in this case, is three $(135 - 132 = 3)$, so this would indicate that two messages were lost.

In a highly dynamic network such as a VANET, the emphasis should be placed on the most recent conditions of the network, for this reason a weighted average is used. To calculate the $EstimateReceptionRate$, an approach similar to the TCP's round-trip-time is used, and it is calculated with Equation 3.1.

$$EstReceptionRate = \alpha * EstReceptionRate + (1 - \alpha) * SampledReceptionRate \tag{3.1}$$

The $SampledReceptionRate$, in Equation 3.1, will contain either the value of a one or a zero. A one is used if a message was correctly received for a sequence number, and a zero is used if the message for a sequence number was not received. If the gap between the sequence numbers is larger than 1, the $EstimatedReceptionRate$ is calculated multiple times. For example, if the difference between sequence is three

(i.e., 135 - 132 = 3) the average would be calculate twice using a value of zero (to indicated the message was not received) for the $SampledReceptionRate$ and then a final time using a one for the $SampledReceptionRate$. If the gap between sequence numbers is greater than one, the equation is used for multiple iterations.

The value α, in Equation 3.1, controls how quickly the $EstimatedReceptionRate$ reacts to the changing condition of the network. As the value of α moves closer to 1.00 less weight is placed on the current network conditions. On the other hand, as the value of α moves towards 0.00, more weight will be placed on the current network conditions. For instance, if $\alpha = 0.85$, the previously recorded $EstimatedRecptionRate = 0.91$, and there is a gap of two between the sequence numbers than the new $EstimateReceptionRate = 0.807$. The following is a sample of the two iterations needed to calculate the $EstimateReceptionRate$:

$$
\begin{aligned}
0.7735 &= 0.85 * 0.91 + 0.15 * 0 \\
EstimateReceptionRate &= 0.7735 \\
0.8075 &= 0.85 * 0.7735 + 0.15 * 1 \\
EstimateReceptionRate &= 0.8075
\end{aligned}
$$

3.3.2 Update Timer

Each node also maintains an update timer and a variable, *update_interval_*. The variable *update_interval_* controls how often the timer, that a node mantains, expires. When the timer expires, a node will determine the condition of the network and accordingly adjust the transmission parameters. For example, it is possible to set the *update_interval_* equal to 0.5 s. If a timer expires at intervals of 0.5 s, then every half of a second a node adjusts its transmission parameters. When the *update_interval_* expires, a node will scan through the table it maintains to deter-

mine the local conditions of the network. The value of *update_interval_* determines how often the transmission parameters are adjusted for a node. It may be beneficial to adjust the transmission parameters more or less frequently.

3.3.3 Local Reception Rate

Based upon the information collected in the table, a node is able to adjust its transmission parameters on the expiration of the timer. When the timer expires, a node will already have the reception rates calculated for each node. The node will then calculate the *LocalReceptionRate* used to predict the network condition. The *LocalReceptionRate* is the average of the *EstimatedReceptionRates*. In other words, an average of the reception rates is calculated.

As previously explained, the *EstimatedReceptionRate* is determined for each node whenever a frame is received. On the other hand, an average of the reception rates is used to determine the *LocalReceptionRate*, and it is only calculated periodically. Equation 3.2 is used to calculate the *LocalReceptionRate*.

$$LocalReceptionRate = \sum EstRecptionRate/NumberofNodes \tag{3.2}$$

3.3.4 Contention Window Adjustment

Once a node determines the *LocalReceptionRate*, it compares the value against the previously stored *LocalReceptionRate* in order to adjust the size of the CW. The following comparison is used to adjust the CW:

```
IF (average - previous average >= threshold)
    Decrease the window
ELSE IF (-(average - previous average) >= threshold)
    Increase the window
ELSE
```

```
Maintain the current window
```

For example, if the timer expires every 0.5 s the algorithm attempts to adjust the size of the CW every 0.5 s. In addition, the value of the *threshold*, in the algorithm, is fixed. Some possible values for the *threshold* are 0.03, 0.05, and 0.07. If the *threshold* is set to 0.03, the algorithm will react more quickly to changing network conditions than it would if it was set to 0.07. In other words, the *threshold* determines the change that must occur in the local reception rate between consecutive measurements to cause the contention window to change size.

3.3.5 Contention Window Sizes

The size of the contention window may either increase or decrease. One question that remains unanswered is how much will the size of the contention window change? The algorithm exponential increases the window size (i.e., $CW_{new} = 2 * CW_{old} + 1$), in the same manner as the IEEE 802.11 algorithm does. In other words, to increase the size of CW, the size of the window doubled. For example, if a node is initially transmitting with a CW equal to 15 and the algorithm determines that the window must be increased, then the new CW is 31. The node will then randomly set a backoff in the range of $[0, 31]$. If, when the next $LocalReceptionRate$ is calculated, it is determined that the window must be increased again, the new value for the CW will be 63. The CW will continue to increase until the maximum size of the window is reached (i.e., $CW = CW_{max}$).

If the size of the contention window needs to be reduced, the CW is cut in half (i.e., $\lfloor CW/2 \rfloor$). If the current CW is 31 and the window must be decreased, the new value for the CW is 15. The size of the contention window is able to continue to decrease until the minimum value is reached (i.e., $CW = CW_{min}$).

Table 3.2: Contention Window Sizes

7	15	31	63	127	255	511	1023

The size of the CW will continuously fluctuate based on the condition of the network. As such, the CW will either double, be cut in half, or the current window will be maintained. Table 3.2, contains the typical sizes that are used for a contention window. The primary difference between this algorithm and the standard 802.11 CW algorithm is that for the broadcast algorithm the size of the contention window remains fixed until the next time the $LocalReceptionRate$ is calculated. The standard 802.11 protocol, however, may use a new value at any time for the contention window for a unicast traffic. In the case of the broadcast algorithm, a node maintains a contention window size until the update timer expires. As a result, if a node currently has a $CW = 127$, the node will always select back-off in the range of $[0, 127]$ until the next update.

3.4 Prioritized Access (QoS)

Safety-related applications are the distinguishing applications of a VANET, and they are the preeminent applications on the control channel. The DSRC standard provides only one control channel; this channel is used both for exchanging different types of safety messages and service announcements. Since various types of messages are present on the control channel, the packets on this channel should be prioritized according to the quality of service (QoS) requirements of the application that is generating the packets. As a result, each frame is assigned an access category (AC) before it is placed on the channel, and Enhanced Distributed Channel Access (EDCA) [26], of 802.11e, is used to provide prioritized access to the wireless channel. Quality of service is achieved through different choices of inter-frame spacing and contention

Table 3.3: Access Categories

Access Category	Message Type
0	Emergency Warning Message (EWM)
1	Emergency Vehicle Approaching (EVA)
2	Periodic Broadcast Message (PBM)
3	Service Advertisement Message (SAM)

window sizes, as explained in Section 2.3

3.4.1 Access Categories

A number of different access categories are possible. Table 3.3 lists some of the possible access categories that may be used in a VANET. When VANETs are deployed, it is likely that the traffic classes on the control channel will be similar to those presented in Table 3.3. In this case, AC-0 is the highest priority access category, and AC-3 is the lowest priority. With 802.11e, it is possible to have up to eight different access categories. So in the future, the differentiation of additional types of messages is possible.

3.4.2 Control Channel Applications

To begin, AC-0 is the highest priority access category and is used to transmit Emergency Warning Messages (EWM). When an abnormal event occurs, EWMs are transmitted to warn surrounding vehicles of the dangerous road condition. For instance, EMWs are transmitted if a vehicle decelerates abruptly and a threshold is reached. Also, EWMs are transmitted if an accident occurs. Emergency warning must be delivered with the shortest delay. For this reason, EWMs are given the highest priority. Second, Emergency Vehicle Approaching (EVA) messages, AC-1, are issued by emergency vehicles such as fire trucks, ambulance, or police cars, when responding to an emergency. These messages are used for purposes such as alerting other drivers that public safety vehicles are approaching or coordinating traffic lights

so that emergency vehicles do not get delayed waiting on red lights when responding to an emergency. Third, each vehicle periodically broadcasts its state (e.g. location, direction, speed, acceleration, etc.) in the form of a Periodic Broadcast Message (PBM) to its neighbors. The PBMs, AC-2, are used so that vehicles are able to avoid emergencies or unsafe situations even before they appear. These messages are used to create a snap-shot of the surrounding vehicles. Periodic broadcast messages will likely account for the largest percentage of traffic on the control channel. As stated earlier, every 100 ms a vehicle attempts to transmit a PBM. Fourth, Service Advertisement Messages (SAM), AC-3, are a class of traffic that are transmitted from a RSU to announce the availability of a value-added service. When a vehicle finds a service of interest, it switches to one of the service channels to use the service. To conclude, these access categories are not part of the DSRC standard. These access categories and possible control-channel applications were provided to show the need for service-level differentiation between traffic classes.

3.4.3 Arbitration Inter-Frame Space

The EDCA protocol requires that any node attempting to access the channel must wait for the channel to remain idle for the duration of the arbitration inter-frame space (AIFS) before starting its back-off timer. Instead of using a single value for IFS (as is done for 802.11) a separate value $AIFS[AC]$ is maintained for each access category. If high-priority traffic is assigned less time-slots in which to wait for $AIFS[AC]$, a node with high priority traffic is able to start decrementing its back-off timer sooner.

The value of $AIFS[AC]$ is defined as the number of time-slots to wait for in addition to $SIFS$. For example, if $AIFS[0] = DIFS$ which also equals $SIFS + 2 * \textit{slot-time}$. To further illustrate, if $SIFS = 32\ \mu s$ and $\textit{slot-time} = 13\ \mu s$ than

$AIFS[0] = 58\ \mu s$, so a node attempting to transmit an AC-0 packet would have to wait 58 μs before beginning to back-off. If emergency warnings, AC-0 were assigned $AIFS[0] = SIFS + 2$ * *slot-time* and all other access categories assigned $AIFS[AC]$ $= SIFS + 3$ * *slot-time* or more, than emergency warning application could possibly seize the channel before any other application was able to start the back-off. As a result, high-priority traffic may be placed on the channel before another access categories $AIFS[AC]$ timer even expires for low-priority traffic. As such, a message with a shorter time length for $AIFS[AC]$ is able to start its backoff timer sooner and access the channel quicker.

3.4.4 Access Category Contention Window Sizes

The size of the contention window is also used to differentiate an access category. After $AIFS[AC]$ has expired, a backoff procedure is invoked and a back-off counter is randomly chosen from the range of [0, $CW[AC]$), where $CW[AC]$ represents the contention window size. The backoff timer corresponds to the number of idle slots the sender has to wait before accessing the channel. At any instance in time, the size of $CW[AC]$ ranges from $CW_{min}[AC]$ to $CW_{max}[AC]$. When a node is first initialized and becomes active, the node will begin by selecting a backoff timer within the range of [0, $CW_{min}[AC]$). Over time, as the local network conditions are evaluated, the range of values used to select the backoff timer will increase or decrease, depending on the state of the network. The size of the $CW[AC]$ is able to increase up to the value of $CW_{max}[AC]$. So, if the $CW_{max}[1]$ equals 63, the maximum size of the backoff timer for AC-1 is 63 time slots. If $CW_{min}[1] = 15$, the size of the contention window, for AC-1, will always be in the range of $[15, 63]$.

As discussed in Section 3.3, the size of the contention window is adjusted based on the percentage of frames successfully received. When the size of the contention

Table 3.4: Access Categories

AC	$AIFS[AC]$	$CW_{min}[AC]$	$CW_{max}[AC]$
0	2	7	15
1	2	15	63
2	3	15	511
3	4	31	1023

window increases, each access categories contention window increases simultaneously as long as the new value for $CW[AC]$ does not exceed $CW_{max}[AC]$. For example, if a large number of collisions were recently experienced, the size of the all the contention windows will increase to account for the congestion. On the other hand, if the number of collisions detected is below a threshold, then $CW[AC]$ decreases. Therefore, $CW[AC]$ can decrease until $CW[i]_{min}$ is reached.

The scaling factor, $SF[AC]$, determines how much $CW[AC]$ either increases or decreases. If $SF[AC]$ is set equal to 2, then the contention window, $CW[AC]$, will increase as it would with the traditional 802.11 protocol. If on the other hand, $SF[AC]$ is set equal 1.5 the size of the contention window, $CW[AC]$, will take longer to react to changes in the network conditions. The default value for $SF[AC]$ is 2 for all of the access categories, but by enabling this feature is possible to also differentiate how much the contention window is increased for an access catergory.

Figure 3.3 contains a sample of the of parameters maintained at a node for each access category. Through message differentiation, not only can higher priority messages access a channel faster than lower priority messages, but also collisions between access categories are avoided to a large extent.

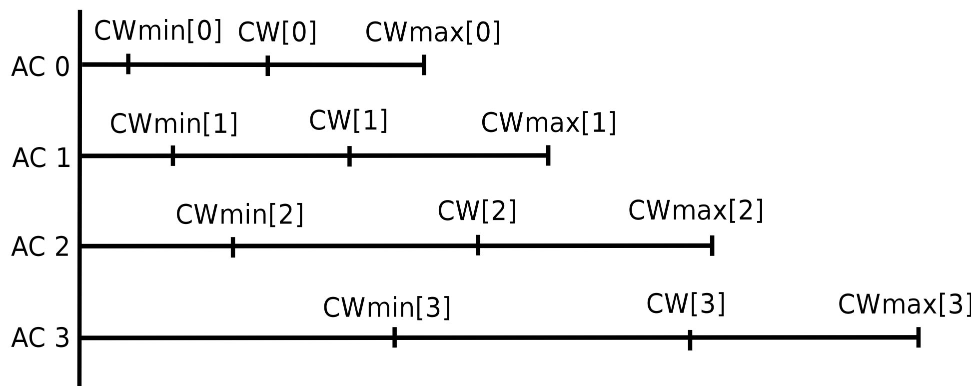

Figure 3.3: Contention Window Sizes

CHAPTER IV

Simulation and Results

This chapter examines how the protocol, proposed in Chapter III, was simulated along with the simulation results. To begin, Section 4.1 contains a description of, ns-2, the network simulator used to evaluate the broadcast protocol. Next, Section 4.2 explains the mobility model used in the simulations and the programs used to create vehicular movement for the simulations. Section 4.3 characterizes the network traffic present during the simulations as well as the programs used to generate this traffic. Next, Section 4.4 formalizes the metrics used to evaluate the simulations along with the programs used to extract the data from a simulation trace. In addition, Section 4.5 expounds on the implementation of the broadcast protocol for ns-2. Finally, Section 4.6 explains the results of the simulations.

4.1 The Network Simulator ns-2

To simulate the proposed broadcast protocol and to compare the performance against the the existing 802.11 broadcast algorithm, the network simulator ns-2 [2, 1] was used. Ns-2 is a discrete-event simulator developed by the University of California at Berkely, and the simulator can be downloaded from SourceForge.Net. A discrete-event simulator changes the state of a model (i.e., the wireless network) at only discrete, but possibly random, points in time [21].

The ns-2 simulator has a large user-base from academia, and it is the software of choice for researching wireless networks. The primary reasons for ns-2's popularity is its open source nature (i.e., it is published under GNU General Public License) and the ease with which the simulator can be extended. A large number of network protocols and routing algorithms have been implemented for ns-2, and users are free to create their own protocols to evaluate. In addition, extensive documentation exists for ns-2, and the documentation is available from various user web sites, "The ns Manual" [4], and the ns-users mailing list [3].

The advantage of a network simulator, such as ns-2, is that any potential protocol can be thoroughly tested before it is commercially deployed. For instance, various scenarios can be evaluated to determine if the proposed network protocol's performance is sufficient under diverse conditions.

4.1.1 NS-2 Architecture

The ns-2 simulator uses both the C++ and OTcl programming languages. To begin, OTcl language was developed at MIT to provide object-oriented extensions to the Tcl scripting language. The main advantages of OTcl is its ease of use and its ability to rapidly develop applications. In contrast, C++ is a compiled, object-oriented language created by Bjarne Stroustrup at Bell Labs. The chief advantage of C++, over OTcl, is the efficiency of the language. Because each of these languages excels in a different aspect, the OTcl and C++ languages are both used to control specific aspects of the simulations.

The network simulations of ns-2 are controlled by user-created simulation scripts written with OTcl, the high-level scripting language. On the other hand, the event scheduler and the basic network component objects are written with C++, and these network components are compiled to improve efficiency. The compiled objects,

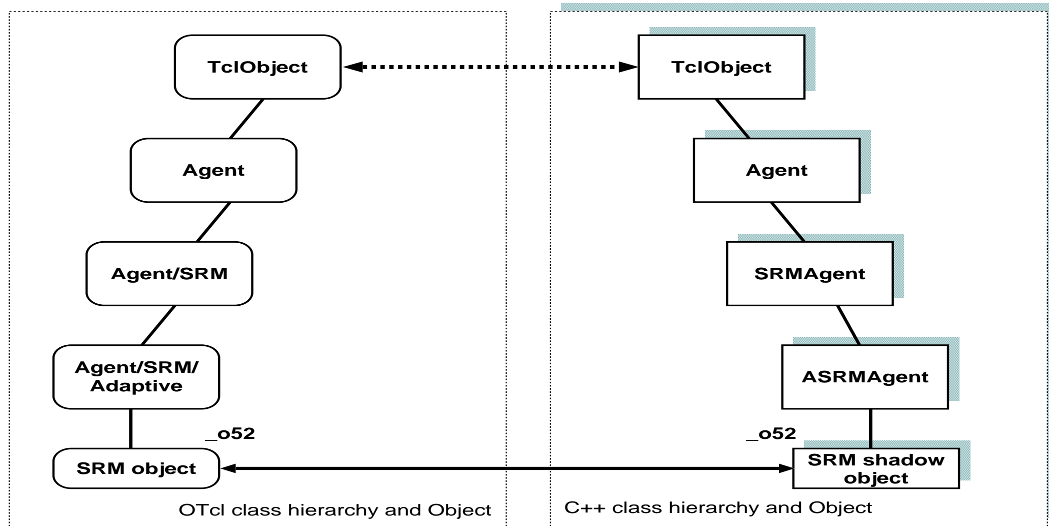

Figure 4.1: Mirrored NS-2 Objects

written in C++, are made available to the user through OTcl linkage. For each object created in C++, a mirrored OTcl object may exist. Figure 4.1 contains a sample of mirrored ns-2 objects [4]. The configuration variables and control function of a C++ object are made available to a user through a mirrored OTcl object. In contrast, objects written in C++ that do not need to be controlled from a simulation script do not require OTcl linkage.

A user who is only concerned with simulating existing protocols, such as those protocols already implemented for ns-2, only needs to use OTcl to create simulation scripts. On the contrary, in order to simulate most new network protocols, a user generally must use both C++ and OTcl. To simulate most new protocols, the user implements the protocol with the C++ language.

The following explanation from [9] describes the process of setting up a simulation script for ns-2:

> In a simplified user's view, NS is Object-oriented Tcl (OTcl) script interpreter that has a simulation event scheduler and network component

> object libraries, and network setup (plumbing) module libraries (actually, plumbing modules are implemented as member functions of the base simulator object). In other words, to use NS, you program in OTcl script language. To setup and run a simulation network, a user should write an OTcl script that initiates an event scheduler, sets up the network topology using the network objects and the plumbing functions in the library, and tells traffic sources when to start and stop transmitting packets through the event scheduler. The term "plumbing" is used for a network setup, because setting up a network is plumbing possible data paths among network objects by setting the "neighbor" pointer of an object to the address of an appropriate object. When a user wants to make a new network object, he or she can easily make an object either by writing a new object or by making a compound object from the object library, and plumb the data path through the object. This may sound like complicated job, but the plumbing OTcl modules actually make the job very easy. The power of NS comes from this plumbing.

The Architecture of ns-2 is shown in Figure 4.2 [9].

4.1.2 Simulation Scripts

The bulk of the commands used to perform a simulation are contained within one file, the simulation script. The conducted simulations were controlled by a file called `wireless-simulation.tcl`. Simulation scripts generally follow the same pattern as the script `wireless-simulation.tcl`. The following sequence is used for most simulations:

- the options for the simulation are set;

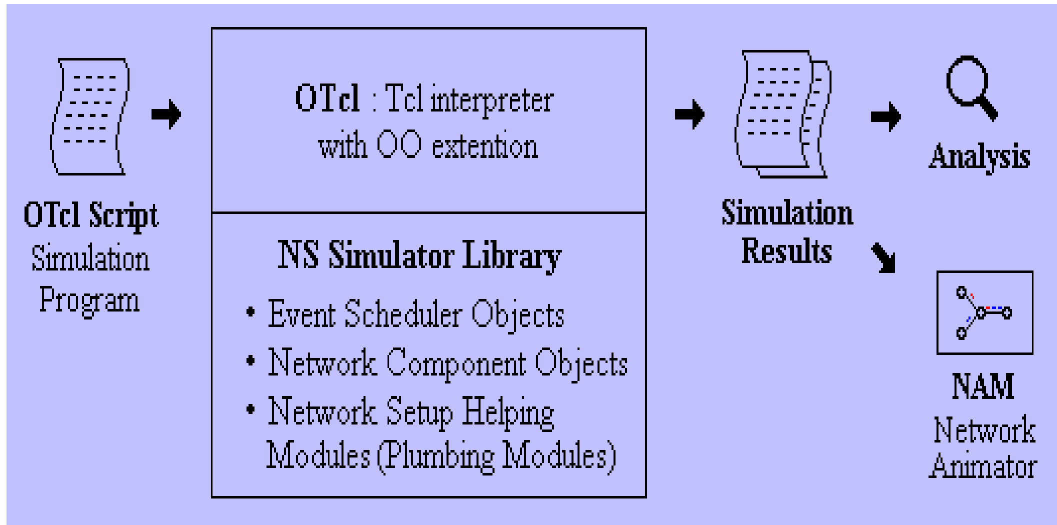

Figure 4.2: NS-2 Architecture

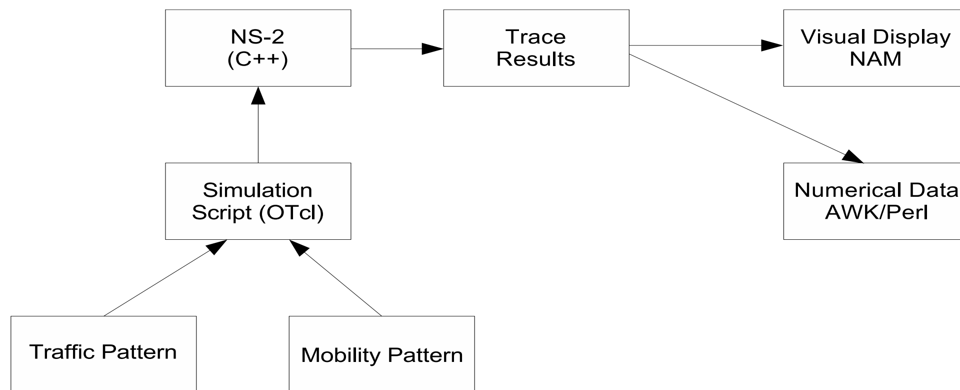

Figure 4.3: Steps Involved in an NS-2 Simulation

- the various network components are created;
- the mobile nodes are created;
- the mobile nodes generate network traffic;
- the post processing of the simulation occurs.

Figure 4.3 contains the general steps involved to run an ns-2 simulation [8].

Simulation Configuration

The beginning of a simulation script is primarily used to configure the network components of a simulation. For example, components such as the topography, queues, wireless channel, routing protocols, etc. are usually configured at the top a simulation script.

Typically, an array, `opt`, is used to store various elements that control a facet of the simulation. A block of code is used to initialize array elements of `opt`. For example, the command `set opt(x) 1000` assigns 1000 to the array element x. When a topography object is later created, the size of x component of the topography is assigned the value contained in `opt(x)`. By putting all of the options in a block of code, the user can easily go back and make any changes to the simulation. The following sample of code is used to set some of the element of `opt` that are later used when creating network objects:

```
# Set simulation options
set opt(chan)  Channel/WirelessChannel    ;# channel type
set opt(prop)  Propagation/TwoRayGround   ;# radio-propagation model
set opt(ant)   Antenna/OmniAntenna        ;# antenna type
set opt(x)     1000                       ;# x size for topography
set opt(y)     1000                       ;# y size for topography
....
```

Many of the network objects are configurable through the OTcl linkage, as discussed in Section 4.1.1. For example, the following code sample configures the 802.11 protocol's slot-time to 13 μs and SIFS to 32 μs. In addition, the wireless physical layer settings are also configured through the OTcl linkage.

```
# set the 802.11 parameters.
Mac/802_11 set SlotTime_      0.000013   ;# 13 us
Mac/802_11 set SIFS_          0.000032   ;# 32 us
....
Phy/WirelessPhy set freq_ 5.9e+9         ;# 5.9 GHz
Phy/WirelessPhy set bandwidth_ 6.0e6     ;# 6 Mbps
```

Numerous other aspects of the simulation can be specified at the start of the simulation script. To use the same physical layer setting as DSRC, the wireless channel's frequency is set to 5.9 GHz and the bandwidth is set 6 *mbps*. By setting the variables identical to those specified by DSRC standard, a simulation is able to realistically approximate a VANET.

The file `ns-2.29/tcl/lib/ns-default.tcl` sets various simulator components to their default values; this file can also be used to identify what simulation components a user is able to configure.

Creating Network Objects

After all of the configuration options have been set, the next step is to create the necessary network objects. To simulate a wireless network, a number of network components must be created. Some of the components needed to simulate a wireless network are `Simulator`, `Topography`, `god`, and `WirelessChannel`. The following code is used to create a new instance of the ns simulator and to create some of the standard objects that are needed for a wireless simulation:

```
# create a new simulator object
set ns_    [new Simulator]
# create the topography
```

```
set topo [new Topography]

$topo load_flatgrid $opt(x) $opt(y)

# create god, the god object must

# be created for a wireless network

set god_ [ create-god $opt(nn) ]

# create a channel object

set chan_ [new Channel/WirelessChannel]

....

# Continue setting up the simulation.
```

Most wireless simulation will follow the same format as the source code listed above. This code first creates an instance of the simulator. Next, the topography of the simulation is created. It is followed by the creation of a god object, and finally a channel object is instantiated.

After the basic network objects have been created, the next step is to configure the mobile nodes. The mobile nodes are configured by issuing a command similar to the one that follows:

```
# configure the nodes of the simulation

$ns_ node-config -adhocRouting $opt(rp) \

                 -llType $opt(ll) \

                 -macType $opt(mac) \

                 -ifqType $opt(ifq) \

                 -ifqLen $opt(ifqlen) \

                 -antType $opt(ant) \

                 -propInstance [new $opt(prop)] \

                 -phyType $opt(netif) \
```

```
                        -topoInstance $topo \
                        -channel $chan_ \
                        -agentTrace $opt(at) \
                        -routerTrace $opt(rt) \
                        -macTrace $opt(mact) \
                        -movementTrace $opt(movt)
```

As previously discussed, the elements of `opt` are set at the beginning of the simulation script. Most of the parameters used to configure a mobile node are from the `opt` array. The ns-2 documentation describes the configuration options for a mobile node.

After the nodes have been configured, the user is free to actually create mobile nodes. The following code creates an array of mobile nodes:

```
# Create the mobile nodes for the simulation.
for {set i 0} {$i < $opt(nn)} {incr i} {
    set node_($i) [$ns_ node]
}
```

After the mobile nodes are created, the network events are scheduled to complete the simulation. An example of an event that must be scheduled is the transmission of a packet. In addition, the user will typically add some functions such as one that performs clean up when the simulation is completed. The execution of clean-up function is scheduled as are all the other events that occur in a simulation. After all of the simulation is configured, the final step is to run the simulation. A simulation is executed with the following command:

```
....
# Run the simulation; this command is typically
```

```
# found at the very end of a simulation script.
$ns_ run
```

Generating Movement and Traffic

In the case of a VANET, a mobility model is needed for the movement of nodes. It is important that the simulation of a VANET is as realistic to an actual VANET as possible. Typically, a separate file contains all of the commands needed to move the nodes during the length of the simulation. To insert the code contained in another Tcl file into the main simulation script, the `source` command is used. The following commands are required to execute all of the commands contained in the file `mobility.tcl`:

```
set opt(sc)  "mobility.tcl"  ;# mobility scenario
....
# Source the mobility scenario file to execute
# all of the commands with the mobility file
source $opt(sc)
```

Section 4.2 contains a detailed description of the mobility model used for the simulations.

The user has a number of options available for creating network traffic (transmitting packets). To begin, all of the commands needed to send packets could be included in the main simulation script. The drawback to this approach is a large number of lines of code may need to be added to the simulation script, and this approach would reduce the readability and manageability of the simulation. Another alternative is to use a program such as `cbrgen.tcl` to genrate all of the traffic. In this case, the user supplies a few command line options to the program, and the

output of the `cbrgen` program is written to a file. Another choice is for the user to write his/her own program to generate the network traffic. This solution offers the user the greatest control over the generation of traffic.

If a separate file is used that contains all the commands needed to transmit packets, the file containing the network traffic must also be sourced, in the same manner as the mobility scenario. The following code is used to source a traffic file:

```
# Source the file that contains the broadcast traffic.
puts "sourcing traffic file"
source $opt(cp)
```

Section 4.3 contains a detailed description of the network traffic used in the simulations.

Running a Simulation

To run a simulation, the user types `ns wireless-simulation.tcl`, and the ns interpreter executes all of the commands contained in the `wireless-simulation.tcl` script. The time it takes to execute a simulation can range from a few minutes, to several hours, to days, depending on the complexity of the simulation.

A number of command-line options are available to the user when conducting a simulation with the `wireless-simulation.tcl` simulation script. Table 4.1 contains a list of the command-line options for `wireless-simulation.tcl`.

For many of the simulations that a user will run, a single parameter of the simulation is varied, and a number of simulations are run to determine the effect that changing the parameter has on the simulation results. In this scenario, it is beneficial to create another script with `bash` or `perl` to launch consecutive simulations and automate as much of the simulation process as possible.

Table 4.1: Command-Line Options for wireless-simulation.tcl

Option	***Description***
nn	The number of nodes contained in the simulation.
stop	The time to end the simulation.
seed	The seed for the random number generator.
sc	The name of the mobility scenario file.
cp	The name of the traffic connection pattern file.
tr	The name of the trace file to write the output to.
nam_tr	The name of the nam-trace file to write the output to.
x	The size of the x-axis of the topography.
y	The size of the y-axis of the topography.
pkt_size	The size of the periodic broadcast packets.
emer_pkt_size	The size of the emergency warning packets.
modified	Set to 1 use the modified algorithm and 0 for the 802.11 algorithm.
sliding	The threshold to use to adjust the CW.
help	Displays help to the user.

Table 4.2: Data Contained in an NS-2 Trace

Event	Time	From node
To node	Packet type	Packet size
Flags	Source address	Source port
Destination address	Desination port	Sequence number

As a simulation executes, the results of the simulation are written to a trace file. A trace contains detailed account of the events that occurred during a simulation. A trace is used to record the time at which specific events occurred, and each event is written on separate line of the trace to make parsing the file easy. Furthermore, the user can specify what events are written to the trace. Some of the events that can be written to a trace are routing events, MAC layer events, and mobility events (e.g., the position and movement of nodes). In addition, the simulation time can be improved if tracing is turned off for events of no interest.

The typical data contained on a line of a trace is given in Table 4.2. The text-based data of a trace is usually parsed with a script, which is typically written with Perl or AWK, to evaluate a simulation's performance. Section 4.4 contains a explanation of the method used for extracting data from trace and the metrics used to evaluate a simulation.

4.1.3 NAM: Network Animator

The NAM: Network Animator is a visualization tool that can be used to further examine a simulation. The NAM program is written with Tcl/TK, and NAM is used for visually displaying a simulation. To use NAM, the user must enable the creation of a nam-trace file during the simulation. As a simulation executes, the results are written to a nam-trace. The format of a nam-trace is similar to a regular trace generated by the ns-2 simulator. When a simulation is complete, the results of the simulation will be written to nam-trace; it is then used as input to the NAM program.

The following commands are issued to enable the creation of a nam-trace:

```
set opt(x)       1000            ;# x size for topology
set opt(y)       1000            ;# y size for topology
set opt(nam_tr)  "nam-trace.tr"  ;# name of the nam-trace file
....
# Set the file to write nam trace to and enable
# writing to the nam trace.
set namtracefd [open $opt(nam_tr) w]
$ns_ namtrace-all-wireless  $namtracefd $opt(x) $opt(y)
```

To execute NAM, at the command line, the user types `nam nam-trace.tr`, where `nam` executes the NAM program and `nam-trace.tr` is the name of nam-trace file. Once NAM has started, the interface to the NAM is similar to a video player. For example, the user can both fast-forward or rewind a simulation. Also, the user can control time resolution, to either speed up or slow down a simulation. Figure 4.4 [12] contains a sample of the user controls available with NAM for viewing a simulation.

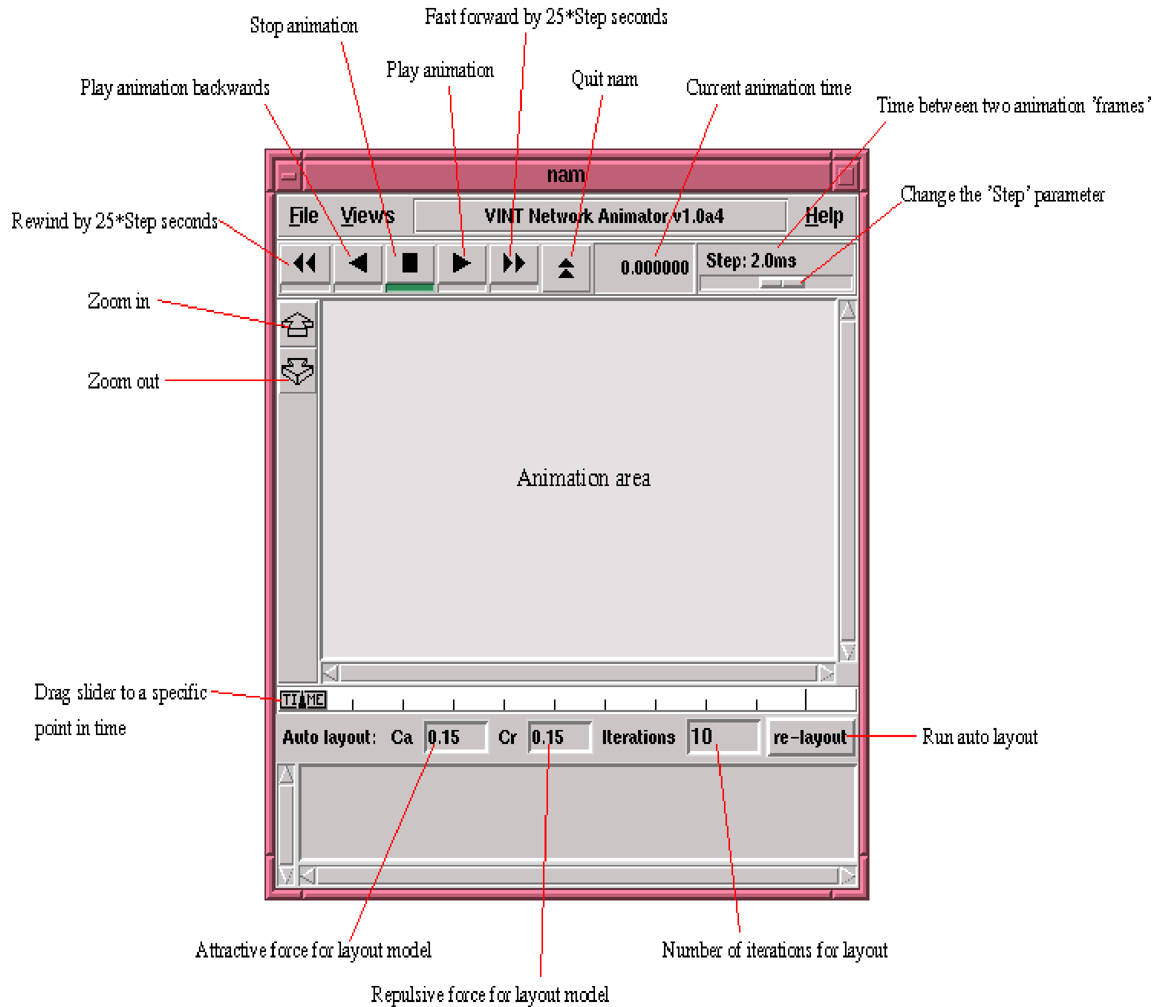

Figure 4.4: NAM Controls

The benefit of NAM is it give the user visual feedback for a simulation. The drawback of NAM is it does not present the user with quantitative results.

4.2 Mobility Model

The majority of mobile ad hoc simulations use the random waypoint mobility model; with this model, each node in the network randomly chooses a location to travel towards, and the node randomly selects a velocity with which to travel towards that destination. The velocity is randomly chosen from $[0, velocity_{max}]$, where

$velocity_{max}$ is the maximum allowable velocity for a node. Subsequently, the node continues moving toward its destination until it reaches the destination. When the node arrives at its destination, it will pause for the amount of time specified by its "pause timer". Then, the node randomly selects another destination to travel towards. The random waypoint model is provided by the `setdest` tool provided by ns-2.

While the random waypoint model is frequently used to model ad hoc networks, this model is inappropriate for modeling a VANET because of the unique characteristics of a vehicle's movement. Vehicles, in a VANET, cannot randomly travel anywhere within the topography. Vehicles are only able to travel where there is an adequate road. Since a vehicle's movement is constrained by the road, the random waypoint model is an unrealistic model for a vehicular network. For this reason, an alternative mobility model must be used for a VANET.

4.2.1 Freeway Mobility Model

The freeway mobility model is one possible model that can be used as a foundation for the movement of nodes in a VANET. The freeway model emulates the movement of vehicles on a freeway. The freeway model uses maps to create the mobility of the nodes. Nodes are only able to travel where a road is defined by a map. With this model, a map can contain several freeways, and each freeway may have multiple lanes. In addition, the lanes within a freeway can travel in either one or two directions.

Freeway Model Characteristics

The freeway model is characterized as follows:

- Each node is restricted to travel only within its own lane of the freeway. To simplify the complexity of this model, the model sacrifices some realism. As a

result, vehicles do not possess the ability to change lanes as they would on real freeway.

- The velocity of each node is temporally restricted based on the node's previous velocity. Equation 4.1 is used to determine a nodes velocity.

$$velocity_i(t+1) = velocity_i(t) + random() * acceleration_i(t) \tag{4.1}$$

- A safety distance is maintained so that a node cannot exceed the velocity of the node in front of it, if they are within the safety distance. Formally, the safety distance is defined as follows: if $distance_{i,j}(t) < safety_distance$, then $velocity_i(t) < velocity_j(t)$, if j is ahead of i in its lane.

USC Mobility Generator Software

The USC Mobility Generator [6] software implements the freeway model, and the software creates ns-2 mobility traces for a number of different mobility models, including the freeway mobility model. The source code for the USC Mobility Generator is freely available from the project's website. After the `freeway.cpp` source code is obtained, the source code must be compiled. When executing the `freeway` program, the user is prompted for the following information:

1. The number of nodes contained in the simulation.
2. The acceleration speed of the nodes. (The software authors recommend setting the acceleration of nodes to 10% of the maximum velocity.)
3. The name of the map file that contains a description of the freeways.
4. The name of the file to create. (Where the the ns-2 mobility trace will be written.)

After these four items are specified, the freeway program executes and writes an ns-2 mobility trace to a file. The mobility trace can later be sourced within an ns-2 simulation script.

4.2.2 Map File

A topography file is a critical component that determining the topology of the network. Hence, the map file defines all of the freeways that exist in a simulation. As a result, it is possible to model any freeway by suppling the correct data (in the form of a map file) to the freeway program. For example, the major freeways of the Metro Detroit area can be modeled by including the data points for these freeways in a map file. The following is the specification of the map file format. This specification was taken from the manual.txt file of the USC Mobility Generator program:

```
FREEWAY
FREEWAY_NUM <total_number_of_freeways>
LANE_NUM <total_number_of_lanes>
LANE_BEGIN <freeway_id> <lane_id_in_this_freeway>
           <lane_id_in_all_freeway> <direction>
           <number_of_phases_of_this_lane>
PHASE <phase_id> (<phase_start_x0,phase_start_y0>)
                 (<phase_end_x1,phase_end_y1>) <v_min> <v_max>
PHASE <phase_id> (<phase_start_x1,phase_start_y1>)
                 (<phase_end_x2,phase_end_y2>) <v_min> <v_max>
....
```

The documentation, that comes with the USC Mobility Generator, thoroughly describes all of the above fields.

To create a map file for the simulations, the program `create-map.pl` was written with the Perl language. The `create-map.pl` program, available in Appendix B.1, generates an approximation of a circular map. The length of the inner-radius is specified for the map, and the inner-radius is the distance, in meters, from the center of the map to the inner-lane of the freeway. The default size of the inner-radius is 300 m. Freeway lanes are then placed 5 m a part. For example, if the inner-radius of the freeway is 300 m, the next lane of the freeway is 305 m followed by another lane placed at 310 m. Lanes are added to the freeway until the number of lanes specified for the freeway is reached. By default, the program creates an eight-lane freeway, with four lanes traveling in each direction. In addition, the user can change the inner-radius of the circle to increase or decrease the size of the freeway. Also, the user can specify the number of points used to construct the freeway. A map becomes more circular as more points are used to construct the map file.

The following is a portion of a map file generated with the `create-map.pl` program:

```
FREEWAY
FREEWAY_NUM 1
LANE_NUM 8
LANE_BEGIN 0 0 0 1 24
PHASE 0 (800.00000, 500.00000) (789.77775, 577.64571) 17.000 25.000
PHASE 1 (789.77775, 577.64571) (759.80762, 650.00000) 17.000 25.000
PHASE 2 (759.80762, 650.00000) (712.13203, 712.13203) 17.000 25.000
....
PHASE 21 (712.13203, 287.86797) (759.80762, 350.00000) 17.000 25.000
PHASE 22 (759.80762, 350.00000) (789.77775, 422.35429) 17.000 25.000
```

```
PHASE 23 (789.77775, 422.35429) (800.00000, 500.00000) 17.000 25.000

LANE_BEGIN 0 1 1 1 24

PHASE 0 (805.00000, 500.00000) (794.60738, 578.93981) 17.000 25.000

...

PHASE 23 (794.60738, 421.06019) (805.00000, 500.00000) 17.000 25.000

LANE_BEGIN 0 2 2 1 24

PHASE 0 (810.00000, 500.00000) (799.43701, 580.23390) 17.000 25.000

...

PHASE 23 (799.43701, 419.76610) (810.00000, 500.00000) 17.000 25.000

LANE_BEGIN 0 3 3 1 24

PHASE 0 (815.00000, 500.00000) (804.26664, 581.52800) 17.000 25.000

...

PHASE 23 (804.26664, 418.47200) (815.00000, 500.00000) 17.000 25.000

LANE_BEGIN 0 4 4 -1 24

PHASE 0 (820.00000, 500.00000) (809.09626, 417.17791) 17.000 25.000

...

PHASE 23 (809.09626, 582.82209) (820.00000, 500.00000) 17.000 25.000

LANE_BEGIN 0 5 5 -1 24

PHASE 0 (825.00000, 500.00000) (813.92589, 415.88381) 17.000 25.000

...

PHASE 23 (813.92589, 584.11619) (825.00000, 500.00000) 17.000 25.000

LANE_BEGIN 0 6 6 -1 24

PHASE 0 (830.00000, 500.00000) (818.75552, 414.58972) 17.000 25.000

...

PHASE 23 (818.75552, 585.41028) (830.00000, 500.00000) 17.000 25.000
```

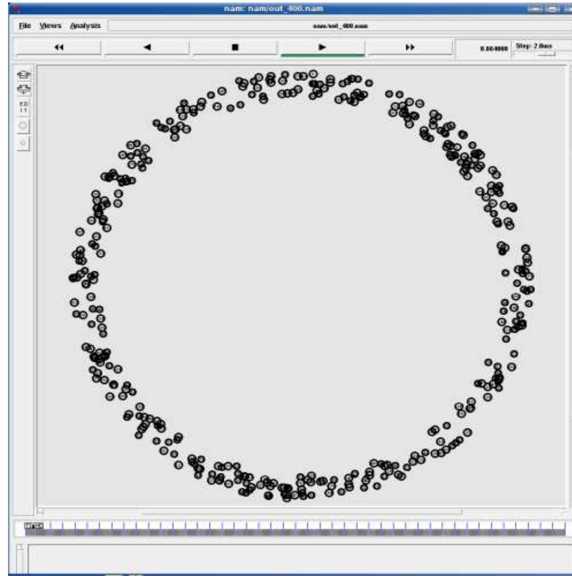

Figure 4.5: Simulation Topography

```
LANE_BEGIN 0 7 7 -1 24
PHASE 0 (835.00000, 500.00000) (823.58515, 413.29562) 17.000 25.000
...
PHASE 23 (823.58515, 586.70438) (835.00000, 500.00000) 17.000 25.000
```

The sample map file displayed above contains one freeway that is made up of eight lanes. In addition, each freeway lane consists of 24 points that are used to construct this portion of the freeway. `PHASE 0` through `PHASE 23` represent these points in the map. The velocity of the vehicles on a portion of the freeway is defined by the final two fields in each row that starts with the `PHASE` identifier. For each lane and phase of freeway, it is possible to assign different values to $velocity_{min}$ and $velocity_{max}$. In this case, minimum and maximum velocity are set the same throughout the freeway, and $velocity_{min}$ is 17.0 m/s and $velocity_{max}$ is 25.0 m/s for all of the lanes. Figure 4.5 contains a screen capture from the NAM program when this map is used for a simulation.

4.2.3 Mobility Trace

After a map is generated by `create-map.pl`, the map file is used as input to the `freeway` program. The output of the `freeway` program is a mobility trace. The following mobility trace, `mobility.tcl`, was generated by the `freeway` program:

```
# set the initial postion
$node_(0) set X_ 491.621429
$node_(0) set Y_ 833.896912
$node_(0) set Z_ 0.000000
$node_(1) set X_ 200.855087
$node_(1) set Y_ 406.107025
$node_(1) set Z_ 0.000000
...
$ns_ at 1.000000 "$node_(0) setdest 541.063599 829.593872 25.000000"
$ns_ at 1.000000 "$node_(1) setdest 215.333633 371.152771 18.642939"
....
```

The `mobility.tcl` trace starts off by assigning each node an initial x, y, or z coordinate. As of the current implementation of ns-2 version 2.29, the z coordinate has no affect on the simulation, but the z coordinate must be specified. After the nodes' initial positions are set, the nodes are put in motion. The command, "`$ns_ at 1.000000`" schedules an event to occur at 1.0 s. The rest of the previous command is "`$node_(0) setdest 541.063599 829.593872 25.000000`". In this case, $node(0)$ is set to travel towards the x coordinate 541.063599 and the y coordinate 829.593872 at a velocity of 25.0 m/s. $Node(0)$ will continue to travel to this destination until periodically $node(0)$'s destination and velocity will be altered in the mobility trace

to send it to a new destination.

The `mobility.tcl` file provides the movement of the mobile nodes throughout a simulation. The radius of the freeway was made large enough so that a transmission on one-side of the road would not affect a concurrent transmission on the other-side of the road. Also, the diameter of the inner-lane of the freeway is 600 m, and the circumference of the inner-lane is approximately 1884 m. One trade-off that was considered was the portion of road to model for the simulation. If the diameter of the road was increased, more nodes would have to be added to the network to have the same amount of network traffic that would result in the length of the simulation increasing.

4.3 Network Traffic

Various types of networks exist, and the traffic pattern of each network differs from that of the other networks. Furthermore, the traffic pattern of a simulation should resemble an actual VANET's traffic.

Most networks favor certain applications, and most network applications favor the use of a specific transport protocol. For example, the majority of traffic on one network may use TCP while another network will primarily use UDP. The transport protocols used will affect the traffic profile of a network. The transport protocol used to deliver broadcast messages in a VANET is UDP. Because of the nature of broadcast traffic, no transport-layer connection is established before transmitting.

The network applications used also affects the traffic profile. For instance, some applications have bursty traffic, while other applications provide a constant stream of packets to the network. Typically, applications that use the HTTP protocol have bursty traffic, and VoIP applications transmit a steady stream of bytes. In a VANET,

nodes will broadcast their status every 100 ms. In turn, this application provides the network with a constant stream of traffic. On the other hand, when an accident occurs, a burst of warning messages will follow the event.

The ns-2 simulator comes with a number of stand-alone programs which generate simulation traffic. The problem with using the applications provided by ns-2 for creating network traffic is they only create unicast traffic. The program `cbrgen.tcl` is one of the more popular programs for generating network traffic. For instance, the `cbrgen.tcl` program creates random TCP connections between nodes and uses the FTP application protocol for generating traffic at a constant bit rate. In the case of a VANET, TCP connections are usually not established before application data is transmitted. For this reason, an additional application was created (`traffic-w-jitter.pl`) to generate the network traffic for the simulations.

4.3.1 QoS

The EDCA protocol, of 802.11e, is currently not implemented for the default installation of ns-2. As a result, a number of people have created third-party extensions for ns-2 that implement the EDCA protocol. Sven Wiethölter and Christian Hoene created an EDCA extension that is used for the simulations [24].

The simulation will use two classes of traffic. The first class $AC0$ represents high priority traffic, and this class of traffic is used for transmitting emergency warnings. The second class $AC1$ represents low priority traffic and is used to periodically broadcast a vehicle's state.

4.3.2 Creating Network Traffic

After a node is created in a simulation script, agents are attached to a node. Agents represent endpoints where network-layer packets are constructed or con-

sumed, and they are used in the implementation of protocols at various layers [4]. A number of different agents exist for ns-2 including UDP Agent, TCP Agent, Null Agent, MessagePassing Agent, etc. To create a new TCP agent, the following code is used:

```
# Add Transport agents
set tcp [new Agent/TCP]
$ns_ attach-agent $n(0) $tcp
set tcpsink [new Agent/TCPSink]
$ns_ attach-agent $n(1) $tcpsink
$tcp connect $tcpsink
```

The code listed above creates a new TCP agent object and establishes a connection between nodes $n(0)$ and $n(1)$.

After a transport agent is attached to a node, an application agent is then typically attached to a transport agent. Some of the application agents that are available for ns-2 are FTP, CBR, Worm, etc. The code below is used to attach an FTP application to a TCP agent:

```
# Add application
set ftp [new Application/FTP]
$ftp attach-agent $tcp
```

After the FTP application is attached to a transport agent, a node can then schedule the application.

In the terms of the VANET simulation, all of the immediate neighbors of a node should receive a broadcast transmission. The MessagePassing agent is one of the agents that can be used for transmitting packets. The benefit of using this agent

is the MessagePassing agent allows the user to specify both an address and a port for a transmission. For the simulations, the destination address should be -1, the broadcast address. Also, the port number to which a packet is sent provides a way to determine what application sent which packets.

The MessagePassing agent is the base agent class used in the simulations. The MessagePassing agent class is extended, and a new class is created for each access category used in the simulation. For instance, the class `EmerBroadcast` extends the `MessagePassing` class. `EmerBroadcast` is used for access category AC-0, and is used to transmit EWMs. The `EmerBroadcast` class defines two methods, `recv` and `send_message`, that are used to receive and transmit messages.

The following code defines a new class, `EmerBroadcast`, that extends the base class `Agent/MessagePassing`:

```
# EmerBroadcast class used to send emergency
# messages to a nodes one-hop neighbors.
Class Agent/MessagePassing/EmerBroadcast
                             -superclass Agent/MessagePassing
Agent/MessagePassing/EmerBroadcast instproc recv
                                   {source sport size data} {
    # This empty function is needed so receive works.
}

Agent/MessagePassing/EmerBroadcast instproc send_message {} {
    $self instvar node_
    global ns_ EMER_PORT BROADCAST_ADDR opt
    # send the broadcast message
```

```
    $self sendto $opt(pkt_size_emer) 0 $BROADCAST_ADDR $EMER_PORT
}
```

The following code creates new `EmerBroadcast` agents and attaches an agent to each of the nodes in the simulation. Each node attaches the agent to a specific port, in this case `EMER_PORT` is port 21. Each broadcast agent is attached to a different port so that it is easy to differentiate the access categories when processing a simulation trace.

```
# Attach a new Agent/MessagePassing/EmerBroadcast
# to each node on port $EMER_PORT, 21
for {set i 0} {$i < $opt(nn)} {incr i} {
    set em_agent($i) [new Agent/MessagePassing/EmerBroadcast]
    $em_agent($i) set class_ 0
    $em_agent($i) set prio_ 0
    $node_($i) attach  $em_agent($i) $EMER_PORT
}
```

4.3.3 The Network Traffic Program: traffic-w-jitter.pl

Because there is no standard ns-2 program that creates network traffic identical to a VANET's traffic profile, the program `traffic-w-jitter.pl`, listed in Appendix C.1, was created to generate the network traffic. As previously discussed, each node broadcasts its status to its neighbors every 100 ms. Consequently, a node must schedule a broadcast approximately every 100 ms.

The program `traffic-w-jitter.pl` uses the variables `start` and `end` time, and these variables determine the duration of transmissions for a simulation. If the user fails to supply the `start` and `end` time from the command-line, the default values are

Table 4.3: Command-Line Options for traffic-w-jitter.pl

Option	***Description***
start	The time to start sending packets.
end	The time to stop sending packets.
nodes	The number of nodes contained in the simulation.
jitter	Maximum variation between consecutive broadcasts by a node.
seed	The seed for the random number generator.
file-name	The name of the file to write the output to.
traffic-rate	The number of periodic broadcasts a node transmits per second.
emer-duration	The number of emergency warnings to transmit for a node.
help	Displays help to the user.

used. Consequently, the `start` variable is assigned 1 and `end` is assigned 61 resulting in 60 s of network traffic. Also, the user can specify the transmission rate (e.g., how many transmissions per second) for periodic broadcasts; the default value is 10 times per second. In addition, the amount of jitter is able to be specified. The jitter adds a small variation in the time period between consecutive broadcasts by a node. Table 4.3 contains a description of all of the command-line options available to the user of the program.

When the `traffic-w-jitter.pl` program is executed, the start time is used to select the time of the simulation to start transmitting packets. For example if the `start` time is 1 s and the `traffic-rate` is 10 (i.e., a node broadcast every 100 ms) each node is randomly assigned a time to start transmitting packets in the range of [1.000 s, 1.100 s). As a result, the transmission time of the next broadcast will be in the range of [1.100 s, 1.200 s). Based on the random start time selected for a node, in the case of a transmission rate of 10 times-per-second, the next transmission is scheduled every 100 ms from the start time. To increase the realism of the simulations, jitter is introduced into the time that packets are scheduled for transmission. The default value for the `jitter` variable is 0.10 (i.e., 10% jitter), which results in the transmission time being randomly varied, in the previous example, by $\pm$ 10 ms. In other words, if the start time selected for a

node is 1.030 s, and the next transmission time is 1.130 s, the jitter would cause the actual transmission time to be in the range [1.120 s, 1.140 s]. The transmission times continue to be selected in the same manner until the `end` time of the simulation is reached.

Emergency warning messages are also scheduled by the `traffic-w-jitter.pl` program. A variable in the program, `perc_emer`, is used to randomly select nodes to transmit emergency warnings. The default value of the variable `perc_emer` is set to 0.015 (e.g., nodes have a 1.5% chance of being selected to transmit a series of emergency warnings). Before scheduling a periodic broadcast, the following `rand(1) <= $perc_emer` comparison is made. If the comparison returns *true*, the node is selected to transmit a series of emergency warnings. The variable `num_emer_messages` determines the number of emergency warnings to transmit.

The result of the execution of the `traffic-w-jitter.pl` program is a file containing all of the commands to generate network traffic for the length of a simulation. The following output was generated by the `traffic-w-jitter.pl` program:

```
$ns at 1.02871501295167 "$bc_agent(0) send_message"
$ns at 1.05178765152554 "$bc_agent(1) send_message"
$ns at 1.0036645125636 "$bc_agent(2) send_message"
....
$ns at 1.12164568447645 "$em_agent(27) send_message"
....
```

The code listed above schedules numerous similar events. Hence, the `send_message` function of the `EmerBroadcast` and `PeriodicBroadcast` classes are called resulting in the transmission of broadcast messages.

Table 4.4: Performance Metrics

Metric	***Description***
Reception rate	The percentage of packets successfully received at a specific distance.
Access delay	The amount of time it takes from when a packets is passed down to the MAC layer until it is placed on the wireless channel.

4.4 Performance Metrics

A metric provides a standard measure for accessing the performance of a specific subject. To evaluate the simulations, metrics are needed to determine the effectiveness of the broadcast protocols. The two quantitative metrics contained in Table 4.4, reception rate and access delay, are used to evaluate the proposed broadcast algorithm.

Reception rate measures the percentage of packets successfully received at a specific distance. Because of the unreliable nature of broadcast transmissions, a percentage of the frames transmitted will fail to be delivered. As discussed in Section 1.2, the probability that a frame will be received is affected by the distance between the sender and receiver. Therefore, the distance between the sender and receiver is considered when the reception rate is calculated. As the distance between nodes increases the probability of reception decreases. The ideal delivery rate of any broadcast protocol is 100% delivery. The metric reception rate is used to determine how far a protocol's performance is from the ideal case.

Access delay measures the amount of time it takes from when a packet is passed down to the MAC layer until it is placed on the wireless channel. If a packet is held by the MAC layer for too long, the data contained in the packet may no longer be relevant when it is finally received. For example, it is unacceptable if it takes a second or longer to receive a location update. Within a second, a vehicle's location is able to drastically change. One requirement of DSRC is that the delay should be

kept less than 100 ms, and the proposed protocol should also attempt to meet this requirement.

Two Perl scripts were created, `reception.pl` and `access-time.pl`, for the purpose of evaluating the simulations. The program `reception.pl` is used to determine the overall reception rates at specific distances. In addition, the program `access-time` measures the access delay for a simulation. At the completion of a simulation, the programs `reception.pl` and `access-time.pl` parse the simulation trace and calculate the performance metrics.

4.4.1 NS-2 Simulation Trace Format

As discussed in Section 4.1.2, the simulation results are written to a trace file. To measure quantitatively the performance of a network protocol, the relevant data must be extracted from a trace. Consequently, the trace format follows a specific format with fields being placed in specific locations and the "ns-2 Manual" describes in detail the format of a trace[4].

An ns-trace is comprised of fields, and each trace field is delimited by white space. As result of the trace fields being separated by white space, a trace is easily parsed with a language such as Perl or AWK. Also, to simplify parsing a trace, each trace field is in a specific location. For example, the first trace field records the type of event that occurred. Some of the valid simulation events are: `s` for send, `r` for receive, `d` for drop, and `M` for mobility. In addition, the second field records the time that the event occurred. Furthermore, the third field is the node ID that is used to uniquely identify each node. The format of data contained in a trace is described in the ns-2 documentation. Numerous other fields, that can be extracted, also contain information about a simulation. While some of the fields' content is always the same (e.g., the second field always contains the time of an event), the content of many of

the fields varies based on the type of event. For instance, the fields after the third column have different meanings for mobility events as compared to send events.

The following is sample output from a trace:

```
...
M 1.00000 48 (1178.05, 245.00, 0.00), (1200.69, 245.00), 22.64
M 1.00000 49 (2681.49, 255.00, 0.00), (2660.85, 255.00), 20.64
s 1.001393522 _34_ AGT  --- 0 message 200 [0 0 0 0] ------- [34:42 -1:42 32 0]
s 1.001488522 _34_ MAC  --- 0 message 260 [0 fffffff 22 0] ------- [34:42 -1:42 32 0]
s 1.003523553 _37_ AGT  --- 1 message 200 [0 0 0 0] ------- [37:42 -1:42 32 0]
r 1.003568779 _9_ MAC  --- 0 message 200 [0 fffffff 22 0] ------- [34:42 -1:42 32 0]
r 1.003568890 _1_ MAC  --- 0 message 200 [0 fffffff 22 0] ------- [34:42 -1:42 32 0]
...
```

4.4.2 Reception Rate

The program **reception.pl**, listed in Appendix D.1, calculates the reception rate metric for each access category involved in an ns-2 simulation. In the terms of calculating the reception rate, the distance between the nodes in the simulation must be maintained. The reception rate of an access category is the percentage of packets successfully received at a specific distance from the sender for the length of the simulation. In addition, reception rate is evaluated on the percentage of packets received at certain intervals. The reception.pl script uses 10 m intervals for calculating the reception rate. For instance, the reception rate is calculated for the ranges [0 m, 9 m], [10 m, 19 m], [20 m, 29 m], ...

The following is a sample of the output generated from the execution of the `reception.pl` program (in addition, the `--output` switch is used to specify the name of the file to save results in a format to easily generate a graph):

```
./reception.pl --trace trace/trace_400.tr --output data/reception-400.txt

Number of Packets Received and Dropped for 400 Nodes
-------------------------------------------------------------------------
port 21
--------------------------------------------------------------------------------------
dist:   0 -   9 m, recv:  49430, dropped:    1028, not sensed:   46, percent recv: 97.873
dist:  10 -  19 m, recv: 151195, dropped:    5967, not sensed:  167, percent recv: 96.101
dist:  20 -  29 m, recv: 213404, dropped:   12587, not sensed:  306, percent recv: 94.303
dist:  30 -  39 m, recv: 227151, dropped:   18039, not sensed:  479, percent recv: 92.462
```

```
dist:  40 -  49 m, recv: 189342, dropped:   19430, not sensed:  506, percent recv: 90.474
dist:  50 -  59 m, recv: 178739, dropped:   22274, not sensed:  630, percent recv: 88.641
dist:  60 -  69 m, recv: 173036, dropped:   26114, not sensed:  759, percent recv: 86.557
dist:  70 -  79 m, recv: 170549, dropped:   29687, not sensed:  781, percent recv: 84.843
dist:  80 -  89 m, recv: 163553, dropped:   32384, not sensed:  942, percent recv: 83.073
dist:  90 -  99 m, recv: 159166, dropped:   40277, not sensed: 1042, percent recv: 79.390
....
port 42
---------------------------------------------------------------------------------------
dist:   0 -   9 m, recv: 196724, dropped:   5319, not sensed:   277, percent recv: 97.234
dist:  10 -  19 m, recv: 608390, dropped:  31860, not sensed:  1111, percent recv: 94.859
dist:  20 -  29 m, recv: 843626, dropped:  66362, not sensed:  1913, percent recv: 92.513
dist:  30 -  39 m, recv: 888188, dropped:  92960, not sensed:  2535, percent recv: 90.292
dist:  40 -  49 m, recv: 748554, dropped:  99098, not sensed:  2870, percent recv: 88.011
dist:  50 -  59 m, recv: 695853, dropped: 111780, not sensed:  3365, percent recv: 85.802
dist:  60 -  69 m, recv: 677853, dropped: 127794, not sensed:  3903, percent recv: 83.732
dist:  70 -  79 m, recv: 664750, dropped: 145231, not sensed:  4635, percent recv: 81.603
dist:  80 -  89 m, recv: 638136, dropped: 158730, not sensed:  5089, percent recv: 79.573
dist:  90 -  99 m, recv: 621535, dropped: 185702, not sensed:  5601, percent recv: 76.465
....
```

Within an ns-2 simulation script, a user can turn on the tracing mobility events and specify how often the location of a node is written to a trace. The more often a node's position is written to a trace, the more accurate the position is when determining a node's location. The drawback of frequently updating a node's location is it increases the run-time of a simulation. For the simulations, the nodes' locations are written to the trace every 100 ms. As a result, the nodes' locations within the topography must be maintained when calculating the reception rate. Each line that starts with an `M` signifies a mobility event. When a trace line starting with an `M` is read, the node location is updated within the D.1 program.

When the D.1 program begins execution, the trace file is first opened. The trace to open is specified by the command-line option `--trace`. After the trace is opened, the program starts to read and process each line of the file. The only events of interest are those that occur at the MAC layer. Specifically, the events `s` for send, `r` for receive and `d` for drop are of interest.

A line starting with a `s` indicates that a packet is sent. For each packet sent from the MAC layer, the neighboring nodes within the transmission range of the sender

are recorded. When a packet is transmitted, the packet ID and the neighbors that should receive the packet are recorded. In addition, a line that begins with a `r` signals that a packet was received. When a packet is received for a packet ID, the receive count at the distance between the nodes is incremented by one. Also, a `d` indicates that a packet was received in error, so the count of dropped packets is incremented.

After the entire trace is read, the percentage of packets successfully received at each distance is calculated, and the results are displayed to the user.

4.4.3 Access Delay

The program **access-time.pl**, listed in Appendix D.2, calculates the access-delay metric for a simulation. A number of different access categories may be used in a simulation, so the access delay must determined for each access category. The program, `access-time.pl`, calculates the average access delay and the number of frames dropped for each access category.

The following is a sample of the output generated from the execution of the `access-time.pl` program:

```
./access-time.pl --trace trace/trace_800.tr --output delay.txt
packet type = MESSAGE
number nodes    = 800
avg access time = 0.00918130335886552
dropped at MAC  = 0
sent packets    = 383943

packet type = EMERGENCY
number nodes    = 800
```

```
avg access time = 0.00388675118733659

dropped at MAC  = 0

sent packets    = 96057
```

Upon execution of the `access-time.pl` program, the program begins by opening a trace file, specified by the command-line option `--trace`, and starts reading each line of the trace. The only events of interest in determining the access delay are send events (e.g., trace lines that start with `s`). Table 4.5 contains a description of the events of interest.

Each line contained in a trace is read by the `access-time.pl` program. If a line begins with an `s`, the event for the line starting with an `s` is processed by the `access-time.pl` program.

When a send event, `s`, is processed, it must be determined if the event occurred at the `AGT` layer or the `MAC` layer. If the event is an `AGT` event, it signals that a packet was passed down from the application agent. In other words, the packet was passed down from the application layer to the MAC layer. In terms of this event, the time that a packet is passed down to the MAC is recorded. The other event of interest is a send event at the MAC layer, which means that the frame was placed on the physical channel. The differences in time between the send `AGT` time and send `MAC` time is the access delay for a packet. For each frame passed to the MAC, it must be determined what access category is used to transmit the frame. When the access delay is determined for a specific frame, it is added to a total.

After all of the trace is read, the average access delay for each access category is calculated. The program also determines how many frames were dropped at the MAC layer for each access category. The results of the simulation are displayed to the user and are also written to a file in a format that is easy to generate a graph

Table 4.5: Trace Events to Determine the Access Delay

Event	***Layer***	***Description***
`s`	`AGT`	A packet was passed down from the application layer to the MAC layer.
`s`	`MAC`	A packet was passed from the MAC layer to the physical layer to be placed on the wireless channel.

from using the `gnu-plot` program.

4.5 Adaptive Broadcast Implementation

To implement the adaptive 802.11 broadcast protocol, as suggested in Chapter III, modifications are needed to the existing 802.11 MAC-layer source code for the ns-2 simulator. The ns-2 code must be modified to provide the additional functionality for the proposed broadcast protocol that is not already provided by the existing implementation. In addition to modifying the existing source code, a few additional C++ classes must be created to simulate the proposed broadcast protocol.

The simulations use the 802.11e protocol for ns-2 (the specific 802.11e implementation is provided by Telecommunication Networks Group at Technische Universität Berlin) instead of the standard 802.11 MAC protocol. The reason for using 802.11e, as discussed earlier in Section 2.3, is that it supports the differentiation of traffic classes. The existing 802.11e code must also be modified to integrate the new C++ classes (e.g., the broadcast table for the CW algorithm) with the existing MAC protocol. Specifically, the code for the 802.11e protocol that implements the adjustment of contention window must be modified.

In addition, two new C++ classes were developed for the implementation of the broadcast protocol. These classes were created to help predict the state of the network. For instance, each node maintains a table that stores data about the other nodes in the network, and the node uses the information contained in the table to

predict the state of the network. An efficient data structure is needed to store the data about the other nodes. The data from the table is later retrieved from the data structure and used to calculate the state of the network. The class `broadcast_table` stores data about the nodes that have recently been within the communication range. The data structure used to implement the `broadcast_table` is a hash table, so that the information can be quickly retrieved. The class `broadcast_entry` is used to store individual entries of the `broadcast_table`.

4.5.1 Broadcast Table Implementation

The files `broadcast_table.h` and `broadcast_table.cc` contain two new C++ classes for the ns-2 simulator. Consequently, these files contain the implementation for the `broadcast_entry` class and the `broadcast_table` class. As a result, these classes are used to provide the functionality of a hash table, and each mobile node maintains this hash table.

broadcast_entry

The `broadcast_entry` class contains the data members needed to determine the percentage of packets successfully received from a specific node in the network. The `broadcast_entry` objects are the data structure used to store an element of the broadcast table (hash table), as discussed in Section 3.2. Furthermore, Figure 4.6 is the Unified Modeling Language (UML) class diagram for the `broadcast_entry` class. In addition, the source code for the `broadcast_entry` class is listed in Appendix E.1.

The class `broadcast_entry` contains the following data members:

- `mac_address_` is the MAC address of a node.
- `sequence_number_` is the sequence number of the last overheard packet from the node with the MAC address contained in the `mac_address_` data member.

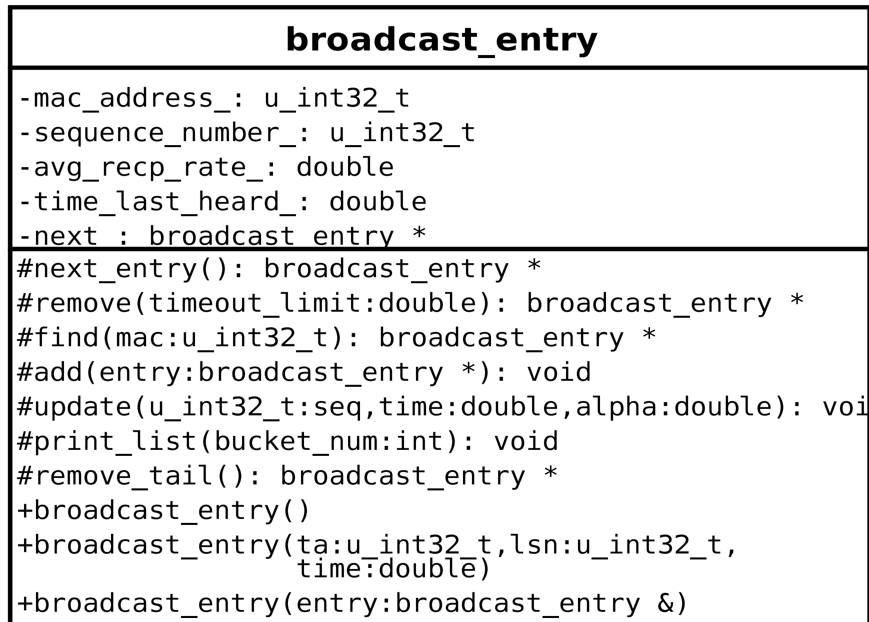

Figure 4.6: Broadcast Table Entry Class Diagram

- `avg_recp_rate_` is a moving average of the reception rate (the percentage of packets successfully received).
- `time_last_heard_` is the time that a packet was last overheard coming from the node with the MAC address contained in the `mac_address_` data member.
- `next_` is a pointer to the next node of a linked list of `broadcast_entry` objects.

The class `broadcast_entry` contains a number of member functions and each of these member functions are described below:

- `next_entry` returns the next `broadcast_entry` contained in the linked list.
- `remove` iterates through the linked list and removes all of the entries in the linked list that have reached the timeout limit. The `timeout_limit` is passed to the function. The member function returns the new head of the list and `NULL` if all elements are deleted.

- `find` searches the linked list for the entry with the MAC address passed to the method. If a broadcast_entry is found for the MAC address it is returned or else `NULL` is returned.
- `add` appends a new `broadcast_entry` to the linked list.
- `update`, updates the information stored for a node and calculates the new estimated reception rate.
- `print` displays the data maintained for a mobile node.
- `print_list` displays all nodes contained in the linked list. The function iteratively calls the print method of each `broadcast_entry` object.
- `remove_tail` removes the last entry in the linked list. The method returns the new tail item in the linked list or returns `NULL` to indicate no more items exist in the list.

broadcast_table

Figure 4.7 contains the UML class diagram for the `broadcast_table` class. In addition, the `broadcast_table` source code is listed in Appendix E.1.

The `broadcast_table` class contains the following data members:

- `table_` is the "broadcast table" used to store an array of `broadcast_entry` objects. In other words, `table_` is an array of linked lists of `broadcast_entry` objects.
- `table_size_` is the size of the broadcast table (i.e., the number of array elements allocated to `table_`).
- `timeout_` is the time used to invalidate old table entries. Entries are removed

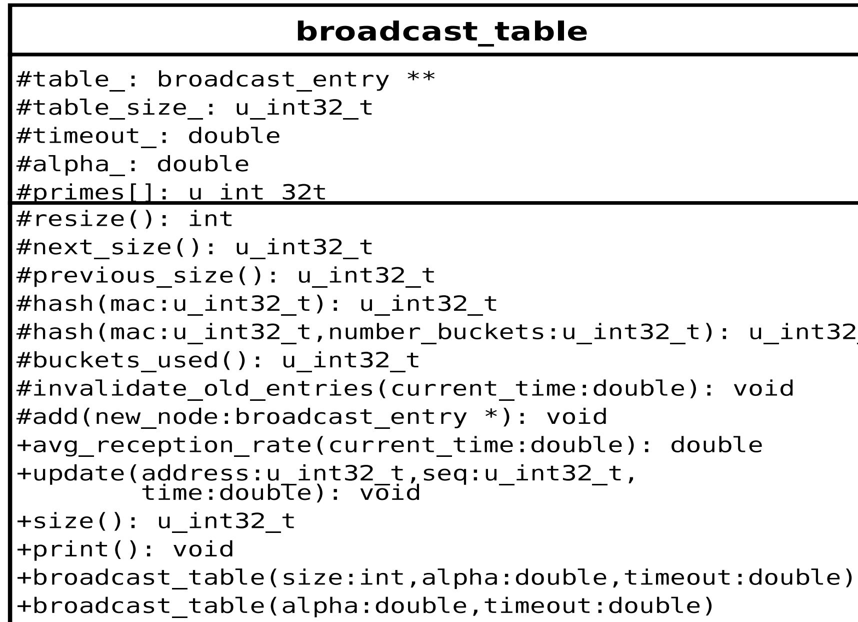

Figure 4.7: Broadcast Table Class Diagram

from the table if the `time_last_heard_` < `timeout_` for a `broadcast_entry` object.

- `alpha_` is used to estimate the weighted average for the reception rate. The value of alpha determines how fast the reception rate reacts to changing network conditions.

- `primes_` is an array of prime numbers that are used for the hash function and the table size.

The `broadcast_table` class contains the following member functions:

- `resize`, resizes the hash table. The function returns 1 if the size of the table is increased; returns -1 if the size of the table is decreased; and returns 0 if the size of the table remains the same.

- `next_size` returns the next value of increasing size to be used for resized the hash table and the value returned from this function is used to increase the

size of the hash table. The new table size is selected from the array of prime numbers.

- previous_size is similar to next_size, the method returns the next smallest table size used to resize the hash table. The table size is selected from the array of prime numbers.

- hash returns the hash value for a MAC address. Consequently, the value returned from the hash function is used to select the element (bucket) of the table_ array.

- buckets_used calculates the number of buckets (array elements) that are used in the hash table. The function is used to determine how full the hash table is.

- invalidate_old_entries removes any entries from the hash table that have not been updated within the timeout_ period. For example, if the value of timeout_ is 1 s than entries are removed from the hash table if they were last updated more than a second ago.

- add, adds a new entry to the hash table.

- avg_reception_rate calculates the average of the reception rates for all of the entries contained in the hash table. The value returned from this function is used to determine whether to increase, decrease or maintain the contention window size.

- update, updates the information contained in a table entry for the given MAC address.

- size returns the size of the hash table.

- print displays all of the entries contained in the hash table.

4.5.2 Changes to the 802.11e Source Code

The source code contained in Appendix E.2 provides the media access control for the simulations. While the hash table, `broadcast_table`, provides the ability to estimate the condition of the network, the `broadcast_table` object must be intergrated with the 802.11e MAC layer. Each node maintains a `broadcast_table` object at the MAC layer. When the constructor of the `Mac802_11e` class is invoked a `broadcast_table` object is created.

For each packet that is passed up to the MAC layer from the physical layer, the entry in the `broadcast_table` for that MAC address is updated. For example, when a packet is overheard the sequence number and the packet time is updated. This is how data is gathered to predict the condition of the network.

An `UpdateTableTimer` object is maintained. This object is used to create a timer at the MAC layer for when to adjust the CW size. When the timer maintained by `UpdateTableTimer` expires, the average reception rate is calculated, and the size of the contention window is adjusted based on the change in the reception rate.

The code that resets the contention window was modified. At the end of a successful transmission, the CW is reset to $CW[AC]_{min}$ and this behavior was disable, so that the window is only able to change when the `UpdateTableTimer` expires. If this resetting of the CW was not disabled after every transmission the value for $CW[AC]$ would have been reset to $CW_{min}[AC]$.

4.6 Simulation Results

To determine the effectiveness of the broadcast algorithm, that was proposed in Chapter III, a number of simulations were conducted. The simulations compared the performance of the standard, 802.11e, protocol against an adaptive version of

the protocol. The simulations used the freeway mobility model, which was discussed in Section 4.2. In addition, the metrics used to evaluate the protocols were those discussed in Section 4.4. Also, the network traffic was generated with the `traffic-w-jitter.pl` program that was discussed in Section 4.3.3. In addition, the length of the VANET simulations was set to 60 s.

One additional point before discussing the simulation results, the parameters used for the 802.11e protocol must be configured before the simulations are executed. To configure the traffic classes for the 802.11e protocol, a separate file is used to set the parameters of each access category. The file `priority.tcl`, that is part of the 802.11e package and is located with the other 802.11e source code, configures the parameters for the 802.11e access categories. Furthermore, after the `priority.tcl` file is modified, the ns-2 source code must be recompiled for the changes to take affect. The following are access category values used for the simulations:

```
# parameters for Queue 0
$ifq Prio 0 PF 2
$ifq Prio 0 AIFS 2
$ifq Prio 0 CW_MIN 7
$ifq Prio 0 CW_MAX 31
$ifq Prio 0 TXOPLimit 0.003264
$ifq Prio 0 CW_MAX 15

#parameters for Queue 1
$ifq Prio 1 PF 2
$ifq Prio 1 AIFS 2
$ifq Prio 1 CW_MIN 15
$ifq Prio 1 CW_MAX 127
$ifq Prio 1 TXOPLimit 0.00601

#parameters for Queue 2
$ifq Prio 2 PF 2
$ifq Prio 2 AIFS 3
$ifq Prio 2 CW_MIN 15         ;# aCWmin
$ifq Prio 2 CW_MAX 511        ;# aCWmax
$ifq Prio 2 TXOPLimit 0

#parameters for Queue 3
$ifq Prio 3 PF 2
$ifq Prio 3 AIFS 4
$ifq Prio 3 CW_MIN 31
$ifq Prio 3 CW_MAX 1023
$ifq Prio 3 TXOPLimit 0
```

Table 4.6: Simulation Parameters for the Access Categories

AC	$CW_{min}[AC]$	$CW_{max}[AC]$	$AIFS[AC]$	*Packet Size*
0	7	31	2	500 Bytes
2	15	511	3	250 Bytes

4.6.1 Dynamic Window Simulation

A set of simulations were used to determine the effect that the modified contention window algorithm has on the performance of VANET. The simulation try to accurately depict the actual conditions that will be present when VANETs are finally commercially deployed. The simulations were conducted using the the software components described earlier in this chapter.

The simulation used two classes of traffic AC-0 and AC-2 (originally there was a third class of traffic in the simulations, AC-1, but it was removed because it did not show much insight into the behavior of the contention window in the simulations). As previously mentioned, AC-0 is used to transmit emergency warnings, and the warnings are transmitted once every 100 ms for 1.5 s with 10% jitter. On the other hand, AC-2 is used for periodically transmit a vehicle's state every 100 ms with 10% jitter. The packet sizes vary with AC-0 having 500 *byte* packets and AC-0 having 250 *bytes* packets. In the simulations, AC-0 accounts for 20% of the traffic, and AC-2 accounts for 80% of the traffic. Table 4.6 contains a summary of the 802.11e parameters that were used for the simulation.

The wireless channel was configured to use the values of the DSRC standard. For example, the bandwidth was set to 6 $mbps$, and the frequency was set to 5.9 GHz The rest of values for the physical parameters can be found in the `traffic-w-jitter.pl` source code found in Appendix C.1

A number of different simulation were conducted. Each simulation varied the number of nodes contained in the freeway. By increasing the number of nodes, one is

able to observe the effect that increasing the network traffic has on the performance of the protocols. The simulation varied the number of nodes in the simulation by 200 nodes, and simulation were executed for the freeways containing the following number of nodes: 200, 400, 600, and 800.

Simulation for 200 Node

For the simulation conducted with 200 nodes, this configuration results in a relatively sparse network. For this reason, modifying the size of the contention window has a minor impact on the reception rate. The gain in the reception rate for 200 nodes was marginal for the modified CW algorithm compared to the standard algorithm. In this simulation, because not many nodes are trying to simultaneously access the channel, the access delay remains short. Overall, the modified CW algorithm didn't make much difference in the reception rate because the reception rate was already fairly well.

Simulation for 400 Node

Figure 4.8 contains the reception rate for a simulation with 400 nodes. In addition, Table 4.7 contains the access delay for the different access categories of the simulation. In this simulation, the reception rate slightly increases, for both access categories, when the transmission range is less than 130 m. One surprising finding is the reception rate actually became worse at large distances between nodes. One possible reason for the decreased reception rate are at long distance is the hidden terminal problem. All of the access categories maintained an acceptable access delay. The largest access delay for this approximately 10 ms for AC-2 for the modified CW algorithm. In this case, the access delay was is below the 100 ms target, so the performance of this aspect of the simulation is acceptable.

Table 4.7: Average Access Delay for 400 Nodes

	Standard CW	Modified CW
AC0	0.001018	0.001580
AC2	0.001861	0.010133

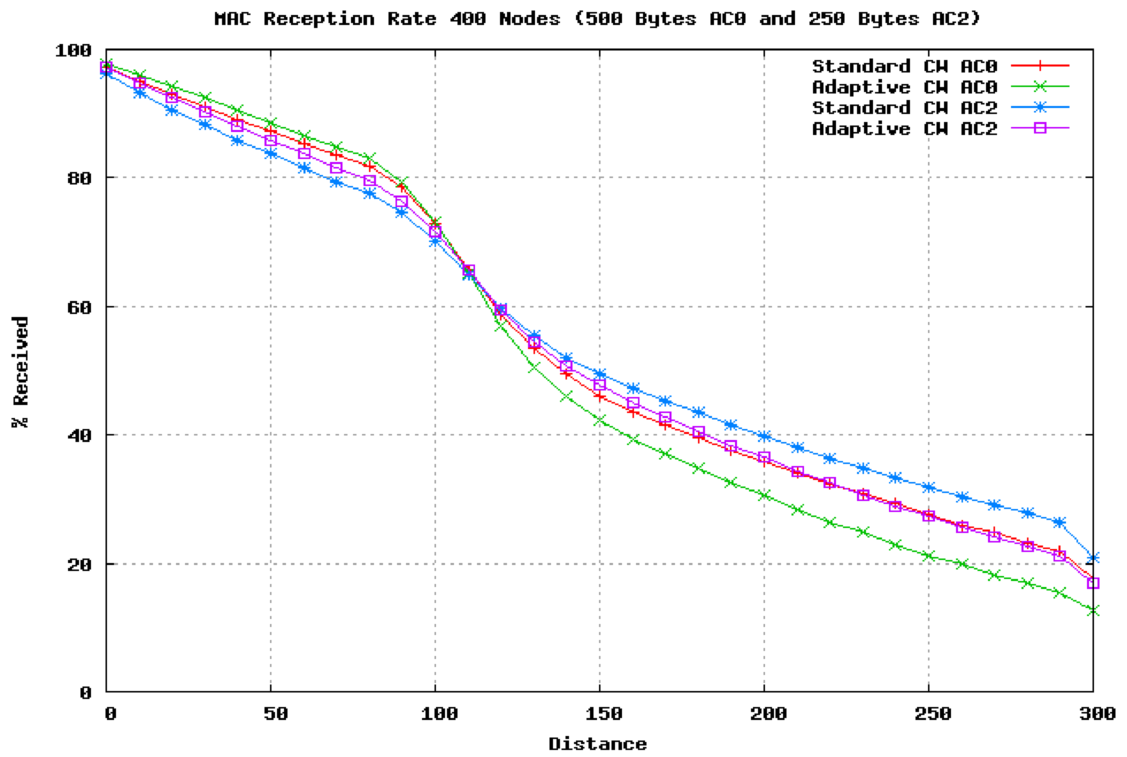

Figure 4.8: Simulation Reception Rate 400 Nodes

Table 4.8: Average Access Delay for 600 Nodes

	Standard CW	Modified CW
AC0	0.002428	0.001580
AC2	0.010687	0.066329

Simulation for 600 Node

Figure 4.9 contains the reception rate for a simulation with 400 nodes. In addition, Table 4.8 contains the access delay for the different access categories of the simulation. The results of the simulation for 600 nodes were similar to the results of the simulation for 400 nodes. The reception rate increased for both access categories at distances close to the sender. On thing that is observable from the simulation is as the traffic on the network in increases, the difference in the reception rate between access categories increases. The access delay for all of the access categories remained acceptable. The largest access delay was 66 ms.

Simulation for 800 Node

A final simulation was conducted with 800 nodes. This simulation models the case where the network becomes saturated with network traffic. The results of the reception rate for the simulation were almost identical to the simulation with 600 nodes. The modified algorithm slightly increased the reception rate of the broadcast traffic for each access category. The problem with this simulation is the access delay for AC-2 of the modified algorithm was greater than 100 ms. One solution to reduce the delay is decrease the size of $CW_{max}[AC]$. For example, if $CW_{max}[2]$ was reduced from 511 to 255 the access delay for the simulation would decrease. One other solution when the network becomes saturated with traffic is to reduce the transmission rate.

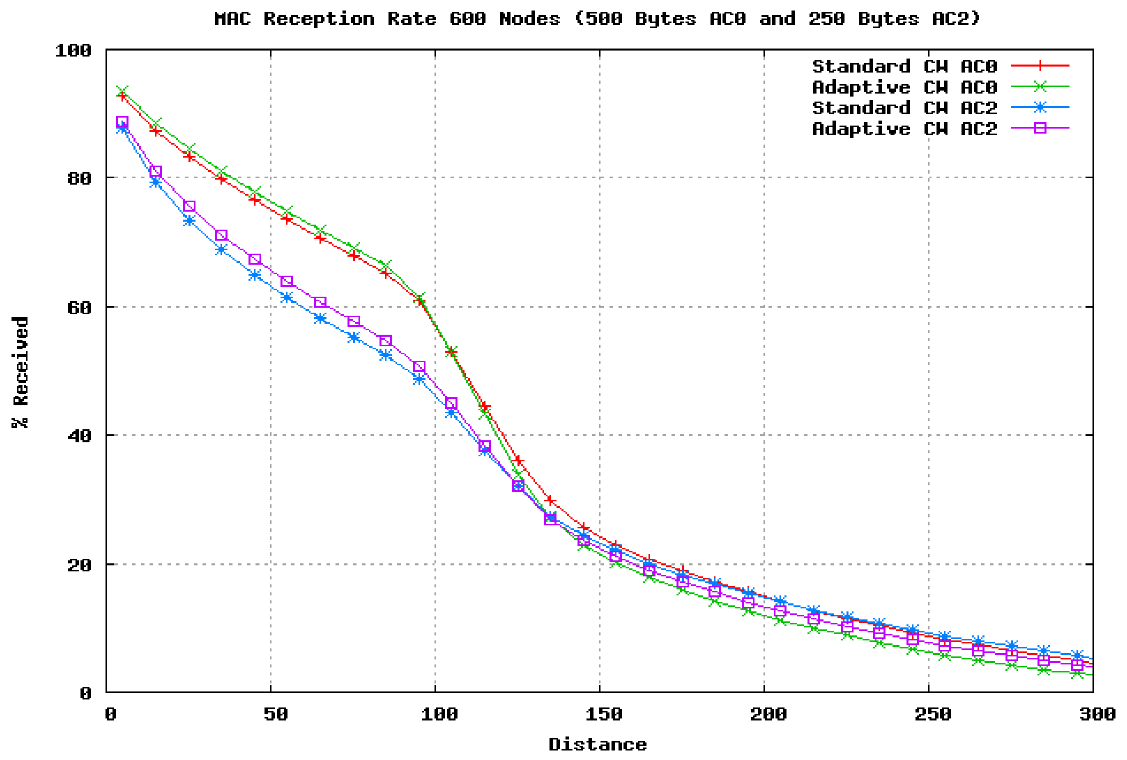

Figure 4.9: Simulation Reception Rate 600 Nodes

4.6.2 Static CW Simulation

The results of the initial simulation of the dynamic contention window did not perform as well as was hoped. To gain further insight into the behavior of the contention window, a number of additional simulation were conducted. In this case, instead of using the modified contention window algorithm, the unmodified IEEE 802.11 protocol was used. Only a single access category was used for each simulation. Network traffic in simulations was generated at a rate of one packet every 100 ms, and the size of the packets are 250 bytes.

The simulations used a fix number of nodes and varied the size of the initial value of the contention window (i.e., CW_{min}). In this case, a node will always select a back-off in the range of [0, CW_{min}]. Simulation were conducted using the following sizes for $CW_{min}[AC]$: 7, 15, 31, 63, 127. 255, 511, 1023.

Figure 4.10 contains the reception rates that are achieved from using different sizes for CW_{min}. The figure only shows the upper and lower bounds on CW to improve visualization of the results. In addition, Figure 4.11 contains the access delay for each contention window size.

The static analysis shows that at certain distances a 5% to 6% increase in the reception rate is achieved by increasing the size of CW_{min} from $CW_{min} = 7$ to $CW_{min} = 511$. Similar to the simulations discussed in Section 4.6.1, a larger value for CW_{min} resulted in a lower reception rate at large distance.

The dynamic CW algorithm results in slightly lower reception rates as compared to using static windows. Dynamically adjusting the contention window improves the rate of reception under certain conditions. If the amount of network traffic is close to the theoretical limit, it would be beneficial instead to drop some packets or to adjust the packet transmission rate. In this case increasing the CW has a limited impact.

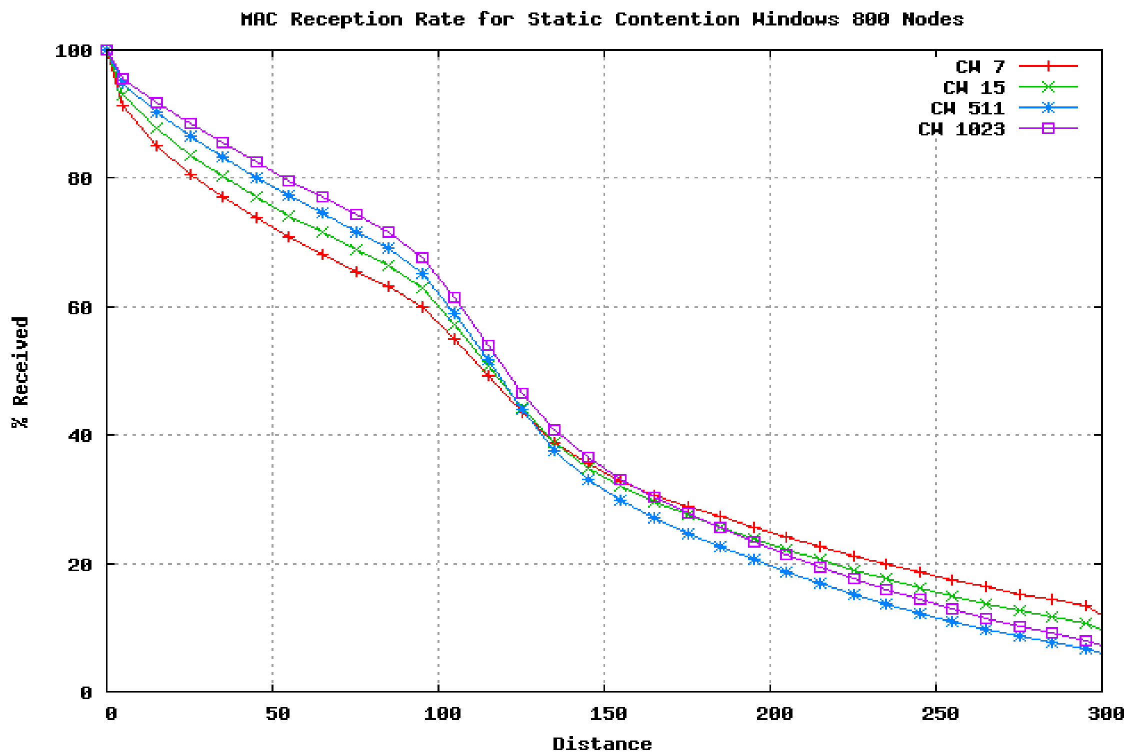

Figure 4.10: Static CW Reception Rate for 800 Nodes

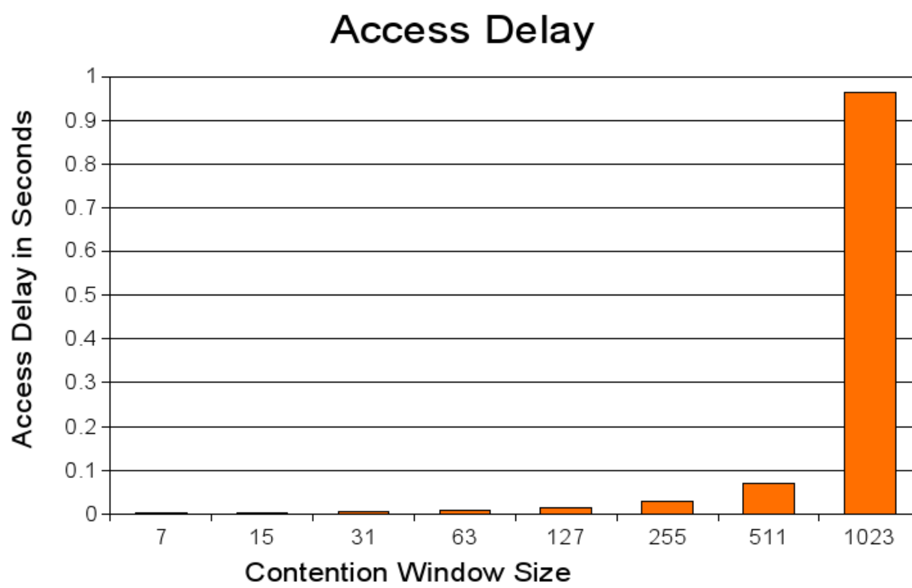

Figure 4.11: Static CW Access Delay for 800 Nodes

CHAPTER V

Conclusion and Further Research

In this chapter some final remarks on the results of the simulations are given along with some future areas of research. To begin, Section 5.1 describes the results of the project. In addition, Section 5.2 gives some possible direction for future research into the area of improving the broadcast reception rate for a VANET.

5.1 Conclusion

Dynamically adjusting the contention window improves the reception rate under certain conditions. The results of the simulations show that adaptively adjusting the contention window can increase the reception rate of broadcast frames by approximately 5% when there is moderate network traffic. It is unlikely to expect much better results than a 5% improvement in the reception rate, under the typical conditions found in a VANET.

In the extreme cases of very light or very heavy traffic, the adaptive broadcast protocol has a limited impact on the reception rate. On the other hand, the adaptive protocol shows promising results when the network traffic is moderate. The proposed additions to the boradcast protocol are not resource intensive or difficult to implement, as a result, implementing the suggested changes provides of increasing the performance of the 802.11 broadcast protocol for a VANET.

In the case of *light network traffic*, dynamically adjusting the contention window has little impact on the reception rate. The primary reason why the protocol does not help is because the reception rate is already quite high when the traffic is light, so not much room is left for improvement. For example it is not unreasonable to find reception rates of 95% when traffic is very light and only a few nodes are transmitting. In addtion, the adaptive protocol marginally affects the access delay when traffic is light. In this case, the access delay only slightly increases, and the access delay remains at an accpetable level. To sum it up, when network traffic is light the reception rate remains virtually the same and the access delay only slightly increases, so the protocol shows no performance gain but there is no major drawbacks to using it.

In the case of *extremely heavy network traffic*, the performance of the adaptive broadcast protocol can actually become worse. The reception rate remains approximately the same when the network is saturated with traffic. For example, if nodes attempt to transmit 15 *mbps* of traffic on a 6 *mbps* channel, increasing the size of CW has virtually no effect on improving the reliability. The reason for this is each node that is attempting to transmit will probably, at most, be able to allow one time-slot of the back-off to expire before a transmission occurs. Increasing the size of the contention window when the network is overloaded with traffic only results in increased access delays. As a result, increasing the CW for a saturated network results in the time it takes to access the channel increasing dramatically. For instance, if the maximum size of the contention window ($CW[AC]_{max}$) is too large the delay experienced before a transmission can become unacceptable. For example, the in some of the simulations if an access category had a maximum contention window size of 1023 time-slots and more traffic was attempted to be transmitted into the

network than it could handle, the access delay became a second or more.

A strange phenomenon found during the simulations is reception rate actually started to get worse as the distance from the sender increased. The adaptive protocol improves the reception rate for nodes that are close to the transmitter. For instance, if the transmission range of a node was 300 m, from approximately [0 m, 150 m] the reception rate improves and from approximately [151 m, 300 m] the reception rate actually becomes worse. The cause for the poor reception rates at large distances need to be further investigated. The hidden terminal problem becomes more intense as the distance between nodes increases. One of the causes leading to the poorer reception rates may be the hidden terminal problem, but at the present time it is unknown for certain what causes the decreased performance.

Even with the modification to the IEEE 802.11 protocol, the broadcast transmission are still unreliable. Without developing a broadcast reservation protocol for the wireless channel, any changes to the inter-frame space and contention window will have a limited impact on increasing the probability of reception. Furthermore, a drawback of increasing these times is it results in a decreased amount time that traffic can actually be handled by the channel. If for a large portion of time the channel remains ideal, because of the back-offs and inter-frame space, than the bandwidth ineffectively used.

The adaptive protocol does not solve all of the problems associated with broadcasting in a VANET. The biggest problem affecting the reliability of a broadcast protocol is the hidden terminal problem. The optimal solution to the problem would be to develop a reservation scheme for broadcast transmission for a VANET. However, developing a reservation scheme for broadcast transmission is still an open problem, and it doesn't appear that a solution to this problem will be found any

time soon. For this reason, other solutions that aim at incrementally increase the performance of broadcast transmission should be futher explored.

If the amount of network traffic is close to the theoretical limit, it would be beneficial to employ another method to increase message reception. One possible solution is to throttle the rate that vehicle's state is transmitted. Instead of continuing to send traffic into a saturated channel and further contributing to contention and congestion, the transmission rate can be adjusted. Another possibility is to randomly drop some packets instead of changing the transmission rate (i.e., using a method similar to Random early detection (RED)). In this case increasing the CW has a limited impact.

5.2 Future Work

A number of additional modifications can be made to the 802.11 broadcast protocol to improve the reliability. The following are some areas of future work to improve the performance of the broadcast protocol for a VANET:

- **Adaptive transmission rate**, an algorithm that throttles the rate at which the vehicle's state is transmitted.
- **Adaptive transmission range**, the transmission range can be adjusted to keep the network load on the channel below a certain threshold.
- **Mathematical mode**, the simulation used a heuristic method to evaluate the broadcast protocol, a model could be developed to determine the maximum theoretical improvement that can be expected from adjusting the contention window.

5.2.1 Adaptive Transmission Rate Control

Due to the hidden terminal problem and other interference, it is unrealistic to achieve a 100% delivery rate without retransmission in a wireless network. Safety messages typically need to be repeatedly transmitted at a certain rate to ensure reliable delivery. For example, in the event of an emergency, Emergency Warning Messages are repeatedly broadcasted to ensure delievery. Similarly, a vehicle broadcasts its state information every $100\,ms$. Broadcasting a message multiple times increases the probability of reception, but also increases the load on the network.

If the network is highly loaded, increasing the contention window alone will not be effective and may result in very high delivery latency. In this case, a node will have to decrease its transmission rate. The authors of [27] propose the VCWC protocol to transmit emergency warning messages, which is only based on application-specific properties to help controlling channel congestion. When an accident first occurs, the vehicle starts transmitting emergency warning messages at the maximum rate and over time decreases the rate. This approach focuses only on the delivery of emergency messages, while ignoring other public safety and private messages.

We will utilize the aforementioned channel feedback, packet collision rate and number of nodes within transmission range, to effectively adjust the transmission rates for all traffic classes. For example, when the packet loss rate is larger than some threshold, we then first minimize the transmission rate of all traffic classes. If this is not enough, we will further drop all non-safety related messages and reduce the transmission rate of low priority safety messages, and so on.

5.2.2 Dynamic Transmission Power Control

Controlling the communication range, by adjusting the transmission power, can be used to mitigate the adverse effects caused by nodes being densly populated. The choice of the communication range has a direct impact on a fundamental property of an ad-hoc network, the connectivity. In a VANET, a static transmission range cannot maintain the networks connectivity due to the non-homogeneous conditions. It is shown in [11, 20] that a dynamic transmission range is needed to maintain connectivity in non-homogeneous networks to take advantage of power saving and increased capacity.

In [11], a dynamic transmission range based on estimation of vehicle density is proposed. Their approach assumes that each vehicle has an OBU to access the network. However, this is very implausible. The Vehicle Safety Communication technology will likely be initially deployed on a small percentage of vehicles. Moreover, the deployment is also likely to be unbalanced. It is very likely that some areas will have higher deployment rates than others. With the same high vehicle density, it is possible that the VANET may be highly congested in one area and mostly idle in another area. In the later case, if the transmission power is mistakenly reduced, the VANET might not maintain basic connectivity. Thus, we propose dynamically adjusting the transmission power based on the actual network condition instead of vehicle density.

The transmission range of all nodes can be adjusted, using the power control, to keep the network load below a certain threshold. By adjusting the transmission range to the minimum range required by a safety application, the load on the channel can be reduced as a result of having highly accurate information of neighboring vehicles.

APPENDICES

APPENDIX A

Simulation Script

A.1 wireless-simulation.tcl

```
#======================================================================
#
#  Wireless simulation, a simulation of a broadcast protocol
#  for vehicular ad hoc networks.
#
#  Nathan Balon
#  University of Michigan - Dearborn
#  CIS 695
#
#  Created: 5/17/2006
#
#======================================================================


#======================================================================
# Options
#======================================================================

set opt(chan)       Channel/WirelessChannel   ;# channel type
set opt(prop)       Propagation/TwoRayGround  ;# radio-propagation model
set opt(ant)        Antenna/OmniAntenna       ;# antenna type
set opt(ll)         LL                        ;# link layer
set opt(mac)        Mac/802_11e               ;# MAC type
set opt(ifq)        Queue/DTail/PriQ          ;# interface queue type
set opt(ifqlen)     50                        ;# max packet in ifq
set opt(netif)      Phy/WirelessPhy           ;# network interface type
set opt(rp)         DumbAgent                 ;# routing protocol
set opt(nn)         0                         ;# number of mobile nodes
set opt(pkt_size)   250                       ;# size of  broadcast message
set opt(pkt_size_emer) 500                    ;# emergency packet size
set opt(sc)         ""                        ;# mobility scenario
set opt(cp)         ""                        ;# connection pattern
set opt(tr)         trace.tr                  ;# trace file
set opt(nam_tr)     ""                        ;# name trace file
set opt(seed)     0                         ;# seed the random number generator
set opt(stop)       65.0                      ;# time to end simulation
set opt(x)          1000                      ;# x size for topology
set opt(y)          1000                      ;# y size for topology
set opt(lm)         ON                        ;# log movement
set opt(at)         ON                        ;# agent trace
```

```
set opt(rt)         OFF                        ;# routing trace
set opt(mact)       ON                         ;# mac trace
set opt(movt)       ON                         ;# movement trace
set opt(modified)   2                          ;# use modified broadcast
set opt(sliding)    0.05                       ;# threashold to slide CW


#============================================================================

# unity gain, omni-directional antennas
# set up the antennas to be centered in the node and 1.5 meters above itns_
Antenna/OmniAntenna set X_ 0
Antenna/OmniAntenna set Y_ 0
Antenna/OmniAntenna set Z_ 1.5
Antenna/OmniAntenna set Gt_ 4.0
Antenna/OmniAntenna set Gr_ 4.0


#original settings
Phy/WirelessPhy set CPThresh_ 6.0
Phy/WirelessPhy set CSThresh_ 2.5118864e-13    ;# -96 dBm
Phy/WirelessPhy set RXThresh_ 1.0e-12          ;# -90 dBm
Phy/WirelessPhy set bandwidth_ 6.0e6
Phy/WirelessPhy set Pt_ 0.0003754
Phy/WirelessPhy set freq_ 5.9e+9
Phy/WirelessPhy set L_ 1.0

# slot times
Mac/802_11 set SlotTime_      0.000013        ;# 13us
Mac/802_11 set SIFS_          0.000032        ;# 32us

# physical layers headers and rates
Mac/802_11 set PreambleLength_       32      ;# 144 bit
Mac/802_11 set PLCPHeaderLength_     40      ;# 48 bits
Mac/802_11 set PLCPDataRate_  6.0e6          ;# 6Mbps

Queue/DTail set drop_front_ false
Queue/DTail set summarystats_ false
Queue/DTail set queue_in_bytes_ false
Queue/DTail set mean_pktsize_ 250
Queue/DTail/PriQ set Prefer_Routing_Protocols    1
Queue/DTail/PriQ set Max_Levels   4
Queue/DTail/PriQ set Levels    4

Mac/802_11e set cfb_ 0


Mac/802_11e set RTSThreshold_ 3000         ;# bytes
Mac/802_11e set ShortRetryLimit_       7   ;# retransmittions
Mac/802_11e set LongRetryLimit_        4   ;# retransmissions

# added parameters to 802.11e
Mac/802_11e set update_interval_   0.4  ;# interval to adjust the CW
Mac/802_11e set timeout_entries_   0.8  ;# remove nodes if not heard from in this time
Mac/802_11e set WMA_alpha_         0.8  ;# alpha used in the weighted average
Mac/802_11e set scaling_factor_    2.0  ;# how much the CW should be increased by

set BROADCAST_ADDR -1        ;# broadcast address
```

```
set MESSAGE_PORT 42          ;# periodic message port
set WARNING_PORT 33          ;# warning message
set EMER_PORT 21             ;# port to listen for emergency messages

#=============================================================================
#    Functions
#============================================================================

# Display program usage
proc usage { argv0 }  {
puts "Usage: $argv0"
puts "\tmandatory arguments:"
puts "\t\t\[-x MAXX\] \[-y MAXY\]"
puts "\t\t\[-cp conn pattern\] \[-sc scenario\] \[-nn nodes\]"
puts "\toptional arguments:"
        puts "\t\t\[-seed seed\] \[-stop sec\] \[-tr tracefile\]\n"
        puts "\t\t\[-pkt_size size\] \[-nam_tr nam trace\]\n"
puts "\t\t\[-modified 1 or 0\] \[-sliding threashold\]\n"
}

# set the options from the command line arguments
proc getopt {argc argv} {
global opt
lappend optlist cp nn seed sc stop tr x y pkt_size pkt_size_emer nam_tr modified sliding

for {set i 0} {$i < $argc} {incr i} {
set arg [lindex $argv $i]
if {[string range $arg 0 0] != "-"} continue

if {[string range $arg 0 0] != "-"} continue

set name [string range $arg 1 end]
set opt($name) [lindex $argv [expr $i+1]]
}
}

# perform clean up at the end of the program
proc finish {} {
        global ns_ tracefd namtracefd opt
        $ns_ flush-trace
        close $tracefd
        if {$opt(nam_tr) != ""} {
                close $namtracefd
        }
        $ns_ halt
        exit 0
}

# log the movement of a node every 0.1 seconds
proc log-movement {} {
        global logtimer ns_ ns

        set ns $ns_
        source ../tcl/mobility/timer.tcl
        Class LogTimer -superclass Timer
        LogTimer instproc timeout {} {
                global opt node_;
                for {set i 0} {$i < $opt(nn)} {incr i} {
                        $node_($i) log-movement
```

```
                }
                $self sched 0.05
        }
        set logtimer [new LogTimer]
        $logtimer sched 0.05
}

# ======================================================================
# Main Program
# ======================================================================

#get command line arguments
getopt $argc $argv

if { $opt(nn) == 0 || $opt(sc) == "" || $opt(cp) == ""} {
usage $argv0
exit 1
}
puts "x: $opt(x), y: $opt(y)"

Mac/802_11e set number_nodes_ $opt(nn)            ;# use modified broadcast algorithm
Mac/802_11e set modified_ $opt(modified)          ;# use modifed algorithm
Mac/802_11e set sliding_threshold_ $opt(sliding)  ;# threshold to slide CW

if {$opt(seed) > 0} {
puts "Seeding Random number generator with $opt(seed)\n"
ns-random $opt(seed)
}

# create a new simulator
set ns_ [new Simulator]

# set up the traces
set tracefd [open $opt(tr) w]
$ns_ trace-all $tracefd

# Set up the nam trace if desired.
if {$opt(nam_tr) != ""} {
        set namtracefd [open $opt(nam_tr) w]
        $ns_ namtrace-all-wireless  $namtracefd $opt(x) $opt(y)
}
# set the topology
set topo [new Topography]
$topo load_flatgrid $opt(x) $opt(y)

set god_ [ create-god $opt(nn) ]
set chan_ [new Channel/WirelessChannel]

# configure the nodes of the simulation
$ns_ node-config -adhocRouting $opt(rp) \
                         -llType $opt(ll) \
                         -macType $opt(mac) \
                         -ifqType $opt(ifq) \
                         -ifqLen $opt(ifqlen) \
                         -antType $opt(ant) \
                         -propInstance [new $opt(prop)] \
                         -phyType $opt(netif) \
                         -topoInstance $topo \
                         -channel $chan_ \
```

```
                        -agentTrace $opt(at) \
                        -routerTrace $opt(rt) \
                        -macTrace $opt(mact) \
                        -movementTrace $opt(movt)

###############################################################################


# PeriodicBroadcast class used to send broadcast message to a nodes one-hop neighbors
Class Agent/MessagePassing/PeriodicBroadcast -superclass Agent/MessagePassing
Agent/MessagePassing/PeriodicBroadcast instproc recv {source sport size data} {
    # This empty function is needed so receive works.
}

Agent/MessagePassing/PeriodicBroadcast instproc send_message {} {
    $self instvar node_
    global ns_ MESSAGE_PORT BROADCAST_ADDR opt

    # send the broadcast message
    $self sendto $opt(pkt_size) 0 $BROADCAST_ADDR $MESSAGE_PORT
}

################################################################################

Class Agent/MessagePassing/EmerBroadcast -superclass Agent/MessagePassing
Agent/MessagePassing/EmerBroadcast instproc recv {source sport size data} {
    # This empty function is needed so receive works.
}

Agent/MessagePassing/EmerBroadcast instproc send_message {} {
    $self instvar node_
    global ns_ EMER_PORT BROADCAST_ADDR opt

    # send the broadcast message
    $self sendto $opt(pkt_size_emer) 0 $BROADCAST_ADDR $EMER_PORT
}

################################################################################

# Create the mobile nodes for the simulation.
for {set i 0} {$i < $opt(nn)} {incr i} {
    set node_($i) [$ns_ node]
    $ns_ initial_node_pos $node_($i) 40
}


# Source the mobility scenario file.
source $opt(sc)

# Attach a new Agent/MessagePassing/PeriodicBroadcast to each node on port $MESSAGE_PORT
for {set i 0} {$i < $opt(nn)} {incr i} {
    set bc_agent($i) [new Agent/MessagePassing/PeriodicBroadcast]
    $bc_agent($i) set class_ 2
    $bc_agent($i) set prio_ 2

    $node_($i) attach  $bc_agent($i) $MESSAGE_PORT
}

# Attach a new Agent/MessagePassing/EmerBroadcast to each node on port $EMER_PORT
```

```
for {set i 0} {$i < $opt(nn)} {incr i} {
    set em_agent($i) [new Agent/MessagePassing/EmerBroadcast]
    $em_agent($i) set class_ 0
    $em_agent($i) set prio_ 0
    $node_($i) attach  $em_agent($i) $EMER_PORT
}



if { $opt(lm) == "ON" } {
    puts "Logging movement..."
    log-movement
}

# Source the file that contains the broadcast traffic.
puts "sourcing traffic file"
source $opt(cp)


$ns_ at $opt(stop) "finish"

# Add to the trace simulation parameters.
puts $tracefd "M 0.0 nn $opt(nn) x $opt(x) y $opt(y) rp $opt(rp)"
puts $tracefd "M 0.0 sc $opt(sc) cp $opt(cp) seed $opt(seed)"
puts $tracefd "M 0.0 prop $opt(prop) ant $opt(ant)"

puts "Starting Simulation..."
$ns_ run
```

APPENDIX B

Mobility

B.1 create-map.pl

```
#! /usr/bin/perl
#
#  create-map.pl: creates a circular road map for the freeway model.
#  The map is then read by the mobility generator found at
#  http://nile.usc.edu/important/software.htm, to create the mobility
#  for nodes in an ns2 simulation.
#
#  Nathan Balon
#  University of Michigan - Dearborn
#

use strict;
use warnings;

use constant PI => 3.1415926535;
use constant LANE_DIST => 5;   # distance between lane

# parameter used to generate a circular road
my $nodes = 24;                # the number of points on the circle
my $inner_edge_dist = 300;     # the distance to the inner lane
my $traffic_direction = 2;     # direction of traffic (either 1 or 2)
my $lanes = 8;                 # number of lanes
my $lane_dist = 0;             # distance between the inner lane
my @center = (500,500);        # the center point of the circular road
my $min_velocity = 17.0;       # minimum velocity of the nodes
my $max_velocity = 25.0;       # maximum velocity of the nodes

my $theta_const = 2 * PI / $nodes;  # angle between nodes
my $theta = 0;                      # theta value for the current node

# Check that an even number of lanes exist if traffic
# flows in both directions on the road.
if($lanes % 2 != 0 && $traffic_direction == 2){
    die "ERROR: if traffic flows in both direction, you must " .
        "have an even number of lanes\n";
}

# display the information about the freeway
print "FREEWAY\n";
print "FREEWAY_NUM 1\n";
```

```
print "LANE_NUM $lanes\n";

for(my $i = 0; $i < $lanes; $i++,$lane_dist += LANE_DIST){
    $theta = 0;
    my $direction = 1;
    # set the phase of the traffic for traffic to
    # flow in the opposite direction for half the lanes
    if($i/$lanes >= 0.5 && $traffic_direction == 2){
        $direction = -1;
    }

    # create a new lane in the map
    print "LANE_BEGIN 0 $i $i $direction $nodes\n";

    # determine the position for two nodes connected by an edge
    # and display the results
    for(my $j = 0; $j < $nodes; $j++){
        my $next_x = 0;
        my $next_y = 0;

        my $initial_x = $center[0] + cos($theta) * ($inner_edge_dist + $lane_dist);
        my $initial_y = $center[1] + sin($theta) * ($inner_edge_dist + $lane_dist);
        if($direction == 1){
            $next_x = $center[0] + cos($theta + $theta_const) *
                                   ($inner_edge_dist + $lane_dist);
            $next_y = $center[1] + sin($theta + $theta_const) *
                                   ($inner_edge_dist + $lane_dist);
            $theta += $theta_const;
        }else{
            # traffic going in the opposite direction
            $next_x = $center[0] + cos($theta - $theta_const) *
                                   ($inner_edge_dist + $lane_dist);
            $next_y = $center[1] + sin($theta - $theta_const) *
                                   ($inner_edge_dist + $lane_dist);
            $theta -= $theta_const;
        }
        printf("PHASE %d \(%4.5f\, %4.5f\) \(%4.5f\, %4.5f\) %3.3f %3.3f\n",
              $j, $initial_x, $initial_y, $next_x, $next_y, $min_velocity, $max_velocity);
    }
}
```

B.2 map.txt

```
FREEWAY
FREEWAY_NUM 1
LANE_NUM 8
LANE_BEGIN 0 0 0 1 24
PHASE 0 (800.00000, 500.00000) (789.77775, 577.64571) 17.000 25.000
PHASE 1 (789.77775, 577.64571) (759.80762, 650.00000) 17.000 25.000
PHASE 2 (759.80762, 650.00000) (712.13203, 712.13203) 17.000 25.000
PHASE 3 (712.13203, 712.13203) (650.00000, 759.80762) 17.000 25.000
PHASE 4 (650.00000, 759.80762) (577.64571, 789.77775) 17.000 25.000
PHASE 5 (577.64571, 789.77775) (500.00000, 800.00000) 17.000 25.000
PHASE 6 (500.00000, 800.00000) (422.35429, 789.77775) 17.000 25.000
PHASE 7 (422.35429, 789.77775) (350.00000, 759.80762) 17.000 25.000
PHASE 8 (350.00000, 759.80762) (287.86797, 712.13203) 17.000 25.000
PHASE 9 (287.86797, 712.13203) (240.19238, 650.00000) 17.000 25.000
```

```
PHASE 10 (240.19238, 650.00000) (210.22225, 577.64571) 17.000 25.000
PHASE 11 (210.22225, 577.64571) (200.00000, 500.00000) 17.000 25.000
PHASE 12 (200.00000, 500.00000) (210.22225, 422.35429) 17.000 25.000
PHASE 13 (210.22225, 422.35429) (240.19238, 350.00000) 17.000 25.000
PHASE 14 (240.19238, 350.00000) (287.86797, 287.86797) 17.000 25.000
PHASE 15 (287.86797, 287.86797) (350.00000, 240.19238) 17.000 25.000
PHASE 16 (350.00000, 240.19238) (422.35429, 210.22225) 17.000 25.000
PHASE 17 (422.35429, 210.22225) (500.00000, 200.00000) 17.000 25.000
PHASE 18 (500.00000, 200.00000) (577.64571, 210.22225) 17.000 25.000
PHASE 19 (577.64571, 210.22225) (650.00000, 240.19238) 17.000 25.000
PHASE 20 (650.00000, 240.19238) (712.13203, 287.86797) 17.000 25.000
PHASE 21 (712.13203, 287.86797) (759.80762, 350.00000) 17.000 25.000
PHASE 22 (759.80762, 350.00000) (789.77775, 422.35429) 17.000 25.000
PHASE 23 (789.77775, 422.35429) (800.00000, 500.00000) 17.000 25.000
LANE_BEGIN 0 1 1 1 24
PHASE 0 (805.00000, 500.00000) (794.60738, 578.93981) 17.000 25.000
PHASE 1 (794.60738, 578.93981) (764.13775, 652.50000) 17.000 25.000
PHASE 2 (764.13775, 652.50000) (715.66757, 715.66757) 17.000 25.000
PHASE 3 (715.66757, 715.66757) (652.50000, 764.13775) 17.000 25.000
PHASE 4 (652.50000, 764.13775) (578.93981, 794.60738) 17.000 25.000
PHASE 5 (578.93981, 794.60738) (500.00000, 805.00000) 17.000 25.000
PHASE 6 (500.00000, 805.00000) (421.06019, 794.60738) 17.000 25.000
PHASE 7 (421.06019, 794.60738) (347.50000, 764.13775) 17.000 25.000
PHASE 8 (347.50000, 764.13775) (284.33243, 715.66757) 17.000 25.000
PHASE 9 (284.33243, 715.66757) (235.86225, 652.50000) 17.000 25.000
PHASE 10 (235.86225, 652.50000) (205.39262, 578.93981) 17.000 25.000
PHASE 11 (205.39262, 578.93981) (195.00000, 500.00000) 17.000 25.000
PHASE 12 (195.00000, 500.00000) (205.39262, 421.06019) 17.000 25.000
PHASE 13 (205.39262, 421.06019) (235.86225, 347.50000) 17.000 25.000
PHASE 14 (235.86225, 347.50000) (284.33243, 284.33243) 17.000 25.000
PHASE 15 (284.33243, 284.33243) (347.50000, 235.86225) 17.000 25.000
PHASE 16 (347.50000, 235.86225) (421.06019, 205.39262) 17.000 25.000
PHASE 17 (421.06019, 205.39262) (500.00000, 195.00000) 17.000 25.000
PHASE 18 (500.00000, 195.00000) (578.93981, 205.39262) 17.000 25.000
PHASE 19 (578.93981, 205.39262) (652.50000, 235.86225) 17.000 25.000
PHASE 20 (652.50000, 235.86225) (715.66757, 284.33243) 17.000 25.000
PHASE 21 (715.66757, 284.33243) (764.13775, 347.50000) 17.000 25.000
PHASE 22 (764.13775, 347.50000) (794.60738, 421.06019) 17.000 25.000
PHASE 23 (794.60738, 421.06019) (805.00000, 500.00000) 17.000 25.000
LANE_BEGIN 0 2 2 1 24
PHASE 0 (810.00000, 500.00000) (799.43701, 580.23390) 17.000 25.000
PHASE 1 (799.43701, 580.23390) (768.46788, 655.00000) 17.000 25.000
PHASE 2 (768.46788, 655.00000) (719.20310, 719.20310) 17.000 25.000
PHASE 3 (719.20310, 719.20310) (655.00000, 768.46788) 17.000 25.000
PHASE 4 (655.00000, 768.46788) (580.23390, 799.43701) 17.000 25.000
PHASE 5 (580.23390, 799.43701) (500.00000, 810.00000) 17.000 25.000
PHASE 6 (500.00000, 810.00000) (419.76610, 799.43701) 17.000 25.000
PHASE 7 (419.76610, 799.43701) (345.00000, 768.46788) 17.000 25.000
PHASE 8 (345.00000, 768.46788) (280.79690, 719.20310) 17.000 25.000
PHASE 9 (280.79690, 719.20310) (231.53212, 655.00000) 17.000 25.000
PHASE 10 (231.53212, 655.00000) (200.56299, 580.23390) 17.000 25.000
PHASE 11 (200.56299, 580.23390) (190.00000, 500.00000) 17.000 25.000
PHASE 12 (190.00000, 500.00000) (200.56299, 419.76610) 17.000 25.000
PHASE 13 (200.56299, 419.76610) (231.53212, 345.00000) 17.000 25.000
PHASE 14 (231.53212, 345.00000) (280.79690, 280.79690) 17.000 25.000
PHASE 15 (280.79690, 280.79690) (345.00000, 231.53212) 17.000 25.000
PHASE 16 (345.00000, 231.53212) (419.76610, 200.56299) 17.000 25.000
PHASE 17 (419.76610, 200.56299) (500.00000, 190.00000) 17.000 25.000
PHASE 18 (500.00000, 190.00000) (580.23390, 200.56299) 17.000 25.000
```

```
PHASE 19 (580.23390, 200.56299) (655.00000, 231.53212) 17.000 25.000
PHASE 20 (655.00000, 231.53212) (719.20310, 280.79690) 17.000 25.000
PHASE 21 (719.20310, 280.79690) (768.46788, 345.00000) 17.000 25.000
PHASE 22 (768.46788, 345.00000) (799.43701, 419.76610) 17.000 25.000
PHASE 23 (799.43701, 419.76610) (810.00000, 500.00000) 17.000 25.000
LANE_BEGIN 0 3 3 1 24
PHASE 0 (815.00000, 500.00000) (804.26664, 581.52800) 17.000 25.000
PHASE 1 (804.26664, 581.52800) (772.79800, 657.50000) 17.000 25.000
PHASE 2 (772.79800, 657.50000) (722.73864, 722.73864) 17.000 25.000
PHASE 3 (722.73864, 722.73864) (657.50000, 772.79800) 17.000 25.000
PHASE 4 (657.50000, 772.79800) (581.52800, 804.26664) 17.000 25.000
PHASE 5 (581.52800, 804.26664) (500.00000, 815.00000) 17.000 25.000
PHASE 6 (500.00000, 815.00000) (418.47200, 804.26664) 17.000 25.000
PHASE 7 (418.47200, 804.26664) (342.50000, 772.79800) 17.000 25.000
PHASE 8 (342.50000, 772.79800) (277.26136, 722.73864) 17.000 25.000
PHASE 9 (277.26136, 722.73864) (227.20200, 657.50000) 17.000 25.000
PHASE 10 (227.20200, 657.50000) (195.73336, 581.52800) 17.000 25.000
PHASE 11 (195.73336, 581.52800) (185.00000, 500.00000) 17.000 25.000
PHASE 12 (185.00000, 500.00000) (195.73336, 418.47200) 17.000 25.000
PHASE 13 (195.73336, 418.47200) (227.20200, 342.50000) 17.000 25.000
PHASE 14 (227.20200, 342.50000) (277.26136, 277.26136) 17.000 25.000
PHASE 15 (277.26136, 277.26136) (342.50000, 227.20200) 17.000 25.000
PHASE 16 (342.50000, 227.20200) (418.47200, 195.73336) 17.000 25.000
PHASE 17 (418.47200, 195.73336) (500.00000, 185.00000) 17.000 25.000
PHASE 18 (500.00000, 185.00000) (581.52800, 195.73336) 17.000 25.000
PHASE 19 (581.52800, 195.73336) (657.50000, 227.20200) 17.000 25.000
PHASE 20 (657.50000, 227.20200) (722.73864, 277.26136) 17.000 25.000
PHASE 21 (722.73864, 277.26136) (772.79800, 342.50000) 17.000 25.000
PHASE 22 (772.79800, 342.50000) (804.26664, 418.47200) 17.000 25.000
PHASE 23 (804.26664, 418.47200) (815.00000, 500.00000) 17.000 25.000
LANE_BEGIN 0 4 4 -1 24
PHASE 0 (820.00000, 500.00000) (809.09626, 417.17791) 17.000 25.000
PHASE 1 (809.09626, 417.17791) (777.12813, 340.00000) 17.000 25.000
PHASE 2 (777.12813, 340.00000) (726.27417, 273.72583) 17.000 25.000
PHASE 3 (726.27417, 273.72583) (660.00000, 222.87187) 17.000 25.000
PHASE 4 (660.00000, 222.87187) (582.82209, 190.90374) 17.000 25.000
PHASE 5 (582.82209, 190.90374) (500.00000, 180.00000) 17.000 25.000
PHASE 6 (500.00000, 180.00000) (417.17791, 190.90374) 17.000 25.000
PHASE 7 (417.17791, 190.90374) (340.00000, 222.87187) 17.000 25.000
PHASE 8 (340.00000, 222.87187) (273.72583, 273.72583) 17.000 25.000
PHASE 9 (273.72583, 273.72583) (222.87187, 340.00000) 17.000 25.000
PHASE 10 (222.87187, 340.00000) (190.90374, 417.17791) 17.000 25.000
PHASE 11 (190.90374, 417.17791) (180.00000, 500.00000) 17.000 25.000
PHASE 12 (180.00000, 500.00000) (190.90374, 582.82209) 17.000 25.000
PHASE 13 (190.90374, 582.82209) (222.87187, 660.00000) 17.000 25.000
PHASE 14 (222.87187, 660.00000) (273.72583, 726.27417) 17.000 25.000
PHASE 15 (273.72583, 726.27417) (340.00000, 777.12813) 17.000 25.000
PHASE 16 (340.00000, 777.12813) (417.17791, 809.09626) 17.000 25.000
PHASE 17 (417.17791, 809.09626) (500.00000, 820.00000) 17.000 25.000
PHASE 18 (500.00000, 820.00000) (582.82209, 809.09626) 17.000 25.000
PHASE 19 (582.82209, 809.09626) (660.00000, 777.12813) 17.000 25.000
PHASE 20 (660.00000, 777.12813) (726.27417, 726.27417) 17.000 25.000
PHASE 21 (726.27417, 726.27417) (777.12813, 660.00000) 17.000 25.000
PHASE 22 (777.12813, 660.00000) (809.09626, 582.82209) 17.000 25.000
PHASE 23 (809.09626, 582.82209) (820.00000, 500.00000) 17.000 25.000
LANE_BEGIN 0 5 5 -1 24
PHASE 0 (825.00000, 500.00000) (813.92589, 415.88381) 17.000 25.000
PHASE 1 (813.92589, 415.88381) (781.45826, 337.50000) 17.000 25.000
PHASE 2 (781.45826, 337.50000) (729.80970, 270.19030) 17.000 25.000
```

```
PHASE 3 (729.80970, 270.19030) (662.50000, 218.54174) 17.000 25.000
PHASE 4 (662.50000, 218.54174) (584.11619, 186.07411) 17.000 25.000
PHASE 5 (584.11619, 186.07411) (500.00000, 175.00000) 17.000 25.000
PHASE 6 (500.00000, 175.00000) (415.88381, 186.07411) 17.000 25.000
PHASE 7 (415.88381, 186.07411) (337.50000, 218.54174) 17.000 25.000
PHASE 8 (337.50000, 218.54174) (270.19030, 270.19030) 17.000 25.000
PHASE 9 (270.19030, 270.19030) (218.54174, 337.50000) 17.000 25.000
PHASE 10 (218.54174, 337.50000) (186.07411, 415.88381) 17.000 25.000
PHASE 11 (186.07411, 415.88381) (175.00000, 500.00000) 17.000 25.000
PHASE 12 (175.00000, 500.00000) (186.07411, 584.11619) 17.000 25.000
PHASE 13 (186.07411, 584.11619) (218.54174, 662.50000) 17.000 25.000
PHASE 14 (218.54174, 662.50000) (270.19030, 729.80970) 17.000 25.000
PHASE 15 (270.19030, 729.80970) (337.50000, 781.45826) 17.000 25.000
PHASE 16 (337.50000, 781.45826) (415.88381, 813.92589) 17.000 25.000
PHASE 17 (415.88381, 813.92589) (500.00000, 825.00000) 17.000 25.000
PHASE 18 (500.00000, 825.00000) (584.11619, 813.92589) 17.000 25.000
PHASE 19 (584.11619, 813.92589) (662.50000, 781.45826) 17.000 25.000
PHASE 20 (662.50000, 781.45826) (729.80970, 729.80970) 17.000 25.000
PHASE 21 (729.80970, 729.80970) (781.45826, 662.50000) 17.000 25.000
PHASE 22 (781.45826, 662.50000) (813.92589, 584.11619) 17.000 25.000
PHASE 23 (813.92589, 584.11619) (825.00000, 500.00000) 17.000 25.000
LANE_BEGIN 0 6 6 -1 24
PHASE 0 (830.00000, 500.00000) (818.75552, 414.58972) 17.000 25.000
PHASE 1 (818.75552, 414.58972) (785.78838, 335.00000) 17.000 25.000
PHASE 2 (785.78838, 335.00000) (733.34524, 266.65476) 17.000 25.000
PHASE 3 (733.34524, 266.65476) (665.00000, 214.21162) 17.000 25.000
PHASE 4 (665.00000, 214.21162) (585.41028, 181.24448) 17.000 25.000
PHASE 5 (585.41028, 181.24448) (500.00000, 170.00000) 17.000 25.000
PHASE 6 (500.00000, 170.00000) (414.58972, 181.24448) 17.000 25.000
PHASE 7 (414.58972, 181.24448) (335.00000, 214.21162) 17.000 25.000
PHASE 8 (335.00000, 214.21162) (266.65476, 266.65476) 17.000 25.000
PHASE 9 (266.65476, 266.65476) (214.21162, 335.00000) 17.000 25.000
PHASE 10 (214.21162, 335.00000) (181.24448, 414.58972) 17.000 25.000
PHASE 11 (181.24448, 414.58972) (170.00000, 500.00000) 17.000 25.000
PHASE 12 (170.00000, 500.00000) (181.24448, 585.41028) 17.000 25.000
PHASE 13 (181.24448, 585.41028) (214.21162, 665.00000) 17.000 25.000
PHASE 14 (214.21162, 665.00000) (266.65476, 733.34524) 17.000 25.000
PHASE 15 (266.65476, 733.34524) (335.00000, 785.78838) 17.000 25.000
PHASE 16 (335.00000, 785.78838) (414.58972, 818.75552) 17.000 25.000
PHASE 17 (414.58972, 818.75552) (500.00000, 830.00000) 17.000 25.000
PHASE 18 (500.00000, 830.00000) (585.41028, 818.75552) 17.000 25.000
PHASE 19 (585.41028, 818.75552) (665.00000, 785.78838) 17.000 25.000
PHASE 20 (665.00000, 785.78838) (733.34524, 733.34524) 17.000 25.000
PHASE 21 (733.34524, 733.34524) (785.78838, 665.00000) 17.000 25.000
PHASE 22 (785.78838, 665.00000) (818.75552, 585.41028) 17.000 25.000
PHASE 23 (818.75552, 585.41028) (830.00000, 500.00000) 17.000 25.000
LANE_BEGIN 0 7 7 -1 24
PHASE 0 (835.00000, 500.00000) (823.58515, 413.29562) 17.000 25.000
PHASE 1 (823.58515, 413.29562) (790.11851, 332.50000) 17.000 25.000
PHASE 2 (790.11851, 332.50000) (736.88077, 263.11923) 17.000 25.000
PHASE 3 (736.88077, 263.11923) (667.50000, 209.88149) 17.000 25.000
PHASE 4 (667.50000, 209.88149) (586.70438, 176.41485) 17.000 25.000
PHASE 5 (586.70438, 176.41485) (500.00000, 165.00000) 17.000 25.000
PHASE 6 (500.00000, 165.00000) (413.29562, 176.41485) 17.000 25.000
PHASE 7 (413.29562, 176.41485) (332.50000, 209.88149) 17.000 25.000
PHASE 8 (332.50000, 209.88149) (263.11923, 263.11923) 17.000 25.000
PHASE 9 (263.11923, 263.11923) (209.88149, 332.50000) 17.000 25.000
PHASE 10 (209.88149, 332.50000) (176.41485, 413.29562) 17.000 25.000
PHASE 11 (176.41485, 413.29562) (165.00000, 500.00000) 17.000 25.000
```

```
PHASE 12 (165.00000, 500.00000) (176.41485, 586.70438) 17.000 25.000
PHASE 13 (176.41485, 586.70438) (209.88149, 667.50000) 17.000 25.000
PHASE 14 (209.88149, 667.50000) (263.11923, 736.88077) 17.000 25.000
PHASE 15 (263.11923, 736.88077) (332.50000, 790.11851) 17.000 25.000
PHASE 16 (332.50000, 790.11851) (413.29562, 823.58515) 17.000 25.000
PHASE 17 (413.29562, 823.58515) (500.00000, 835.00000) 17.000 25.000
PHASE 18 (500.00000, 835.00000) (586.70438, 823.58515) 17.000 25.000
PHASE 19 (586.70438, 823.58515) (667.50000, 790.11851) 17.000 25.000
PHASE 20 (667.50000, 790.11851) (736.88077, 736.88077) 17.000 25.000
PHASE 21 (736.88077, 736.88077) (790.11851, 667.50000) 17.000 25.000
PHASE 22 (790.11851, 667.50000) (823.58515, 586.70438) 17.000 25.000
PHASE 23 (823.58515, 586.70438) (835.00000, 500.00000) 17.000 25.000
```

APPENDIX C

Network Traffic

C.1 traffic-w-jitter.pl

```
#! /usr/bin/perl
#
#  traffic-w-jitter.pl generates broadcast traffic for an
#  ns2 simulation.  The script randomly selects a broadcast
#  time for each node in the simulation.  For each time
#  interval during which a node boardcasts, the transmit
#  time is varied by +/- the amount of jitter.  The
#  variables start and end are the start time and end time
#  of  the simulation.  traffic_rate is the number of
#  times that a nodes transmits during one second.
#
#  Nathan Balon
#  University of Michigan - Dearborn
#

use strict;
use warnings;

sub setValuesFromArgs($);
sub getCommentBlock();

# the default values used to generate traffic

our $jitter = 0.1;              # the amount of jitter
our $nodes = 50;                # number of node
our $traffic_rate = 10;         # number of times a node transmits per second
our $seed = 11;                 # seed for the random number generator
our $start = 1;                 # start time of the simulation
our $end = 61;                  # end time of the simulation
our $file_name = 'traffic.tcl'; # file to save traffic to
our $num_emer_messages = 15;    # the number of emergency messages broadcast
our $perc_emer = 0.015;          # percent of emergency

my %emer_broadcast_end;        # nodes to send an emergency broadcast
my $sim_length = 0;             # length of the simulation
my @trans_time = ();            # the transmit time

# read the command line arguments
for(my $argnum = 0; $argnum <= $#ARGV; $argnum++){
    # display help
```

```
    if($ARGV[$argnum] eq "-h"  || $ARGV[$argnum] eq "--help" ){
        printHelp();
        exit;
    # set the values used to generate traffic from the cammand line args
    }elsif($argnum + 1 <= $#ARGV){
        setValuesFromArgs($argnum);
        $argnum++;
    }
}

for(my $i = 0; $i < $nodes; $i++){
     $emer_broadcast_end{$i} = $end;
}

# determine the length of the simulation
if($end > $start){
    $sim_length = $end - $start;
}else{
    die "The end time of the simulation must be " .
         "greater than the start time of the simulation\n";
}

srand $seed;

#open the file and write the comment block to the file
open(FILE, ">$file_name");
print FILE getCommentBlock;

# the transmission rate to send messages
my $trans_rate  = sprintf("%.2f",  1/$traffic_rate);


# Select the initial time a node will
# transmit at based on the transmission rate.
for(my $i = 0; $i < $nodes; $i++){
     $trans_time[$i] = rand($trans_rate);
}

# the number of time each node sends a broadcast
my $num_intervals = $sim_length * $traffic_rate;

# the time interval to transmit during
my $transmit_interval = 0;

# Schedule the broadcast transmissions for the simulation.
for(my $trans_int = 0; $trans_int < $num_intervals; $trans_int++){
    # for each node in the simulation schedule a broadcast
    for(my $node = 0; $node < $nodes; $node++){
        my $trans_time = 0;  # the time the broadcast is sent
        if($trans_int == 0){
            # if first transmission use the
            # randomly selected transmit time
            $trans_time = $start + $trans_time[$node];
        }else{
            # generate emergency warnings
            if(rand(1) <= $perc_emer){
                 # set the time to stop sending emergency warnings by a node
                 $emer_broadcast_end{$node} = $start + $transmit_interval +
                                             $trans_rate * $num_emer_messages +
```

```
                                              $jitter * $trans_rate ;
                my $time_now = $start + $transmit_interval;
 #               print "$node time now = $time_now\n";
 #               print "$node end time: $emer_broadcast_end{$node}\n";
            }
            # get the random amount of jitter for the broadcast
            my $jit = rand($jitter * $trans_rate);
            # if random number is > 0.5 add the jitter or else
            # subtract the jitter
            if(rand(1) > 0.5){
                $trans_time = $start + $trans_time[$node]
                                     + $jit + $transmit_interval;
            }else{
                $trans_time = $start + $trans_time[$node]
                                     - $jit + $transmit_interval;
            }
        }
        # at the expiration of the emergency broadcast set the node
        # so it will no long transmit emergency messages
        if($emer_broadcast_end{$node} < $start + $transmit_interval){
            $emer_broadcast_end{$node} = $end;
#            print "$node $end $trans_time\n";
        }

        if($trans_time <= $emer_broadcast_end{$node} &&
                                        $emer_broadcast_end{$node} != $end){
               print FILE "\$ns at $trans_time \"\$em_agent($node) send_message\"\n";
        } else {
               print FILE "\$ns at $trans_time \"\$bc_agent($node) send_message\"\n";
        }
    }
    $transmit_interval += $trans_rate;
}

close(FILE);

##################################################################
#
#   functions
#
##################################################################

# display help containing the command line arguments to the user
sub printHelp {
    my $space = " " x (length($0) + 6) ;
    print "usage $0 [--nodes \"number of nodes\"]\n" .
          "$space [--jitter \"amount of jitter\"]\n" .
          "$space [--traffic-rate \"rate of traffic per second\"]\n" .
          "$space [--seed \"seed for random number generator\"]\n".
          "$space [--emer-duration \"number of emergency messages\"]\n" .
          "$space [--start \"start time\"]\n" .
          "$space [--end \"end time\"]\n".
          "$space [--file-name \"name of the file to write to\"]\n";
}

# Return a comment block for the tcl traffic file
sub getCommentBlock() {
    # return the comment block to add to the tcl program.
    return "#=========================================================================\n" .
```

```
           "#\n" .
           "#  Broadcast Traffic genrated by generate_traffic tool.\n" .
           "#  Each node within the network randomly selects the time a time to\n" .
           "#  generate broadcast message during each broadcast interval.\n" .
           "#\n" .
           "#  Created by Nathan Balon, University of Michigan - Dearborn\n" .
           "#\n" .
           "#  start time : $start, end time : $end\n" .
           "#  broadcast rate: $traffic_rate\n" .
           "#\n" .
           "#========================================================================\n\n";
}

# Set the parameters used by the program from the
# command line arguments.
sub setValuesFromArgs($) {
    my $argnum = shift;
    my $arg = $ARGV[$argnum + 1];
    if($ARGV[$argnum] eq "--jitter"){
         $jitter = $arg;
    }elsif($ARGV[$argnum] eq "--nodes"){
         $nodes = $arg;
    }elsif($ARGV[$argnum] eq "--traffic-rate"){
         $traffic_rate = $arg;
    }elsif($ARGV[$argnum] eq "--seed"){
         $seed = $arg;
    }elsif($ARGV[$argnum] eq "--start"){
         $start = $arg;
    }elsif($ARGV[$argnum] eq "--end"){
         $end = $arg;
    }elsif($ARGV[$argnum] eq "--file-name"){
         $file_name = $arg;
    }elsif($ARGV[$argnum] eq "--emer-duration"){
         $num_emer_messages = $arg;
    }else{
        die "invalid command line argument";
    }
}
```

APPENDIX D

Performace Metrics

D.1 reception.pl

```
#! /usr/bin/perl
#
# reception.pl
# Calculates the reception rate at various distances
#
#
#
# Nathan Balon
# University of Michigan - Dearborn
#


use strict;
use warnings;

# compare the numbers
sub numerically { $a <=> $b }

# set the program variables from the command
# line arguments.
sub setValuesFromArgs($);
sub displayHelp;

# files to read and write to
our $trace = "trace.tr";   # ns trace file
our $output;               # file to write results to

# default values
our $range = 300;          # transmit radius
our $scale_dist = 10;      # scale of distance
my $number_nodes = 0;      # number of nodes in the simulation

my $location = ();         # the location of the nodes

my $dropped_at_dist  = ();     # dropped packets at a specific distance
my $received_at_dist = ();     # number received packets at a specific distance

our $packets_sent_neighbors = ();  # nodes within range to receive a packet
our $packets_sent_time = ();       # time the packet was sent
our $packets_sent_port = ();
```

```
our $packets_sensed_at_recv = ();  # list of nodes that sensed a packet
our $packets_not_sensed = ();      # list of nodes that did not sense a packet

our $time_interval    = 0.2;   # time interval to remove entries
my  $prev_update_time = 0.0;   # last time old entries were removed

# Simulation Totals
my %total_sent;       # total sent
my %total_received;   # total received
my %total_dropped;    # total dropped
my %total_not_sensed; # total packets not sensed at the receiver

# get the command line arguments
for(my $arg = 0; $arg <= $#ARGV; $arg++){
    if($ARGV[$arg] eq "-h" || $ARGV[$arg] eq "--help"  ){
        displayHelp;
        exit;
    }elsif($arg + 1 <= $#ARGV){
        setValuesFromArgs($arg);
        $arg++;
    }
}

# open the ns-2 trace file
open(TRACE, "<$trace") or
    die "Could not open trace: " . $trace . " \n";



# read through the trace file
while(my $input_line = <TRACE>){
    # read a line from the trace
    chomp $input_line;
    my @line = split(/\s+/, $input_line);
    my $time = $line[1];


    if($time > ($prev_update_time + $time_interval)){
        remove_old_entries($time);
        $prev_update_time = $time;
    }

    # if not a mobility line of the trace
    if($line[0] ne "M" && $line[3] eq "MAC"){
        # get the address in the correct format
        # remove the leading and trailing '_' from the address
        $line[2] =~ s/_//g;

        if($line[0] eq "r" || $line[0] eq "D"){
            my $recv_addr = $line[2];
            # remove the '[' and ':' from the address of
            # the node that transmitted the packet
            my @transmitter = split(/:/, $line[13]);
            $transmitter[0] =~ s/\[//;
            my $port = $transmitter[1];
#            print "$port\n";
            # get the address of the transmitter
            my $trans_addr = $transmitter[0];
            # get the sequence number of the packet
```

```
        my $seq_num = $line[5];

        # get the distance between the transmitter and receiver
        my $dist = distance($location->{$trans_addr}->{'x'},
                            $location->{$recv_addr}->{'x'},
                            $location->{$trans_addr}->{'y'},
                            $location->{$recv_addr}->{'y'});

        # if the packet was dropped add one to the count
        if($line[0] eq "D" && $recv_addr != $trans_addr){
#            print "dropped $port distance $dist\n";
            # add to the total of packets dropped at the distance
            $dropped_at_dist->{$port}->{$dist} += 1;
            # add the node to the array of nodes that sensed the packet
            push @{$packets_sensed_at_recv->{$seq_num}}, $recv_addr;
        # if the packet was received add one to the count
        }elsif($line[0] eq "r" && $recv_addr != $trans_addr){
            # add to the total of packets received at the distance
            $received_at_dist->{$port}->{$dist} += 1;
            # add the node to the array of nodes that sensed the packet
            push @{$packets_sensed_at_recv->{$seq_num}}, $recv_addr;
        }
    # if the packet was sent
    }elsif($line[0] eq "s"){
        my $sender_addr = $line[2];
        my $seq = $line[5];
        # get the transmitting nodes location
        my $x_sender = $location->{$sender_addr}->{'x'};
        my $y_sender = $location->{$sender_addr}->{'y'};

        # get the port number
        my @transmitter = split(/:/, $line[13]);
        my $port = $transmitter[1];

        #increment the packets sent
        $packets_sent_port->{$seq} = $port;
        $total_sent{$port}++;

        # determine which nodes are within the transmission range
        foreach my $neighbor_node_addr (keys %{$location}){
            if($sender_addr != $neighbor_node_addr){
                # determine the distance between nodes
                my $distance = distance($x_sender,
                      $location->{$neighbor_node_addr}->{'x'},
                      $y_sender,
                      $location->{$neighbor_node_addr}->{'y'});

                # if within the node is within the transmission range
                # add the destination to the hash
                if($distance <= $range){
                     # add the address and distance of the neighboring
                     # node to the hash
                     $packets_sent_neighbors->{$seq}->{$neighbor_node_addr} =
                                                  $distance;
                     $packets_sent_time->{$seq} = $time;
                }
            }
        }
    }
```

```
    }

    # if a mobility line update the coordinates of the node
    # once the simulation starts
    elsif($line[0] eq "M" && $line[1] > 0){
        # get the coordinates in the correct format
        $line[3] =~ s/[,\(]//g;
        $line[4] =~ s/,//g;
        # update the x and y coordinates of a mobile node
        $location->{$line[2]}->{'x'} = $line[3];
        $location->{$line[2]}->{'y'} = $line[4];
    }elsif($line[0] eq "M" && $line[2] eq "nn"){
        # set the number of nodes in the simulation
        $number_nodes = $line[3];
    }
}

# print table heading
print "\nNumber of Packets Received and Dropped for $number_nodes Nodes\n";
print "-" x 75; print "\n";

#display the packets received
foreach my $port (sort numerically keys %{$received_at_dist}){
    print "port $port\n";
    print "----------------------------------------------"  .
          "-------------------------------------------\n";
    foreach my $dist (sort numerically keys %{$received_at_dist->{$port}}){
        my $dropped_packets = 0;
        my $not_sensed = 0;

        $total_received{$port} += $received_at_dist->{$port}->{$dist};

        if(defined $dropped_at_dist->{$port}->{$dist}){
            $dropped_packets = $dropped_at_dist->{$port}->{$dist};
            $total_dropped{$port} += $dropped_at_dist->{$port}->{$dist};
        }
        if(defined $packets_not_sensed->{$port}->{$dist}){
            $not_sensed = $packets_not_sensed->{$port}->{$dist};
            $total_not_sensed{$port} += $packets_not_sensed->{$port}->{$dist};
        }
        my $percent_received = ($received_at_dist->{$port}->{$dist} /
                               ($received_at_dist->{$port}->{$dist} +
                                $dropped_packets + $not_sensed)) *100;

        # display the results of the simulation for the transmission range
        printf("dist: %3d - %3d m, recv: %8d, dropped: %7d, not sensed: %7d,
                percent recv: %3.3f \n", $dist * $scale_dist,
                $dist * $scale_dist + $scale_dist - 1 ,
                $received_at_dist->{$port}->{$dist},
                $dropped_packets, $not_sensed, $percent_received);
        # write the results to a file
        if(defined $output){
            open(OUTPUT, ">>$output-$port.txt") or die "unable to open: $output\n";
            print OUTPUT $dist * $scale_dist . " " . $percent_received . "\n";
#            print(OUTPUT $dist * $scale_dist . " " . $received_at_dist->{$port}->{$dist} .
#                  " " . $dropped_packets . " " . $not_sensed .
#                  " " . $percent_received . "\n");
        }
    }
```

```
}


# display the overall totals for the simulation
print "\nSimulation Totals\n";
print "------------------------------------------" .
      "------------------------------------------\n";
foreach my $port (21, 42){
    my $total_recp_rate = 0;
    if($total_received{$port} + $total_dropped{$port} + $total_not_sensed{$port} > 0){
        $total_recp_rate = ($total_received{$port}/($total_received{$port} +
        $total_dropped{$port} + $total_not_sensed{$port})) * 100;
    }
    print "sent $total_sent{$port} received $total_received{$port}, " .
          "dropped $total_dropped{$port}, " .
          "not sensed $total_not_sensed{$port}, rate $total_recp_rate\n";
}

###########################################################################
#
#  Functions
#
############################################################################

# determine the distance between the nodes.
sub distance {
    my $xA = shift;    my $xB = shift;
    my $yA = shift;    my $yB = shift;
    # return the distance between nodes
    my $dist = sqrt((($xA - $xB) ** 2) + (($yA - $yB) ** 2));
    return $dist = int($dist/$scale_dist);
}

# remove entries after the time_interval has expired
# and determine the number of packets not sensed.
sub remove_old_entries {
    my $time = shift;    # the current time

    foreach my $seq_num (keys %{$packets_sent_time}){
        my $port = $packets_sent_port->{$seq_num};
        if($packets_sent_time->{$seq_num} <= ($time - $time_interval)){
            if(defined $packets_sensed_at_recv->{$seq_num}){
                # for each packet that was received, delete the coresponding
                # entry in the hash contain the packets sent
                foreach my $address (@{$packets_sensed_at_recv->{$seq_num}}){
                     delete $packets_sent_neighbors->{$seq_num}->{$address};
                }
                # the remain entries in the hash contains
                # that were not sensed
                foreach my $addr (keys %{$packets_sent_neighbors->{$seq_num}}){
                    my $dist = $packets_sent_neighbors->{$seq_num}->{$addr};
                    $packets_not_sensed->{$port}->{$dist}++;
                    # print "not received: $addr dist: ". $scale_dist * $dist . "\n";
                }
                delete $packets_sensed_at_recv->{$seq_num};
            }else{
                foreach my $address (keys %{$packets_sent_neighbors->{$seq_num}}){
                    my $dist = $packets_sent_neighbors->{$seq_num}->{$address};
                    $packets_not_sensed->{$port}->{$dist}++;
```

```
                    delete $packets_sent_neighbors->{$seq_num}->{$address};
                }
            }
            # delete the entries that are no longer needed
            delete $packets_sent_port->{$seq_num};
            delete $packets_sent_neighbors->{$seq_num};
            delete $packets_sent_time->{$seq_num};
        }
    }
}

# display help
sub displayHelp {
   my $space = " " x (length($0) + 7) ;
   print "usgage $0 [--trace \"trace file\"]\n" .
         "$space [--output \"output file\"]\n" .
         "$space [--range \"transmit radius\"]\n";
}

# Set the parameters used by the program from the
# command line arguments.
sub setValuesFromArgs($) {
    my $arg = shift;
    if($ARGV[$arg] eq "--trace"){
        $trace = $ARGV[$arg + 1];
    }elsif($ARGV[$arg] eq "--output"){
        $output = $ARGV[$arg + 1];
    }elsif($ARGV[$arg] eq "--range"){
        $range = $ARGV[$arg +1];
    }else{
        die "invalid argument";
    }
}
```

D.2 access-time.pl

```
#! /usr/bin/perl
#
#  access_time.pl
#  Access Time - measure the amount of time from when
#  the packet is given to the MAC layer until it is
#  put on the channel.  The script gives an average
#  of the amount of time in the simulation time a
#  packet is placed on the channel.
#
#  Nathan Balon
#  University of Michigan - Dearborn
#

use strict;
use warnings;

sub setValuesFromArgs($);

our $MESSAGE = 42;
our $EMERGENCY = 21;
```

```
# files to read and write to
our $trace = "trace.tr";  # ns trace file
our $output;              # file to write results to


our %total_time;
our %packets_sent;
our %packets_passed;
our %dropped;
our $time_passed_to_mac = ();    # packets passed to the mac layer

my  $number_nodes = 0;           # number of nodes in the simulation

# get the command line arguments
for(my $arg = 0; $arg <= $#ARGV; $arg++){
    if($ARGV[$arg] eq "-h" || $ARGV[$arg] eq "--help"){
        displayHelp();
        exit;
    }elsif($arg + 1 <= $#ARGV){
        setValuesFromArgs($arg);
        $arg++;
    }else{
        die "invalid argument\n";
    }
}

# open the ns-2 trace file
open(TRACE, "<$trace") or
    die "Could not open trace: " . $trace . " \n";

# read through the file for the sent packets
while(my $input_line = <TRACE>){
    chomp $input_line;
    my @line = split(/\s+/, $input_line);

    # get the fields of use from the trace
    my $type = $line[0];
    my $time = $line[1];
    my $layer = $line[3];
    my $seq_num = $line[5];

    # if the packet is sent, set the time it was sent
    if($type eq "s" && $layer eq "AGT" && ($line[13] =~ /:$MESSAGE$/)){
        # store the time the packet was passed to the MAC
        $time_passed_to_mac->{$seq_num} = $time;
        $packets_passed{$MESSAGE}++;
    }elsif($type eq "s" && $layer eq "MAC" && ($line[13] =~ /:$MESSAGE$/)){
        # Add the time the packet was held by the MAC
        # to the totals of the simulation.
        add_to_totals($seq_num, $time, $MESSAGE);
    }elsif($type eq "s" && $layer eq "AGT" && ($line[13] =~ /:$EMERGENCY$/)){
        # store the time the packet was passed to the MAC
        $time_passed_to_mac->{$seq_num} = $time;
        $packets_passed{$EMERGENCY}++;
    }elsif($type eq "s" && $layer eq "MAC" && ($line[13] =~ /:$EMERGENCY$/)){
        # Add the time the packet was held by the MAC
        # to the totals of the simulation.
        add_to_totals($seq_num, $time, $EMERGENCY);
    }
```

```
    elsif($line[0] eq "M" && $line[2] eq "nn"){
        #number of nodes in sim
        $number_nodes = $line[3];
    }
}

close TRACE;
#print "message packet: $message_packet_count\n";
#print "emr packet: $emergency_packet_count\n";

# get the number of packets not passed to PHY
foreach my $packet_type ($MESSAGE, $EMERGENCY){
    $dropped{$packet_type} = $packets_passed{$packet_type} - $packets_sent{$packet_type};
}


# display the totals
foreach my $packet_type ($MESSAGE, $EMERGENCY){
    my $avg_access_time = 0;
    if($packets_sent{$packet_type} != 0){
        $avg_access_time = $total_time{$packet_type}/$packets_sent{$packet_type};
    }
    print "packet type = ";
    if($packet_type == 21){
        print "EMERGENCY\n"
    } else {
        print "MESSAGE\n";
    }
    print "number nodes    = $number_nodes\n";
    print "avg access time = $avg_access_time\n";
    print "dropped at MAC  = $dropped{$packet_type}\n";
    print "sent packets    = $packets_sent{$packet_type}\n";
    print "\n";

    # if output is define write the results to a file
    if(defined $output){
        open(OUT, ">>$output-$packet_type.txt");
        print OUT "$number_nodes $avg_access_time\n";
        close OUT;
    }


}

# For a packet passed to PHY add the time it was held by the MAC
# to total time and increment the number of packets sent.
sub add_to_totals {
    my $seq_num = shift;  # sequence number of the packet
    my $time = shift;     # time taken to put the packet on PHY
    my $port = shift;

    if($port == $MESSAGE){
        $total_time{$MESSAGE} += $time - $time_passed_to_mac->{$seq_num};
        $packets_sent{$MESSAGE}++;
    }elsif($port == $EMERGENCY){
        $total_time{$EMERGENCY} += $time - $time_passed_to_mac->{$seq_num};
        $packets_sent{$EMERGENCY}++;
```

```
    }
    delete($time_passed_to_mac->{$seq_num});
}

# display help
sub dispalyHelp {
   my $space = " " x (length($0) + 7) ;
   print "usgage $0 [--trace \"trace file\"]\n" .
         "$space [--output \"output file\"]\n";
}

# Set the parameters used by the program from the
# command line arguments.
sub setValuesFromArgs($) {
    my $arg = shift;
    if($ARGV[$arg] eq "--trace"){
        $trace = $ARGV[$arg + 1];
    }elsif($ARGV[$arg] eq "--output"){
        $output = $ARGV[$arg + 1];
    }else{
        die "invalid argument";
    }
}
```

APPENDIX E

Modifications to the Simulator

E.1 Broadcast Table

E.1.1 broadcast_table.h

```
#ifndef ns_broadcast_table
#define ns_broadcast_table

#include <stdio.h>
//#include "config.h"
typedef unsigned int u_int32_t;

const unsigned int PRIME_TABLE_SIZE = 8;  // Size of the table of prime values
const double DEFAULT_ALPHA = 0.80;        // alpha to calculate the reception rate
const double TIMEOUT = 1.00;              // The time period used to remove entry if a
                                          // packet is not received for that time period.

/*
 *  A table entry to record the overheard broadcast messages coming from
 *  a mac address.  The entry records the sequence number and time of the last
 *  heard broadcast coming from a mac address.  Each entry contains a bitmap
 *  to record the sequence numbers and determine lost messages.  The table
 *  entries will be added to a hash table so that an entry can be retreived
 *  in near constant time.  The hash table uses chaining so an pointer in
 *  contained to to link entries that hash to the same value.
 */

class broadcast_entry {
    friend class broadcast_table; // Allow broadcast_table to access the private members.

  public:

     broadcast_entry();
     broadcast_entry(u_int32_t ta, u_int32_t lsn, double time);
     broadcast_entry(const broadcast_entry & entry);
     broadcast_entry & operator=(const broadcast_entry & entry);
    ~broadcast_entry();

    void print();               // display an entry

  protected:

     broadcast_entry * next_entry();                // get the next entry in the link list
    broadcast_entry *remove(double timeout_limit);  // remove an entry,
```

```
                                                // if the timelimit is exceeded
    broadcast_entry *find(u_int32_t mac);          // find the entry for the mac address
    void add(broadcast_entry * entry);             // add a new node to the link list
    void update(u_int32_t seq, double time, double alpha);    // update the entry
    void print_list(int bucket_num);               // print all nodes in the linked list
    broadcast_entry *remove_tail();                // remove last entry in the link list.

    /* Data Members */
    u_int32_t mac_address_;     // Ethernet address of the node that sent the broadcast.
    u_int32_t sequence_number_; // The sequence number received.
    double avg_recp_rate_;      // the avg from the last update
    double time_last_heard_;    // The time a broadcast was last heard from the node.
    broadcast_entry *next_;     // The next entry
};


/*
 *  broadcast_table records the overheard broadcast messages coming from other
 *  nodes.  A hash table used to store the store the information for the
 *  overheard broadcast messages.  The broadcast_table contains methods to
 *  add, remove, and update an entry in the hash table.
 */
class broadcast_table {

  public:

    broadcast_table(int size, double alpha =
                    DEFAULT_ALPHA, double timeout = TIMEOUT);
     broadcast_table(double alpha =
                     DEFAULT_ALPHA, double timeout = TIMEOUT);
    ~broadcast_table();

    /* Public Member Functions */
    double avg_reception_rate(double current_time);     // determine the avg reception rate
    void update(u_int32_t address, u_int32_t seq, double time); // update a node
    u_int32_t size();                                    // the size of the hash table
    void print();                                        // display the hash table

  protected:
    int resize();                                        // resize the table.
    u_int32_t next_size();                               // the new next larger size
    u_int32_t previous_size();                           // the new next smaller size
    u_int32_t hash(u_int32_t mac_address);               // compute hash value
    u_int32_t hash(u_int32_t mac, u_int32_t number_buckets);    // compute hash value
    u_int32_t buckets_used();                            // get the number of buckets used
    void invalidate_old_entries(double current_time);   // remove the old entries
    void add(broadcast_entry * new_node);                // add a new node

    /*   Data Members */
    static u_int32_t primes[];     // prime number table for the hash function
    broadcast_entry **table_;      // the broadcast table
    u_int32_t table_size_;         // The size of the broadcast table.
    double timeout_;               // timeout value to invalidate table entries
    double alpha_;                 // alpha is used to estimate the avg collision rate.
    // the value of alpha is used to determine how quickly
    // the estimate will react to changing conditions.
};
```

```
#endif
```

E.1.2 broadcast_table.cc

```
/*
 *
 *  broadcast_table.cc stores entries in a hash table for nodes
 *  in the network that the node has overheard messages from.
 *
 *  Nathan Balon
 *  University of Michigan - Dearborn
 */


#include "broadcast_table.h"
#include <assert.h>
#include <stdlib.h>

const double INCREASE_SIZE_THREASHOLD = 0.5;    // increase the size of the hash table
const double DECREASE_SIZE_THREASHOLD = 0.1;    // decrease the size of the hash table

/* ============================================================
 *  Broadcast entry contains the information needed to
 *  record the broadcast messages coming from other node.
 *=============================================================*/


/*
 *  Contructors and Destructors
 */
broadcast_entry::broadcast_entry():mac_address_(0),
sequence_number_(0), avg_recp_rate_(0),
time_last_heard_(0), next_(NULL)
{
}


broadcast_entry::broadcast_entry(u_int32_t src_addr,
                                 u_int32_t seq, double time)
{
    mac_address_ = src_addr;
    sequence_number_ = seq;
    time_last_heard_ = time;
    next_ = NULL;
    avg_recp_rate_ = 1;
}

broadcast_entry::broadcast_entry(const broadcast_entry & entry)
{
    this->mac_address_ = entry.mac_address_;
    this->sequence_number_ = entry.sequence_number_;
    this->time_last_heard_ = entry.time_last_heard_;
    this->next_ = entry.next_;
    this->avg_recp_rate_ = entry.avg_recp_rate_;
}

broadcast_entry & broadcast_entry::operator=(const broadcast_entry & entry)
{
    this->mac_address_ = entry.mac_address_;
```

```
    this->sequence_number_ = entry.sequence_number_;
    this->time_last_heard_ = entry.time_last_heard_;
    this->next_ = entry.next_;
    this->avg_recp_rate_ = entry.avg_recp_rate_;
    return *this;
}

broadcast_entry::~broadcast_entry()
{
    mac_address_ = 0;
    sequence_number_ = 0;
    time_last_heard_ = 0;
    next_ = NULL;
}

/*
 *  Add an entry to the end of the linked list.
 */
void broadcast_entry::add(broadcast_entry * new_entry)
{
    if (this == NULL) {
        *this = *new_entry;
        return;
    }
    for (broadcast_entry * ent = this; ent != NULL; ent = ent->next_) {
        // If an entry for the mac address already exists do nothing.
        if (ent->mac_address_ == new_entry->mac_address_) {
//          printf("mac address %u already exists\n", ent->mac_address_);
            return;
        } else if (ent->next_ == NULL) {
            // Add the entry to the end of the linked list.
            ent->next_ = new_entry;
            return;
        }
    }
//    fprintf(stderr, "shouldn't reach this point in add\n");
}

/*
 *  Return the entry for the mac address.
 *  If not found the function will return NULL.
 */
broadcast_entry *broadcast_entry::find(u_int32_t mac_addr)
{
    if (this != NULL) {
        for (broadcast_entry * ent = this; ent != NULL; ent = ent->next_) {
            if (ent->mac_address_ == mac_addr) {
                return ent;
            }
            // if the end of the list is found in the list return NULL
            if (ent->next_ == NULL) {
                return NULL;
            }
        }
//      fprintf(stderr, "shouldn't reach this point in find\n");
    }
    return NULL;
}
```

```
/*
 *  Removes all of the entries in the link list that have
 *  reached the timeout limt.
 *  Returns the new head of the list.
 */
broadcast_entry *broadcast_entry::remove(double timeout_limit)
{
    broadcast_entry *previous = this;   // the previous node in the linked list
    broadcast_entry *head = this;       // the head of the linked list
    broadcast_entry *node_to_delete = NULL;     // node to delete
    bool delete_last_node = false;      // delete the node from the last pass

    for (broadcast_entry * ent = this; ent != NULL; ent = ent->next_) {
        // If node was found last time to delete, delete it.
        // The node is saved for the next round so the for loop
        // will continue.
        if (delete_last_node) {
            delete node_to_delete;
            delete_last_node = false;
        }
        // the node has not been updated for the timeout limit
        if (ent->time_last_heard_ < timeout_limit) {
            // delete the head of the list with more linked nodes
            if (head == ent && ent->next_ != NULL) {
                head = ent->next_;
                previous = ent->next_;
                node_to_delete = ent;
                delete_last_node = true;
                // delete the entry from the middle of the list
            } else if (head != ent && ent->next_ != NULL) {
                previous->next_ = ent->next_;
                node_to_delete = ent;
                delete_last_node = true;
                // if last node in the linked list delete it
            } else if (head != ent && ent->next_ == NULL) {
                previous->next_ = NULL;
                delete ent;
                return head;
                // if only the head node exists delete it
            } else if (head == ent && ent->next_ == NULL) {
                head = NULL;
                delete ent;
                return head;
            }
        } else {
            previous = ent;
        }
    }
    return (broadcast_entry *) head;
}



/*
 *  Update the information stored and calculate the new
 *  estimated reception rate.
 */
void broadcast_entry::update(u_int32_t seq, double time, double alpha)
{
```

```
    int diff_seq = seq - sequence_number_;
    assert(diff_seq != 0);
    // update the reception rate
    if (diff_seq > 1) {
        for (int i = 0; i < diff_seq - 1; i++) {
            avg_recp_rate_ = alpha * avg_recp_rate_ + (1.0 - alpha) * 0;
        }
        avg_recp_rate_ = alpha * avg_recp_rate_ + (1.0 - alpha) * (1.0);
    } else if (diff_seq == 1) {
        avg_recp_rate_ =
            alpha * avg_recp_rate_ + (1.0 - alpha) * (1.0 / diff_seq);
    }
//      printf("avg_recp_rate = %f\n", avg_recp_rate_);
    // update the sequence number and time stamp
    sequence_number_ = seq;
    time_last_heard_ = time;
}

/*
 *  remove_tail removes the last entry in the linked list.
 *  The function is used by the broadcast_table in the
 *  resize method, to copy table entries to a resized table.
 *  Returns the tail item in the linked list or returns
 *  NULL to indicate no more items in the list.
 */

broadcast_entry *broadcast_entry::remove_tail()
{
    broadcast_entry *previous_ent = this; // store the previous node in the linked list.
    bool first_pass = true;     // During the first pass the previous_ent is not update.

    for (broadcast_entry * ent = this; ent != NULL; ent = ent->next_) {
        if (ent->next_ != NULL) {
            // do nothing there are more nodes in the list
        } else if (!first_pass && ent->next_ == NULL) {
            // if at the end of the list set previous entries
            // next pointer to NULL.
            previous_ent->next_ = NULL;
            return ent;

        } else if (first_pass && ent->next_ == NULL) {
            // if only entry return NULL
            return NULL;
        } else {
//          fprintf(stderr, "Error shouldn't reach here\n");
            return NULL;
        }
        // set the previous node in the link list
        if (!first_pass) {
            previous_ent = ent;
        }
        first_pass = false;     // set first pass false to indicate
                                // the previous node must be updated
    }
    return NULL;
}


/*
```

```
 *  Get the next entry in the linked list.
 */
broadcast_entry *broadcast_entry::next_entry()
{
    return this->next_;
}


/*
 *  Print out all nodes contained in the linked list.
 */
void broadcast_entry::print_list(int bucket_num)
{
    printf("Bucket %4d ", bucket_num);
    bool first_run = true;
    for (broadcast_entry * entry = this; entry != NULL;) {
        if (first_run) {
            entry->print();
            first_run = false;
        } else {
            printf("            ");
            entry->print();
        }
        entry = entry->next_;
    }
}

/*
 *  Display the information contained in a broadcast entry.
 */
void broadcast_entry::print()
{
    printf
        ("mac: %6u, sequence #: %6u, time: %f, recp_rate %f, next: %p\n",
         mac_address_, sequence_number_, time_last_heard_,
         avg_recp_rate_, next_);
}

/*===============================================================================
 *  Broadcast table is a hash table used to record the overheard messages
 *  coming from neighboring nodes.
 * ===============================================================================*/

/*
 *  Constructors and Destructor
 */
broadcast_table::broadcast_table(int size, double alpha, double timeout)
{
    table_ = new broadcast_entry *[size];
    for (int i = 0; i < size; i++) {
        table_[i] = NULL;
    }
    table_size_ = size;
    alpha_ = alpha;
    timeout_ = timeout;
}


broadcast_table::broadcast_table(double alpha, double timeout)
```

```
{
    table_ = new broadcast_entry *[primes[0]];
    table_size_ = primes[0];
    for (unsigned int i = 0; i < primes[0]; i++) {
        table_[i] = NULL;
    }
    alpha_ = alpha;
    timeout_ = timeout;
}

/*
 *  ~broadcast_table destructor deletes all of the entries in the
 *   table and then deletes the table.
 */
broadcast_table::~broadcast_table()
{
    for (unsigned int i = 0; i < table_size_; i++) {
        // if the bucket contains data
        if (table_[i] != NULL) {
            // delete all entries in the linked list
            while (table_[i]->next_ != NULL) {
                broadcast_entry *next_ent = table_[i]->next_;
                broadcast_entry *ent_to_delete = table_[i];
                table_[i] = next_ent;
                delete ent_to_delete;
            }
            // delete the head of the list
            delete table_[i];
        }
    }
    // delete the table
    delete[]table_;
    table_size_ = 0;
}

/*
 *  Hash value for the mac address used to select a bucket to
 *  insert the entry into
 */
u_int32_t broadcast_table::hash(u_int32_t address)
{
    return (address % table_size_);
}

u_int32_t broadcast_table::hash(u_int32_t address,
                                u_int32_t number_buckets)
{
    return (address % number_buckets);
}

/*
 *  add_node adds a new node to the table
 */
void broadcast_table::add(broadcast_entry * new_node)
{
    // check if the table has to be resized
    int used_buckets = buckets_used();
    double percent_full = (double) used_buckets / table_size_;
//      printf("percent full %f\n", percent_full);
```

```
    if (percent_full > INCREASE_SIZE_THREASHOLD) {
        resize();
    }
    // get the hash key
    u_int32_t key = hash(new_node->mac_address_);
    if (table_[key] == NULL) {
        // if no collision add the node
        table_[key] = new_node;
    } else {
        // If a collision, add the entry at the end of
        // the link list for the bucket
        table_[key]->add(new_node);
    }
}

/*
 *  print displays all of the entries contained in the hash table.
 */
void broadcast_table::print()
{
    printf("\nbroadcast table size: %d, buckets used: %d\n",
           table_size_, buckets_used());
    for (unsigned int i = 0; i < table_size_; i++) {
        if (table_[i] == 0) {
            printf("Bucket %4d empty\n", i);
        } else {
            table_[i]->print_list(i);
        }
    }
}

/*
 *  Update the table entry for the node of a given address.
 */
void broadcast_table::update(u_int32_t address,
                             u_int32_t new_seq, double new_time)
{
    int bucket = hash(address); // the bucket which would contain the mac address

    if (broadcast_entry * ent = table_[bucket]->find(address)) {
        // If a node already exist for the mac address update the information.
        ent->update(new_seq, new_time, alpha_);
    } else {
        // If the node was not found in the table add it to the table.
        // check if the table has to be resized
        int used_buckets = buckets_used();
        double percent_full = (double) used_buckets / table_size_;
        if (percent_full > INCREASE_SIZE_THREASHOLD) {
            // the treshold is reached resize the table
            resize();
            bucket = hash(address);     // get new hash
        }
        // add the new node to the table
        if (table_[bucket] == NULL) {
            // add entry to the bucket
            table_[bucket] =
                new broadcast_entry(address, new_seq, new_time);
        } else {
            // add the entry to the end of the link list
```

```
            table_[bucket]->
                add(new broadcast_entry(address, new_seq, new_time));
        }
    }
}


/*
 *  size: returns the size of the hash table.
 */
u_int32_t broadcast_table::size()
{
    return table_size_;
}

/*
 *  next_size: returns the next table size used to increase
 *  the size of the table.
 */

u_int32_t broadcast_table::next_size()
{
    for (unsigned int i = 0; i < PRIME_TABLE_SIZE; i++) {
        if ((primes[i] == table_size_)
            && (i + 1 <= PRIME_TABLE_SIZE)) {
            return (primes[i + 1]);
        }
    }
    // shoudln't reach here
//    fprintf(stderr, "ERROR: couldn't determine the size of the new table");
    return 0;
}

/*
 *  previous_size: returns the next smallest size used to resize the table.
 */
u_int32_t broadcast_table::previous_size()
{
    for (unsigned int i = 0; i < PRIME_TABLE_SIZE; i++) {
        if (primes[i] == table_size_ && i - 1 >= 0) {
            return (primes[i - 1]);
        }
    }
    // if already using the smallest table
    // return the smallest table size.
    return primes[0];
}


/*
 *  buckets_used: calculates the number of buckets used in the
 *  hash table.
 */
u_int32_t broadcast_table::buckets_used()
{
    int buckets_full = 0;
    for (unsigned int i = 0; i < table_size_; i++) {
        if (table_[i] != NULL) {
            buckets_full++;
```

```
        }
    }
    return buckets_full;
}


/*
 *  resize: resizes the hash table.
 *  returns 1 if the size of the table is increased.
 *  returns -1 if the size of the table is decreased.
 *  returns 0 if the size of the table maintains the same.
 */
int broadcast_table::resize()
{
    broadcast_entry *removed_entry = NULL;
    double percent_full = ((double) buckets_used() / (double) table_size_);
//    this->print();
    // if the table is too full resize it
    if (percent_full > INCREASE_SIZE_THREASHOLD) {
        // Set the new size of the table.
        int new_table_size = next_size();
        // Create the new table.
        broadcast_entry **new_table =
            new broadcast_entry *[new_table_size];
        for (int i = 0; i < new_table_size; i++) {
            new_table[i] = NULL;
        }
        // Copy entries to the new table.
        for (unsigned int i = 0; i < table_size_; i++) {
            // remove the linked nodes from the list
            if (table_[i] != NULL) {
                // remove all nodes at the end of the list
                while ((removed_entry = table_[i]->remove_tail())) {
                    removed_entry->next_ = NULL;
                    u_int32_t bucket = hash(removed_entry->mac_address_,
                                            new_table_size);
                    // add the entry to the new table.
                    if (new_table[bucket] != NULL) {
                        // add to the end of the list
                        new_table[bucket]->add(removed_entry);
                    } else {
                        // add to the head
                        new_table[bucket] = removed_entry;
                    }
                }
                // copy the head of the list to the new table
                if (table_[i] != NULL) {
                    table_[i]->next_ = NULL;
                    u_int32_t bucket = hash(table_[i]->mac_address_,
                                            new_table_size);
                    if (new_table[bucket] != NULL) {
                        new_table[bucket]->add(table_[i]);
                    } else {
                        new_table[bucket] = table_[i];
                    }
                }
            }
        }
        delete table_;          // delete the old table
```

```
        table_ = new_table;
        table_size_ = new_table_size;
        return 1;
    } else if (percent_full < DECREASE_SIZE_THREASHOLD
               && table_size_ != primes[0]) {
//      printf("need to reduce the size of the table");
// need to add code
        return -1;
    }
    return 0;
}



/*
 *  Removes the entries from the table that have not been
 *  updated within the timeout_ period.
 */
void broadcast_table::invalidate_old_entries(double current_time)
{
    double expiration_time = current_time - timeout_;
    for (unsigned int i = 0; i < table_size_; i++) {
        if (table_[i] != NULL) {
            // remove any invalid entries and set table_[i]
            // to the new head of the linked list
            table_[i] = table_[i]->remove(expiration_time);
        }
    }
}

/*
 *  Calculate the percentage of messages received in collision over the
 *  last interval.
 */
double broadcast_table::avg_reception_rate(double current_time)
{
    double total = 0;
    int nodes = 0;

    invalidate_old_entries(current_time);
//      printf("time %f of update\n", current_time);
    for (unsigned int i = 0; i < table_size_; i++) {
        if (table_[i] != NULL) {
            for (broadcast_entry * ent = table_[i];
                 ent != NULL; ent = ent->next_) {
                total += ent->avg_recp_rate_;
                nodes++;
            }
        }
    }
    if (nodes > 0) {
        return (total / nodes);
    } else {
        return 0;
    }
}

/* prime numbers to use for the hash table */
u_int32_t broadcast_table::primes[] =
```

```
    { 31, 67, 127, 257, 521, 1031, 2053, 4111 };



/*
int main(int argc, char** argv){

    broadcast_table* table = new broadcast_table(0.80, 1.0);
    int table_size = table->size();
    for(int i = 0; i < table_size -16; i++)
    table->update(i * 13, i, i * 0.07);
    table->update(1, 1, 0.21);
    table->update(1,2, 0.30);
    delete table;
}
*/
```

E.2 802.11e Source Code

E.2.1 mac-802_11e.h

```
//* -*- Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 * This product includes software developed by the Computer Systems
 * Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#ifndef ns_mac_80211e_h
#define ns_mac_80211e_h
```

```
#include "priq.h"
#include "mac-timers_802_11e.h"
#include "mac/mac-802_11.h"
#include "marshall.h"
#include "broadcast_table.h"

#define DEBUG 0
#define MAX_PRI 4               //max number of priority queues

#define STANDARD_CW 0           // use the standard CW
#define MODIFIED_CW 1           // use a modified CW
#define SLIDING_CW 2            // use the sliding CW

#define GET_ETHER_TYPE(x) GET2BYTE((x))
#define SET_ETHER_TYPE(x,y)            {u_int16_t t = (y); STORE2BYTE(x,&t);}
#define LEVEL(p) HDR_IP(p)->prio()

/* ======================================================================
   Frame Formats
   ====================================================================== */

struct QoS_control {
    u_char tid:4;
    u_char eosp:1;
    u_char ack_policy:2;

    u_char reserved:1;
    u_char txop_qs:8;
};

// This header does not have its header access function because it shares
// the same header space with hdr_mac.
struct hdr_mac802_11e {
    struct frame_control dh_fc;
    u_int16_t dh_duration;
    u_char dh_da[ETHER_ADDR_LEN];
    u_char dh_sa[ETHER_ADDR_LEN];
    u_char dh_bssid[ETHER_ADDR_LEN];
    u_int16_t dh_scontrol;
    u_char dh_adddr4[ETHER_ADDR_LEN];
    struct QoS_control dh_qos;
    u_char dh_body[0];          // XXX Non-ANSI
};

/* ======================================================================
   Definitions
   ====================================================================== */

#define DSSS_EDCA_MaxPropagationDelay        0.000002   // 2us

class EDCA_PHY_MIB {
    friend class Mac802_11e;
  public:
     EDCA_PHY_MIB(Mac802_11e * parent);

    inline double getSlotTime() {
        return (SlotTime);
    } inline double getSIFS() {
        return (SIFSTime);
```

```
    }
    inline double getPIFS() {
        return (SIFSTime + SlotTime);
    }
    inline double getDIFS() {
        return (SIFSTime + 2 * SlotTime);
    }
    inline double getEIFS() {
        return (SIFSTime + (8 * getACKlen()) / PLCPDataRate);
    }
    inline u_int32_t getPreambleLength() {
        return (PreambleLength);
    }
    inline double getPLCPDataRate() {
        return (PLCPDataRate);
    }

    inline u_int32_t getPLCPhdrLen() {
        return ((PreambleLength + PLCPHeaderLength) >> 3);
    }

    inline u_int32_t getHdrLen11() {
        return (getPLCPhdrLen() + sizeof(struct hdr_mac802_11e)
                + ETHER_FCS_LEN);
    }

    inline u_int32_t getRTSlen() {
        return (getPLCPhdrLen() + sizeof(struct rts_frame));
    }

    inline u_int32_t getCTSlen() {
        return (getPLCPhdrLen() + sizeof(struct cts_frame));
    }

    inline u_int32_t getACKlen() {
        return (getPLCPhdrLen() + sizeof(struct ack_frame));
    }

  private:
    double SlotTime;
    double SIFSTime;
    u_int32_t PreambleLength;
    u_int32_t PLCPHeaderLength;
    double PLCPDataRate;
};


class EDCA_MAC_MIB {
    friend class Mac802_11e;

  public:
     EDCA_MAC_MIB(Mac802_11e * parent);

  private:
    u_int32_t RTSThreshold;
    u_int32_t ShortRetryLimit;
    u_int32_t LongRetryLimit;

  public:
```

```
    u_int32_t FailedCount;
    u_int32_t RTSFailureCount;
    u_int32_t ACKFailureCount;

  public:
   inline u_int32_t getRTSThreshold() {
        return (RTSThreshold);
    }
    inline u_int32_t getShortRetryLimit() {
        return (ShortRetryLimit);
    }
    inline u_int32_t getLongRetryLimit() {
        return (LongRetryLimit);
    }
};

/* ======================================================================
   The actual 802.11e MAC class.
   ====================================================================== */
class Mac802_11e:public Mac {
    friend class DeferTimer_802_11e;
    friend class SIFSTimer_802_11e;
    friend class BackoffTimer_802_11e;
    friend class IFTimer_802_11e;
    friend class NavTimer_802_11e;
    friend class RxTimer_802_11e;
    friend class TxTimer_802_11e;
    friend class UpdateTableTimer;

  public:
     Mac802_11e();
    ~Mac802_11e();
    void recv(Packet * p, Handler * h);
    inline int hdr_dst(char *hdr, int dst = -2);
    inline int hdr_src(char *hdr, int src = -2);
    inline int hdr_type(char *hdr, u_int16_t type = 0);
    void setQ(PriQ * priqueue);
    PriQ *queue_;              // for pointer to Queues in priq.cc
    double getAIFS(int pri);
    void defer_stop(int pri);
    void calc_throughput();
    double idle_time;
  protected:
    inline void transmit(Packet * p, double t);
    inline void set_rx_state(MacState x);
    inline void set_tx_state(int pri, MacState x);
    void backoffHandler(int pri);
    void deferHandler(int pri);
    void navHandler(void);
    void recvHandler(void);
    void sendHandler(void);
    void txHandler(void);
    void updateTableHandler();  // update the CW for broadcast
    // methods for priority-parameters
    int getCW(int level);
    double interval_;

  private:
    int command(int argc, const char *const *argv);
```

```
/*
 *  Parameters for the modfied broadcast algorithm
 */

// hash table contain overheard broadcast messages
broadcast_table *reception_table_;
// timer to update the reception_table_
UpdateTableTimer mhUpdateTimer_;

int modified_;              // use modified broadcast algorithm
double update_interval_;    // interval to adjust contention window
int cw_UB_[MAX_PRI];        // cw upper bound for SLIDING_CW
int SF_[MAX_PRI];           // Sliding factor for SLIDING_CW
double WMA_alpha_;          // alpha for weighted average
double timeout_entries_;    // remove entires from table
double previous_recp_rate_; // the recption rate from the previous interval
double sliding_threshold_;  // the threashold value to determine to slide the window
double scaling_factor_;
int number_nodes_;
FILE *cw_file;

/*
 *  Added methods for the modified CW algorithm
 */
void increase_window();     // increase the CW for MODIFIED_CW
void decrease_window();     // decrease the CW for MODIFIED_CW
void slide_window_up();     // slide the CW up for SLIDING_CW
void slide_window_down();   // slide the CW down for SLIDING)CW
int get_backoff(int prio);  // get a backoff for the priority level for SLIDING_CW


/*
 *  QoS parameters
 */
void check_backoff_timer();
bool AIFSset;
bool CWset;
int cw_[MAX_PRI];
int cwmin_[MAX_PRI];
int cwmax_[MAX_PRI];
double txop_limit_[MAX_PRI];
Handler *callback_[MAX_PRI];

/*
 * Called by the timers.
 */
void recv_timer(void);
void send_timer(void);
int check_pktCTRL(int pri);
int check_pktRTS(int pri);
int check_pktTx(int pri);
int levels;
int slotnum;
double aifs_[MAX_PRI];
Packet *packets_[MAX_PRI];

/*
 * Packet Transmission Functions.
 */
```

```
void send(Packet * p, Handler * h);
void sendRTS(int pri, int dst);
void sendCTS(int pri, int dst, double duration);
void sendACK(int pri, int dst);
void sendDATA(int pri, Packet * p);
void RetransmitRTS(int pri);
void RetransmitDATA(int pri);

/*
 * Packet Reception Functions.
 */
void recvRTS(Packet * p);
void recvCTS(Packet * p);
void recvACK(Packet * p);
void recvDATA(Packet * p);

void capture(Packet * p);
void collision(Packet * p);
void discard(Packet * p, const char *why);
void rx_resume(void);
void tx_resume(void);

inline int is_idle(void);

/*
 * Debugging Functions.
 */
void trace_pkt(Packet * p);
void dump(char *fname);

inline int initialized() {
    return (cache_ && logtarget_ && Mac::initialized());
} void mac_log(Packet * p) {
    logtarget_->recv(p, (Handler *) 0);
}
double txtime(Packet * p);
double txtime(double psz, double drt);
double txtime(int bytes) {
    /* clobber inherited txtime() */
    abort();
}

inline void inc_cw(int level) {
    //get persistence factor
    pf = queue_->pri_[level].getPF();
    cw_old = cw_[level];
    //calculate new cw_[pri]
    cw_[level] = ((cw_old + 1) * pf) - 1;
    if (cw_[level] > cwmax_[level]) {
        cw_[level] = cwmax_[level];
    }
}
inline void rst_cw(int level) {
    cw_[level] = cwmin_[level];
}

inline double sec(double t) {
    return (t *= 1.0e-6);
}
```

```
    inline u_int16_t usec(double t) {
        u_int16_t us = (u_int16_t) floor((t *= 1e6) + 0.5);
        return us;
    }
    inline void set_nav(u_int16_t us) {
        double now = Scheduler::instance().clock();
        double t = us * 1e-6;

        if ((now + t) > nav_) {
            nav_ = now + t;
            if (mhNav_.busy()) {
                mhNav_.stop();

            }
            mhNav_.start(t);
        }
    }

    inline void reset_eifs_nav();
    bool inc_retryCounter(int pri);

  protected:
    EDCA_PHY_MIB phymib_;
    EDCA_MAC_MIB macmib_;

  private:
    double eifs_nav_;
    double basicRate_;
    double dataRate_;
    int numbytes_[MAX_PRI];             // for Akaroa Observation
    double start_handle_[MAX_PRI];      // for delay investigation
    double throughput;
    double jitter;
    int rtx_[MAX_PRI];
    int pf;
    int cw_old;

    /*
     * Contention Free Burst
     */
    int cfb_;
    int cfb_broadcast;
    double cfb_dur;
    int cfb_active;
    void cfb(int pri);

    /*
     * Mac Timers
     */
    IFTimer_802_11e mhIF_;      // interface timer
    NavTimer_802_11e mhNav_;    // NAV timer
    RxTimer_802_11e mhRecv_;    // incoming packets
    TxTimer_802_11e mhSend_;    // outgoing packets

    DeferTimer_802_11e mhDefer_;      // defer timer
    SIFSTimer_802_11e mhSifs_;        // defer timer for sifs
    BackoffTimer_802_11e mhBackoff_;  // backoff timer
//    AkaroaTimer AK;
    /* ============================================================
```

```
      Internal MAC State
      ============================================================ */
   double nav_;              // Network Allocation Vector

   MacState rx_state_;       // incoming state (MAC_RECV or MAC_IDLE)
   MacState tx_state_[MAX_PRI]; // outgoint state
   int tx_active_;           // transmitter is ACTIVE
   int sending;              // transmitter is ACTIVE
   Packet *pktRTS_[MAX_PRI];   // outgoing RTS packet
   Packet *pktCTRL_[MAX_PRI];  // outgoing non-RTS packet
   Packet *pktTx_[MAX_PRI];

   u_int32_t ssrc_[MAX_PRI];   // STA Short Retry Count
   u_int32_t slrc_[MAX_PRI];   // STA Long Retry Count
   double sifs_;             // Short Interface Space
   double pifs_;             // PCF Interframe Space
   double difs_;             // DCF Interframe Space
   double eifs_;             // Extended Interframe Space

   NsObject *logtarget_;

   /* ============================================================
      Duplicate Detection state
      ============================================================ */
   u_int16_t sta_seqno_;     // next seqno that I'll use
   int cache_node_count_;
   Host *cache_;
};
#endif                       /* __mac_80211e_h__ */
```

E.2.2 mac-802_11e.cc

```
//* -*- Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 * This product includes software developed by the Computer Systems
 * Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
```

```
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#undef NDEBUG

#include <assert.h>
#include "delay.h"
#include "connector.h"
#include "packet.h"
#include "random.h"
#include "mobilenode.h"
#include "arp.h"
#include "mac-timers_802_11e.h"
#include "mac-802_11e.h"
#include "cmu-trace.h"
#include "priq.h"


/*
 * set the rx state
 */
inline void Mac802_11e::set_rx_state(MacState x)
{
    rx_state_ = x;
    check_backoff_timer();
}


/*
 * set the tx state
 */
inline void Mac802_11e::set_tx_state(int pri, MacState x)
{
    tx_state_[pri] = x;
}


/*
 *  Sends the MAC frame to Phy for transmission.
 */
inline void Mac802_11e::transmit(Packet * p, double t)
{
    /*
     * If I'm transmitting without doing CS, such as when
     * sending an ACK, any incoming packet will be "missed"
     * and hence, must be discarded.
     */
    if (rx_state_ != MAC_IDLE) {
        struct hdr_mac802_11e *dh = HDR_MAC802_11E(p);

        assert(dh->dh_fc.fc_type == MAC_Type_Control);
        assert(dh->dh_fc.fc_subtype == MAC_Subtype_ACK);

        assert(pktRx_);
        struct hdr_cmn *ch = HDR_CMN(pktRx_);
```

```
        ch->error() = 1;        /* force packet discard */
    }

    /*
     * pass the packet on the "interface" which will in turn
     * place the packet on the channel.
     *
     * NOTE: a handler is passed along so that the Network
     *       Interface can distinguish between incoming and
     *       outgoing packets.
     */
    struct hdr_cmn *sd = HDR_CMN(p);
    int prio = LEVEL(p);        // get the priority of the packet
    tx_active_ = 1;             // indicate the MAC is transmitting
    sending = 1;                // indicate the mac is sending
    check_backoff_timer();

    // give the packet to the physical layer
    downtarget_->recv(p->copy(), this);
    if (sd->ptype() == PT_CBR || sd->ptype() == PT_EXP) {
        if (!rtx_[prio]) {
            numbytes_[prio] += sd->size() - phymib_.getHdrLen11();
        }
    }
    mhIF_.start(txtime(p));     // transmission probably failed
    mhSend_.start(t);           // expires then the MAC knows
    // Phy has completed sending


}

/*
 *  Check the back off timer
 */
void Mac802_11e::check_backoff_timer()
{
    // if medium is idle and backoff timer paused resume backoff timer.
    if (is_idle() && mhBackoff_.paused()) {
        mhBackoff_.resume();
    }
    // if the medium is not idle and the backoff timer is running pause the backoff.
    if (!is_idle() && mhBackoff_.busy()
        && !mhBackoff_.paused()) {
        mhBackoff_.pause();
    }
    if (!is_idle() && mhDefer_.busy())
        mhDefer_.stop();
}

/* ======================================================================
   Global Variables
   ====================================================================== */

EDCA_PHY_MIB::EDCA_PHY_MIB(Mac802_11e * parent)
{
    /*
     * Bind the phy mib objects.  Note that these will be bound
     * to Mac/802_11e variables
```

```
     */
    parent->bind("SlotTime_", &SlotTime);
    parent->bind("SIFS_", &SIFSTime);
    parent->bind("PreambleLength_", &PreambleLength);
    parent->bind("PLCPHeaderLength_", &PLCPHeaderLength);
    parent->bind_bw("PLCPDataRate_", &PLCPDataRate);
}

EDCA_MAC_MIB::EDCA_MAC_MIB(Mac802_11e * parent)
{
    /*
     * Bind the phy mib objects.  Note that these will be bound
     * to Mac/802_11 variables
     */

    parent->bind("RTSThreshold_", &RTSThreshold);
    parent->bind("ShortRetryLimit_", &ShortRetryLimit);
    parent->bind("LongRetryLimit_", &LongRetryLimit);
}

/* ======================================================================
   TCL Hooks for the simulator
   ====================================================================== */
static class Mac802_11eClass:public TclClass {
  public:
    Mac802_11eClass():TclClass("Mac/802_11e") {
    } TclObject *create(int, const char *const *) {
        return (new Mac802_11e());
    }
}

class_mac802_11e;


/* ======================================================================
   Mac Class Functions
   ====================================================================== */
Mac802_11e::Mac802_11e():
Mac(), phymib_(this), macmib_(this), mhIF_(this),
mhNav_(this), mhRecv_(this), mhSend_(this),
mhDefer_(this, phymib_.getSlotTime()),
mhSifs_(this, phymib_.getSlotTime()),
mhBackoff_(this, phymib_.getSlotTime()), mhUpdateTimer_(this)
{
    // Pointer to PriQ, Cast in priq.cc
    queue_ = 0;

    //flags to control if PriQ Parameters have already been adopted
    AIFSset = 0;
    CWset = 0;
    for (int i = 0; i < MAX_PRI; i++) {
        packets_[i] = 0;
        pktRTS_[i] = 0;
        pktCTRL_[i] = 0;
        pktTx_[i] = 0;
        tx_state_[i] = MAC_IDLE;
        ssrc_[i] = slrc_[i] = 0;
        callback_[i] = 0;
        numbytes_[i] = 0;
```

```
    rtx_[i] = 0;
    cw_[i] = 0;
    cwmin_[i] = 0;
    cwmax_[i] = 0;
    aifs_[i] = 0;
    txop_limit_[i] = 0;
    start_handle_[i] = 0;
}
jitter = 1000;
throughput = 0;

nav_ = 0.0;

rx_state_ = MAC_IDLE;
tx_active_ = 0;

idle_time = 0;
sending = 0;
cfb_dur = 0;
cfb_active = 0;
cfb_broadcast = 0;

levels = 0;
slotnum = 0;
pf = 0;
cw_old = 0;

sifs_ = phymib_.getSIFS();
pifs_ = phymib_.getPIFS();
difs_ = phymib_.getDIFS();

// see (802.11-1999, 9.2.10)
eifs_ = phymib_.getDIFS();

eifs_nav_ = 0;

sta_seqno_ = 1;
cache_ = 0;
cache_node_count_ = 0;

// chk if basic/data rates are set
// otherwise use bandwidth_ as default;

Tcl & tcl = Tcl::instance();
tcl.evalf("Mac/802_11e set basicRate_");
if (strcmp(tcl.result(), "0") != 0)
    bind_bw("basicRate_", &basicRate_);
else
    basicRate_ = bandwidth_;

tcl.evalf("Mac/802_11e set dataRate_");
if (strcmp(tcl.result(), "0") != 0)
    bind_bw("dataRate_", &dataRate_);
else
    dataRate_ = bandwidth_;

bind("cfb_", &cfb_);

// added to the constructor for the modified algorithm
```

```
    modified_ = 0;
    update_interval_ = 0;
    WMA_alpha_ = 0;
    timeout_entries_ = 0;
    previous_recp_rate_ = 0;
    sliding_threshold_ = 0;

    bind("modified_", &modified_);
    bind("update_interval_", &update_interval_);
    bind("WMA_alpha_", &WMA_alpha_);
    bind("timeout_entries_", &timeout_entries_);
    bind("sliding_threshold_", &sliding_threshold_);
    bind("scaling_factor_", &scaling_factor_);
    bind("number_nodes_", &number_nodes_);

    // If the modified broadcast is used create the table
    // to store the overheard messages.
    if (modified_) {
        reception_table_ =
            new broadcast_table(WMA_alpha_, timeout_entries_);
    }
    // If the modified broadcast is used set the timer to update
    // the table contain overheard broadcasts.
    if (update_interval_ != 0 && modified_) {
        mhUpdateTimer_.start(update_interval_);
    }
}


/*
 *  Mac802_11e detructor to free the broadcast table
 */
Mac802_11e::~Mac802_11e()
{
    if (reception_table_ != NULL) {
        delete reception_table_;
    }
}

/*
 *  TCL hooks for the 802.11e class
 */
int Mac802_11e::command(int argc, const char *const *argv)
{
    if (argc == 3) {
        if (strcmp(argv[1], "log-target") == 0) {
            logtarget_ = (NsObject *) TclObject::lookup(argv[2]);
            if (logtarget_ == 0) {
                return TCL_ERROR;
            }
            return TCL_OK;
        } else if (strcmp(argv[1], "nodes") == 0) {
            if (cache_) {
                return TCL_ERROR;
            }
            cache_node_count_ = atoi(argv[2]);
            cache_ = new Host[cache_node_count_ + 1];
            assert(cache_);
```

```
            bzero(cache_, sizeof(Host) * (cache_node_count_ + 1));
            return TCL_OK;
        }
    }
    return Mac::command(argc, argv);
}

/* ======================================================================
   Debugging Routines
   ====================================================================== */

/*
 * dump and packet trace are not adopted to 802.11e yet!
 */

void Mac802_11e::trace_pkt(Packet * p)
{
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_mac802_11e *dh = HDR_MAC802_11E(p);
    u_int16_t *t = (u_int16_t *) & dh->dh_fc;

    fprintf(stderr, "\t[ %2x %2x %2x %2x ] %x %s %d\n",
            *t, dh->dh_duration,
            ETHER_ADDR(dh->dh_da), ETHER_ADDR(dh->dh_sa),
            index_, packet_info.name(ch->ptype()), ch->size());
}

void Mac802_11e::dump(char *fname)
{
    fprintf(stderr,
            "\tDefer: %d, Backoff: %d (%d), Recv: %d, Timer: %d Nav: %d\n",
            mhDefer_.busy(), mhBackoff_.busy(),
            mhBackoff_.paused(), mhRecv_.busy(),
            mhSend_.busy(), mhNav_.busy());
    fprintf(stderr, "\tBackoff Expire: %f\n", mhBackoff_.expire());
}


/* ======================================================================
   Packet Headers Routines
   ====================================================================== */

/*
 *  Add destination to the MAC header
 */
inline int Mac802_11e::hdr_dst(char *hdr, int dst)
{
    struct hdr_mac802_11e *dh = (struct hdr_mac802_11e *) hdr;

    if (dst > -2) {
        STORE4BYTE(&dst, (dh->dh_da));
    }
    return ETHER_ADDR(dh->dh_da);
}


/*
 *  Set the 802.11 MAC source address.
 */
```

```
inline int Mac802_11e::hdr_src(char *hdr, int src)
{
    struct hdr_mac802_11e *dh = (struct hdr_mac802_11e *) hdr;
    // if a valid souce address
    if (src > -2) {
        STORE4BYTE(&src, (dh->dh_sa));
    }
    return ETHER_ADDR(dh->dh_sa);
}


/*
 *  Set the 802.11 dh_body
 */
inline int Mac802_11e::hdr_type(char *hdr, u_int16_t type)
{
    struct hdr_mac802_11e *dh = (struct hdr_mac802_11e *) hdr;
    if (type) {
        STORE2BYTE(&type, (dh->dh_body));
    }
    return GET2BYTE(dh->dh_body);
}


/* ======================================================================
   Misc Routines
   ====================================================================== */

/*
 *  check the tx and rx state and the NAV
 */
inline int Mac802_11e::is_idle()
{
    if (rx_state_ != MAC_IDLE) {
        idle_time = 0;
        return 0;
    }
    if (sending) {
        idle_time = 0;
        return 0;
    }

    if (nav_ > Scheduler::instance().clock()) {
        idle_time = 0;
        return 0;
    }
    idle_time = Scheduler::instance().clock();
    return 1;
}


/*
 *  Frees the memory allocated to a packet
 */
void Mac802_11e::discard(Packet * p, const char *why)
{
    hdr_mac802_11e *mh = HDR_MAC802_11E(p);
    hdr_cmn *ch = HDR_CMN(p);
```

```
#if 0
    /* old logic 8/8/98 -dam */
    /*
     * If received below the RXThreshold, then just free.
     */
    if (p->txinfo_.Pr < p->txinfo_.ant.RXThresh) {
        Packet::free(p);
        //p = 0;
        return;
    }
#endif                          // 0

    /* if the rcvd pkt contains errors, a real MAC layer couldn't
       necessarily read any data from it, so we just toss it now */
    if (ch->error() != 0) {
        Packet::free(p);
        //p = 0;
        return;
    }

    switch (mh->dh_fc.fc_type) {
    case MAC_Type_Management:
        drop(p, why);
        return;
    case MAC_Type_Control:
        switch (mh->dh_fc.fc_subtype) {
        case MAC_Subtype_RTS:
            if ((u_int32_t) ETHER_ADDR(mh->dh_sa) == (u_int32_t) index_) {
                drop(p, why);
                return;
            }
            /* fall through - if necessary */
        case MAC_Subtype_CTS:
        case MAC_Subtype_ACK:
            if ((u_int32_t) ETHER_ADDR(mh->dh_da) == (u_int32_t) index_) {
                drop(p, why);
                return;
            }
            break;
        default:
            fprintf(stderr, "invalid MAC Control subtype\n");
            exit(1);
        }
        break;
    case MAC_Type_Data:
        switch (mh->dh_fc.fc_subtype) {
        case MAC_Subtype_Data:
            if ((u_int32_t) ETHER_ADDR(mh->dh_da) ==
                (u_int32_t) index_ ||
                (u_int32_t) ETHER_ADDR(mh->dh_sa) ==
                (u_int32_t) index_ ||
                (u_int32_t) ETHER_ADDR(mh->dh_da) == MAC_BROADCAST) {
                drop(p);
                return;
            }
            break;
        default:
            fprintf(stderr, "invalid MAC Data subtype\n");
```

```
            exit(1);
        }
        break;
    default:
        fprintf(stderr, "invalid MAC type (%x)\n", mh->dh_fc.fc_type);
        trace_pkt(p);
        exit(1);
    }
    Packet::free(p);
}


/*
 *  Called if a second packet is received while
 *  the MAC is currently receiving a packet.
 */
void Mac802_11e::capture(Packet * p)
{
    /*
     * Update the NAV so that this does not screw
     * up carrier sense.
     */
    set_nav(usec(eifs_ + txtime(p)));
    Packet::free(p);
}


/*
 *  Called if a collision occurs.
 */
void Mac802_11e::collision(Packet * p)
{
    switch (rx_state_) {
    case MAC_RECV:
        // Set the state to MAC collision
        set_rx_state(MAC_COLL);
        /* fall through */
    case MAC_COLL:
        assert(pktRx_);
        assert(mhRecv_.busy());
        /*
         *  Since a collision has occurred, figure out
         *  which packet that caused the collision will
         *  "last" the longest.  Make this packet,
         *  pktRx_ and reset the Recv Timer if necessary.
         */
        if (txtime(p) > mhRecv_.expire()) {
            mhRecv_.stop();
            discard(pktRx_, DROP_MAC_COLLISION);
            pktRx_ = p;
            //reset the receive timer
            mhRecv_.start(txtime(pktRx_));
        } else {
            discard(p, DROP_MAC_COLLISION);
        }
        break;
    default:
        assert(0);
    }
```

```
}


/*
 *  Used to set some timers when the MAC
 *  is ready to send a packet.
 */
void Mac802_11e::tx_resume()
{
    double rTime;
    assert(mhSend_.busy() == 0);

    for (int pri = 0; pri < MAX_PRI; pri++) {
        //assert(mhDefer_.defer(pri) == 0);
        if (!mhDefer_.defer(pri)) {
            // If a CTS or an ACK is waiting to
            // be sent set defer time to SIFS.
            if (pktCTRL_[pri]) {
                //  Need to send a CTS or ACK.
                mhSifs_.start(pri, sifs_);
                // If a RTS is waiting to be sent
            } else if (pktRTS_[pri]) {
                // Make sure the backoff timer isn't busy.
                if (mhBackoff_.backoff(pri) == 0) {
                    // Start the defer timer
                    if (modified_ == SLIDING_CW) {
                        rTime =
                            get_backoff(LEVEL(pktRTS_[pri])) *
                            phymib_.getSlotTime();
                    } else {
                        rTime =
                            (Random::random() %
                             getCW(LEVEL(pktRTS_[pri]))) *
                            phymib_.getSlotTime();
                    }
                    mhDefer_.start(pri, getAIFS(LEVEL(pktRTS_[pri])));
                }
                // If a data packet is next to be sent.
            } else if (pktTx_[pri]) {
                // check that the isn't currently backing off
                if (mhBackoff_.backoff(pri) == 0) {
                    // get common header
                    hdr_cmn *ch = HDR_CMN(pktTx_[pri]);
                    // get mac header
                    struct hdr_mac802_11e *mh =
                        HDR_MAC802_11E(pktTx_[pri]);
                    // if RTS was not sent before the data packet
                    // set the backoff and DIFS
                    if ((u_int32_t) ch->size() <
                        macmib_.RTSThreshold || (u_int32_t)
                        ETHER_ADDR(mh->dh_da) == MAC_BROADCAST) {
                        if ((u_int32_t)
                            ETHER_ADDR(mh->dh_da) == MAC_BROADCAST) {
                            if (modified_ == SLIDING_CW) {
                                rTime = get_backoff(pri) *
                                    phymib_.getSlotTime();
                            } else {
                                rTime =
                                    (Random::random() %
```

```
                                    (getCW(pri) + 1)) *
                                   phymib_.getSlotTime();
                           }
                       } else {
                           rTime = 0;
                       }
                       mhDefer_.start(pri, getAIFS(pri) + rTime);
                   } else {
                       // The RTS was already sent, so use the SIFS.
                       mhSifs_.start(pri, sifs_);
                   }
               }
               // if no packets are waiting to be sent but the callback
               // is defined, handle the callback to complete the transmission.
           } else if (callback_[pri] != 0) {
               rtx_[pri] = 0;
               Handler *h = callback_[pri];
               callback_[pri] = 0;
               h->handle((Event *) 0);
           }
           // set the MAC state to idle.
           set_tx_state(pri, MAC_IDLE);
       }
   }
}


/*
 *  Called after the recv_timer has completed.
 */
void Mac802_11e::rx_resume()
{
    assert(pktRx_ == 0);
    assert(mhRecv_.busy() == 0);
    set_rx_state(MAC_IDLE);
}


/* ======================================================================
   Timer Handler Routines
   ====================================================================== */


/*
 *  backoffhandler is called when ever the backoff timer expires.
 *  At the expiration of the backoffhandler either a rts or a data
 *  packet should be transmitted.
 */
void Mac802_11e::backoffHandler(int pri)
{
    // Check if there is a control packet waiting to be sent.
    if (pktCTRL_[pri]) {
        assert(mhSend_.busy() || mhDefer_.defer(pri));
        return;
    }

    if (check_pktRTS(pri) == 0) {
        return;
    }
```

```
    if (check_pktTx(pri) == 0) {
        printf("backoffhandler (check_pktTx(pri) == 0)\n");
        return;
    }
}


/*
 *  The function is called when the defer timer expires.
 *  The node will now attempt a transmission.
 */
void Mac802_11e::deferHandler(int pri)
{
    assert(pktCTRL_[pri] || pktRTS_[pri] || pktTx_[pri]);
    if (check_pktCTRL(pri) == 0) {
        return;
    }
    assert(mhBackoff_.backoff(pri) == 0);
    if (check_pktRTS(pri) == 0) {
        return;
    }
    if (check_pktTx(pri) == 0) {
        return;
    }
}


/*
 *  Resume the backoff timer if medium is idle
 *  and backoff is paused.
 */
void Mac802_11e::navHandler()
{
    eifs_nav_ = 0.0;
    if (is_idle() && mhBackoff_.paused()) {
        mhBackoff_.resume();
    }
}


/*
 *  Call the recv_timer
 */
void Mac802_11e::recvHandler()
{
    recv_timer();
}


/*
 *  Call the send_timer
 */
void Mac802_11e::sendHandler()
{
    sending = 0;
    check_backoff_timer();
    send_timer();
}
```

```
/*
 *  Clear the tx_active_ flag to indicate
 *  the transmission is no longer active.
 */
void Mac802_11e::txHandler()
{
    tx_active_ = 0;
    if (cfb_ && !cfb_broadcast)
        cfb_active = 0;
    if (cfb_broadcast)
        cfb_broadcast = 0;
}

/*
 *  start the backoff timer once the defer timer stops
 */
void Mac802_11e::defer_stop(int pri)
{
    if (modified_ == SLIDING_CW) {
        mhBackoff_.start(pri, get_backoff(pri), is_idle());
    } else {
        mhBackoff_.start(pri, getCW(pri), is_idle());
    }
}

/* ======================================================================
   The "real" Timer Handler Routines
   ====================================================================== */

/*
 *  Called at the expiration of the TxTimer, mhSend_.
 *  The timer expires after the calculated timeout.
 */
void Mac802_11e::send_timer()
{
    for (int pri = 0; pri < MAX_PRI; pri++) {
        switch (tx_state_[pri]) {
            /*
             * Sent a RTS, but did not receive a CTS.
             */
        case MAC_RTS:
            RetransmitRTS(pri);
            break;
            /*
             * Sent a CTS, but did not receive a DATA packet.
             */
        case MAC_CTS:
            assert(pktCTRL_[pri]);
            Packet::free(pktCTRL_[pri]);
            pktCTRL_[pri] = 0;
            break;
            /*
             * Sent DATA, but did not receive an ACK packet.
             */
        case MAC_SEND:
            RetransmitDATA(pri);
            break;
            /*
```

```
             * Sent an ACK, and now ready to resume transmission.
             */
        case MAC_ACK:
            assert(pktCTRL_[pri]);
            Packet::free(pktCTRL_[pri]);
            pktCTRL_[pri] = 0;
            break;
        case MAC_IDLE:
            break;
        default:
            assert(0);
        }
    }
    if (!cfb_active) {
        tx_resume();
    }
}

/* ======================================================================
   Outgoing Packet Routines
   ====================================================================== */

/*
 *  Transmits a CTS or an ACK.
 */
int Mac802_11e::check_pktCTRL(int pri)
{

    struct hdr_mac802_11e *mh;
    double timeout;

    //  Check if pktCTRL_ points to a packet, if not return -1
    //  indicating that nothing was transmitted.
    if (pktCTRL_[pri] == 0) {
        return -1;
    }
    // Check if it is currently transmitting a CTS or ACK.
    if (tx_state_[pri] == MAC_CTS || tx_state_[pri] == MAC_ACK) {
        return -1;
    }

    mh = HDR_MAC802_11E(pktCTRL_[pri]);

    // Perform a switch based on the type of control packet.
    switch (mh->dh_fc.fc_subtype) {
        //If the medium is not IDLE, don't send the CTS.
    case MAC_Subtype_CTS:
        if (!is_idle()) {
            discard(pktCTRL_[pri], DROP_MAC_BUSY);
            pktCTRL_[pri] = 0;
            return 0;
        }
        // Set the TX state to indicate the MAC is sending a CTS.
        set_tx_state(pri, MAC_CTS);

        // timeout:  cts + data tx time calculated by
        //           adding cts tx time to the cts duration
        //           minus ack tx time -- this timeout is
        //           a guess since it is unspecified
```

```
        //           (note: mh->dh_duration == cf->cf_duration)
        timeout = txtime(phymib_.getCTSlen(), basicRate_)
            + DSSS_EDCA_MaxPropagationDelay + sec(mh->dh_duration)
            + DSSS_EDCA_MaxPropagationDelay
            - sifs_ - txtime(phymib_.getACKlen(), basicRate_);

        break;

        // IEEE 802.11 specs, section 9.2.8
        // Acknowledments are sent after an SIFS, without regard to
        // the busy/idle state of the medium.
        //
    case MAC_Subtype_ACK:
        set_tx_state(pri, MAC_ACK);
        // Calculate the timeout value.
        timeout = txtime(phymib_.getACKlen(), basicRate_);
        break;
    default:
        fprintf(stderr, "check_pktCTRL:Invalid MAC Control subtype\n");
        exit(1);
    }
    // Give the physical layer the packet for
    // transmission with the timeout.
    transmit(pktCTRL_[pri], timeout);
    return 0;
}


/*
 *  Transmits a RTS packet.
 */
int Mac802_11e::check_pktRTS(int pri)
{

    struct hdr_mac802_11e *mh;
    double timeout;

    assert(mhBackoff_.backoff(pri) == 0);
    // If there is no RTS packet to send return -1 to
    // indicate that no packet was sent.
    if (pktRTS_[pri] == 0) {
        return -1;
    }
    // get the MAC header
    mh = HDR_MAC802_11E(pktRTS_[pri]);

    // detect the packet type
    switch (mh->dh_fc.fc_subtype) {
    case MAC_Subtype_RTS:
        // check if the medium is busy
        if (!is_idle()) {
            // increase the contention window
            // added to not increase the CW for the modified algorithm
            if (modified_ == SLIDING_CW) {
                mhBackoff_.start(pri, get_backoff(pri), is_idle());
            } else if (modified_ == MODIFIED_CW) {
                mhBackoff_.start(pri, getCW(pri), is_idle());
            } else {
                inc_cw(pri);
```

```
                mhBackoff_.start(pri, getCW(pri), is_idle());
            }
            // start backoff timer
            // return without transmitting if busy
            return 0;
        }
        // Set the MAC state to RTS
        set_tx_state(pri, MAC_RTS);
        // Calculate the timeout for the RTS.
        timeout = txtime(phymib_.getRTSlen(), basicRate_)
            + DSSS_EDCA_MaxPropagationDelay     // XXX
            + sifs_ + txtime(phymib_.getCTSlen(), basicRate_)
            + DSSS_EDCA_MaxPropagationDelay;    // XXX
        break;
    default:
        fprintf(stderr, "check_pktRTS:Invalid MAC Control subtype\n");
        exit(1);
    }
    // Give the physical layer the packet to send.
    transmit(pktRTS_[pri], timeout);
    return 0;
}


/*
 *  Transmits the data packet.
 */
int Mac802_11e::check_pktTx(int pri)
{

    struct hdr_mac802_11e *mh;  // MAC header
    double timeout;             // timeout
    assert(mhBackoff_.backoff(pri) == 0);

    // If there is no packet to send signal by returning -1.
    if (pktTx_[pri] == 0) {
        return -1;
    }
    // Get the MAC header.
    mh = HDR_MAC802_11E(pktTx_[pri]);

    // determine the type of packet
    switch (mh->dh_fc.fc_subtype) {
    case MAC_Subtype_Data:
        // Set the MAC state to MAC_SEND
        set_tx_state(pri, MAC_SEND);
        // Set the timeout for a non-broadcast packet.
        if ((u_int32_t) ETHER_ADDR(mh->dh_da) != MAC_BROADCAST)
            timeout = txtime(pktTx_[pri])
                + DSSS_EDCA_MaxPropagationDelay
                + sifs_ + txtime(phymib_.getACKlen(), basicRate_)
                + DSSS_EDCA_MaxPropagationDelay;
        else
            // Set the timeout for the broadcast packet.
            timeout = txtime(pktTx_[pri]);
        break;
    default:
        fprintf(stderr, "check_pktTx:Invalid MAC Control subtype\n");
        exit(1);
```

```
    }
    // Begin the transmission of the data packet.
    transmit(pktTx_[pri], timeout);
    return 0;
}

/*
 * Low-level transmit functions that actually place the packet onto
 * the channel.
 */

/*
 *  Create the RTS packet for the destination.
 */
void Mac802_11e::sendRTS(int pri, int dst)
{

    Packet *p = Packet::alloc();        // Allocate a packet.
    hdr_cmn *ch = HDR_CMN(p);
    struct rts_frame *rf =
        (struct rts_frame *) p->access(hdr_mac::offset_);

    assert(pktTx_[pri]);
    assert(pktRTS_[pri] == 0);

    /*
     *  If the size of the packet is larger than the
     *  RTSThreshold, then perform the RTS/CTS exchange.
     *
     *  XXX: also skip if destination is a broadcast
     */
    if ((u_int32_t) HDR_CMN(pktTx_[pri])->size() <
        macmib_.RTSThreshold || (u_int32_t) dst == MAC_BROADCAST) {
        Packet::free(p);
        return;
    }
    //  Initialize the the RTS header fields.
    ch->uid() = 0;
    ch->ptype() = PT_MAC;
    ch->size() = phymib_.getRTSlen();
    ch->iface() = -2;
    ch->error() = 0;

    bzero(rf, MAC_HDR_LEN);

    rf->rf_fc.fc_protocol_version = MAC_ProtocolVersion;
    rf->rf_fc.fc_type = MAC_Type_Control;
    rf->rf_fc.fc_subtype = MAC_Subtype_RTS;
    rf->rf_fc.fc_to_ds = 0;
    rf->rf_fc.fc_from_ds = 0;
    rf->rf_fc.fc_more_frag = 0;
    rf->rf_fc.fc_retry = 0;
    rf->rf_fc.fc_pwr_mgt = 0;
    rf->rf_fc.fc_more_data = 0;
    rf->rf_fc.fc_wep = 0;
    rf->rf_fc.fc_order = 0;

    STORE4BYTE(&dst, (rf->rf_ra));
```

```
    // store rts tx time
    ch->txtime() = txtime(ch->size(), basicRate_);
    // Set the senders address.
    STORE4BYTE(&index_, (rf->rf_ta));
    // calculate rts duration field
    rf->rf_duration = usec(sifs_ + txtime(phymib_.getCTSlen(), basicRate_)
                           + sifs_ + txtime(pktTx_[pri])
                           + sifs_ + txtime(phymib_.getACKlen(),
                                            basicRate_));

    // Assign the createD RTS packet to pktRTS_
    // and return control to send.
    pktRTS_[pri] = p;
}


/*
 *  Create the CTS packet
 */
void Mac802_11e::sendCTS(int pri, int dst, double rts_duration)
{
    // allocate a new packet
    Packet *p = Packet::alloc();
    hdr_cmn *ch = HDR_CMN(p);
    struct cts_frame *cf =
        (struct cts_frame *) p->access(hdr_mac::offset_);

    assert(pktCTRL_[pri] == 0);

    // Initialize the packet header to the default values.
    ch->uid() = 0;
    ch->ptype() = PT_MAC;
    ch->size() = phymib_.getCTSlen();
    ch->iface() = -2;
    ch->error() = 0;
    bzero(cf, MAC_HDR_LEN);

    cf->cf_fc.fc_protocol_version = MAC_ProtocolVersion;
    cf->cf_fc.fc_type = MAC_Type_Control;
    cf->cf_fc.fc_subtype = MAC_Subtype_CTS;
    cf->cf_fc.fc_to_ds = 0;
    cf->cf_fc.fc_from_ds = 0;
    cf->cf_fc.fc_more_frag = 0;
    cf->cf_fc.fc_retry = 0;
    cf->cf_fc.fc_pwr_mgt = 0;
    cf->cf_fc.fc_more_data = 0;
    cf->cf_fc.fc_wep = 0;
    cf->cf_fc.fc_order = 0;


    STORE4BYTE(&dst, (cf->cf_ra));

    /* store cts tx time */
    ch->txtime() = txtime(ch->size(), basicRate_);

    /* calculate cts duration */
    cf->cf_duration = usec(sec(rts_duration)
                           - sifs_
                           - txtime(phymib_.getCTSlen(), basicRate_));
```

```
    // Assign the newly created CTS packet to pktCTRL_
    // control returns to recvRTS
    pktCTRL_[pri] = p;

}


/*
 *  Create the ACK packet to be sent in response to the data packet.
 */
void Mac802_11e::sendACK(int pri, int dst)
{

    Packet *p = Packet::alloc();
    hdr_cmn *ch = HDR_CMN(p);
    struct ack_frame *af =
        (struct ack_frame *) p->access(hdr_mac::offset_);
    assert(pktCTRL_[pri] == 0);

    ch->uid() = 0;              // ACK-UID
    ch->ptype() = PT_MAC;
    ch->size() = phymib_.getACKlen();
    ch->iface() = -2;
    ch->error() = 0;
    HDR_IP(p)->prio() = pri;    //same priority as data packet
    bzero(af, MAC_HDR_LEN);

    af->af_fc.fc_protocol_version = MAC_ProtocolVersion;
    af->af_fc.fc_type = MAC_Type_Control;
    af->af_fc.fc_subtype = MAC_Subtype_ACK;
    af->af_fc.fc_to_ds = 0;
    af->af_fc.fc_from_ds = 0;
    af->af_fc.fc_more_frag = 0;
    af->af_fc.fc_retry = 0;
    af->af_fc.fc_pwr_mgt = 0;
    af->af_fc.fc_more_data = 0;
    af->af_fc.fc_wep = 0;
    af->af_fc.fc_order = 0;

    STORE4BYTE(&dst, (af->af_ra));

    /* store ack tx time */
    ch->txtime() = txtime(ch->size(), basicRate_);

    /* calculate ack duration */
    af->af_duration = 0;

    pktCTRL_[pri] = p;
}


/*
 *  Build the MAC header for the data packet.
 */
void Mac802_11e::sendDATA(int pri, Packet * p)
{

    hdr_cmn *ch = HDR_CMN(p);
```

```
    struct hdr_mac802_11e *dh = HDR_MAC802_11E(p);
    assert(pktTx_[pri] == 0);
    /*
     * Update the MAC header
     */
    ch->size() += phymib_.getHdrLen11();

    dh->dh_fc.fc_protocol_version = MAC_ProtocolVersion;
    dh->dh_fc.fc_type = MAC_Type_Data;
    dh->dh_fc.fc_subtype = MAC_Subtype_Data;

    dh->dh_fc.fc_to_ds = 0;
    dh->dh_fc.fc_from_ds = 0;
    dh->dh_fc.fc_more_frag = 0;
    dh->dh_fc.fc_retry = 0;
    dh->dh_fc.fc_pwr_mgt = 0;
    dh->dh_fc.fc_more_data = 0;
    dh->dh_fc.fc_wep = 0;
    dh->dh_fc.fc_order = 0;

    /*
     *  If not a broadcast packet include the duration field in the packet.
     *  The duration is the amount of time the communications needs after
     *  the data packet has been transmitted.
     */
    if ((u_int32_t) ETHER_ADDR(dh->dh_da) != MAC_BROADCAST) {
        /* store data tx time for unicast packets */
        ch->txtime() = txtime(ch->size(), dataRate_);

        dh->dh_duration = usec(txtime(phymib_.getACKlen(), basicRate_)
                               + sifs_);
    } else {
        /* store data tx time for broadcast packets (see 9.6) */
        ch->txtime() = txtime(ch->size(), basicRate_);

        dh->dh_duration = 0;
    }
    // Assign pktTx_ the create data packet and return control to send.
    pktTx_[pri] = p;
}

/* ======================================================================
   Retransmission Routines
   ====================================================================== */

/*
 *  Called if a CTS is not received after a RTS has been sent
 */
void Mac802_11e::RetransmitRTS(int pri)
{

    assert(pktTx_[pri]);
    assert(pktRTS_[pri]);
    assert(mhBackoff_.backoff(pri) == 0);

    macmib_.RTSFailureCount++;  // record failed RTS
    ssrc_[pri] += 1;            // STA Short Retry Count

    if (ssrc_[pri] >= macmib_.ShortRetryLimit) {
```

```
        discard(pktRTS_[pri], DROP_MAC_RETRY_COUNT_EXCEEDED);
        pktRTS_[pri] = 0;
        /* tell the callback the send operation failed
           before discarding the packet */
        hdr_cmn *ch = HDR_CMN(pktTx_[pri]);
        if (ch->xmit_failure_) {
            /*
             *  Need to remove the MAC header so that
             *  re-cycled packets don't keep getting
             *  bigger.
             */
            ch->size() -= phymib_.getHdrLen11();
            ch->xmit_reason_ = XMIT_REASON_RTS;
            ch->xmit_failure_(pktTx_[pri]->copy(), ch->xmit_failure_data_);
        }
        if (!modified_) {
            rst_cw(pri);
        }
        // discard the data packet
        discard(pktTx_[pri], DROP_MAC_RETRY_COUNT_EXCEEDED);
        pktTx_[pri] = 0;
        ssrc_[pri] = 0;

    } else {
        // if retry limit is not reached increment retry field
        // and increase the contention window.
        struct rts_frame *rf;
        rf = (struct rts_frame *) pktRTS_[pri]->access(hdr_mac::offset_);
        rf->rf_fc.fc_retry = 1;
        // added so the CW doesn't increase for modfied sim
        if (!modified_) {
            fprintf(stderr,
                    "increasing the CW in RetransmitRTS should not reach this point\n");
            inc_cw(LEVEL(pktTx_[pri]));
        }
        if (modified_ == SLIDING_CW) {
            mhBackoff_.start(LEVEL(pktTx_[pri]), get_backoff(pri),
                             is_idle());
        } else {
            mhBackoff_.start(LEVEL(pktTx_[pri]), getCW(pri), is_idle());
        }
    }
}


/*
 *  Called if an ACK is not received in response to a data packet.
 */
// does post back off for broadcast
void Mac802_11e::RetransmitDATA(int pri)
{
//    fprintf(stderr,"in RetransmitDATA function\n");
    struct hdr_cmn *ch;
    struct hdr_mac802_11e *mh;
    u_int32_t *rcount, *thresh;

    assert(mhBackoff_.backoff(pri) == 0);
    assert(pktTx_[pri]);
    assert(pktRTS_[pri] == 0);
```

```
ch = HDR_CMN(pktTx_[pri]);
mh = HDR_MAC802_11E(pktTx_[pri]);

/*
 *  Broadcast packets don't get ACKed and therefore
 *  are never retransmitted.
 */
if ((u_int32_t) ETHER_ADDR(mh->dh_da) == MAC_BROADCAST) {
    /*
     * Backoff at end of TX.
     */
    if (!cfb_ || rx_state_ != MAC_IDLE) {
        if (!modified_) {
            rst_cw(pri);
        }
        if (modified_ == SLIDING_CW) {
            mhBackoff_.start(pri, get_backoff(pri), is_idle());
        } else {
            mhBackoff_.start(pri, getCW(pri), is_idle());
        }
        Packet::free(pktTx_[pri]);
        pktTx_[pri] = 0;
        return;
    } else {
        // if this is the first packet in cfb, we must take its
        // duration into account, too.
        if (cfb_dur == 0) {
            cfb_dur = txtime(pktTx_[pri]) + sifs_;
        }
        assert(pktTx_[pri]);
        Packet::free(pktTx_[pri]);
        pktTx_[pri] = 0;
        cfb(pri);
        return;             // no more work to do for a broadcast packet
    }
} else if (cfb_) {
    cfb_dur = 0;
}
macmib_.ACKFailureCount++;
rtx_[pri] = 1;
if ((u_int32_t) ch->size() <= macmib_.RTSThreshold) {
    rcount = &ssrc_[pri];
    thresh = &macmib_.ShortRetryLimit;
} else {
    rcount = &slrc_[pri];
    thresh = &macmib_.LongRetryLimit;
}

(*rcount)++;

if (*rcount > *thresh) {
    numbytes_[pri] -= ch->size() - phymib_.getHdrLen11();
    rtx_[pri] = 0;
    macmib_.FailedCount++;
    /* tell the callback the send operation failed
       before discarding the packet */
    hdr_cmn *ch = HDR_CMN(pktTx_[pri]);
    if (ch->xmit_failure_) {
```

```
            ch->size() -= phymib_.getHdrLen11();
            ch->xmit_reason_ = XMIT_REASON_ACK;
            ch->xmit_failure_(pktTx_[pri]->copy(), ch->xmit_failure_data_);
        }
        if (!modified_) {
            rst_cw(pri);
        }
        discard(pktTx_[pri], DROP_MAC_RETRY_COUNT_EXCEEDED);
        pktTx_[pri] = 0;
        *rcount = 0;
    } else {
        // Prepare the data packet for retransmission.
        struct hdr_mac802_11e *dh;
        dh = HDR_MAC802_11E(pktTx_[pri]);
        dh->dh_fc.fc_retry = 1;

        sendRTS(pri, ETHER_ADDR(mh->dh_da));
        if (!modified_) {
            inc_cw(LEVEL(pktTx_[pri]));
        }
        if (modified_ == SLIDING_CW) {
            mhBackoff_.start(pri, get_backoff(pri), is_idle());
        } else {
            mhBackoff_.start(pri, getCW(pri), is_idle());
        }
    }
}

/* ======================================================================
   Incoming Packet Routines
   ====================================================================== */
void Mac802_11e::send(Packet * p, Handler * h)
{

    int pri = LEVEL(p);
    start_handle_[pri] = Scheduler::instance().clock();
    double rTime;
    struct hdr_mac802_11e *dh = HDR_MAC802_11E(p);

    /*
     * drop the packet if the node is in sleep mode
     XXX sleep mode can't stop node from sending packets
     */
    EnergyModel *em = netif_->node()->energy_model();
    // Drop the packet if in sleep mode.
    if (em && em->sleep()) {
        em->set_node_sleep(0);
        em->set_node_state(EnergyModel::INROUTE);
    }
    // Set the callback to the handler passed to send.
    callback_[pri] = h;
    // framing and calculation of  tx Duration
    // builds the mac headers for data and RTS packet
    sendDATA(pri, p);
    //check whether size exceeds RTSthreshold
    sendRTS(pri, ETHER_ADDR(dh->dh_da));

    // Assign the data packet a sequence number.
    dh->dh_scontrol = sta_seqno_++;
```

```
    //  If the medium is IDLE, we must wait for a DIFS
    //  Space before transmitting.
    assert(mhDefer_.defer(pri) == 0);

    //Mac can be still  in post-backof
    if (mhBackoff_.backoff(pri) == 0) {
        if (is_idle()) {
            // If we are already deferring, there is no
            // need to reset the Defer timer
            if (mhDefer_.defer(pri) == 0) {
                double slot = phymib_.getSlotTime();

                rTime = ((Random::random() % getCW(LEVEL(p))) * slot);
                mhDefer_.start(LEVEL(p), getAIFS(LEVEL(p)));

            }
        }
        // If the medium is NOT IDLE, then we start
        // the backoff timer.
        else {
            if (modified_ == SLIDING_CW) {
                mhBackoff_.start(LEVEL(p), get_backoff(LEVEL(p)),
                                 is_idle());
            } else {
                mhBackoff_.start(LEVEL(p), getCW(LEVEL(p)), is_idle());
            }
        }
    }

}


/*
 *  recv is called whenever a packet is received
 *  from an upper or lower layer.
 */
void Mac802_11e::recv(Packet * p, Handler * h)
{
    struct hdr_cmn *hdr = HDR_CMN(p);

    //  Handle outgoing packets.
    if (hdr->direction() == hdr_cmn::DOWN) {
        send(p, h);
        return;
    }
    /*
     *  If the interface is currently in transmit mode, then
     *  it probably won't even see this packet.  However, the
     *  "air" around me is BUSY so I need to let the packet
     *  proceed.  Just set the error flag in the common header
     *  to that the packet gets thrown away.
     */
    if (tx_active_ && hdr->error() == 0) {
        hdr->error() = 1;
    }

    if (rx_state_ == MAC_IDLE) {
        // set rx state to receive
        set_rx_state(MAC_RECV);
```

```
        // assign incoming packet to pktRx
        pktRx_ = p;

        // Schedule the reception of this packet, in
        // txtime seconds.
        mhRecv_.start(txtime(p));
    } else {
        /*
         *  If the power of the incoming packet is smaller than the
         *  power of the packet currently being received by at least
         *  the capture threshold, then we ignore the new packet.
         */
        if (pktRx_->txinfo_.RxPr / p->txinfo_.RxPr >= p->txinfo_.CPThresh) {
            // ignore the new packet
            capture(p);
        } else {
            // drop the incoming packet
            collision(p);
        }
    }
}


/*
 *  Receive timer handler called when mhRecv_ expires
 *  (indirectly through the RecvHandler).  The receive timer
 *  expiring means that the packet has been succesfully received
 *  and the can be acted upon.
 */
void Mac802_11e::recv_timer()
{
    u_int32_t src;              // source
    hdr_cmn *ch = HDR_CMN(pktRx_);      // common header
    hdr_mac802_11e *mh = HDR_MAC802_11E(pktRx_);        // MAC header
    u_int32_t dst = ETHER_ADDR(mh->dh_da);      // destiation address

    // frame control type
    u_int8_t type = mh->dh_fc.fc_type;
    // frame control subtype (e.g. ACK, RTS)
    u_int8_t subtype = mh->dh_fc.fc_subtype;

    assert(pktRx_);
    assert(rx_state_ == MAC_RECV || rx_state_ == MAC_COLL);

    /*
     *  If the interface is in TRANSMIT mode when this packet
     *  "arrives", then I would never have seen it and should
     *  do a silent discard without adjusting the NAV.
     */
    if (tx_active_) {
        Packet::free(pktRx_);
        goto done;
    }

    /*
     * Check the rx state to determine if a collision occured.
     * Handle collisions.
     */
    if (rx_state_ == MAC_COLL) {
```

```
    discard(pktRx_, DROP_MAC_COLLISION);
    set_nav(usec(eifs_));
    eifs_nav_ = eifs_;
    goto done;
}

/*
 * Check to see if this packet was received with enough
 * bit errors that the current level of FEC still could not
 * fix all of the problems - ie; after FEC, the checksum still
 * failed.
 */
if (ch->error()) {
    Packet::free(pktRx_);
    set_nav(usec(eifs_));
    eifs_nav_ = eifs_;
    goto done;
}

/*
 * avoid Nav-reset bug:
 * if received packet has no  errors and had no collision, but nav
 * was set due to an earlier collision, nav has to be reset!
 */
if (mhNav_.busy())
    reset_eifs_nav();

/*
 * IEEE 802.11 specs, section 9.2.5.6
 *  - update the NAV (Network Allocation Vector)
 */
if (dst != (u_int32_t) index_) {
    set_nav(mh->dh_duration);
}

/* tap out - */
if (tap_ && type == MAC_Type_Data && MAC_Subtype_Data == subtype)
    tap_->tap(pktRx_);
/*
 * Adaptive Fidelity Algorithm Support - neighborhood infomation
 * collection
 *
 * Hacking: Before filter the packet, log the neighbor node
 * I can hear the packet, the src is my neighbor
 */
if (netif_->node()->energy_model() &&
    netif_->node()->energy_model()->adaptivefidelity()) {
    src = ETHER_ADDR(mh->dh_sa);
    netif_->node()->energy_model()->add_neighbor(src);
}
/*
 * Address Filtering, where all packets not destined to current
 * node are discarded.
 */
if (dst != (u_int32_t) index_ && dst != MAC_BROADCAST) {
    /*
     *  We don't want to log this event, so we just free
     *  the packet instead of calling the drop routine.
     */
```

```
        discard(pktRx_, "---");
        goto done;
    }
    // Decide what to do based on the packet that was received.
    switch (type) {

    case MAC_Type_Management:
        // if the packet is a management packet drop it.
        discard(pktRx_, DROP_MAC_PACKET_ERROR);
        goto done;
        break;

    case MAC_Type_Control:
        switch (subtype) {
        case MAC_Subtype_RTS:
            recvRTS(pktRx_);
            break;
        case MAC_Subtype_CTS:
            recvCTS(pktRx_);
            break;
        case MAC_Subtype_ACK:
            recvACK(pktRx_);
            break;
        default:
            fprintf(stderr,
                    "recvTimer1:Invalid MAC Control Subtype %x\n",
                    subtype);
            exit(1);
        }
        break;
    case MAC_Type_Data:
        switch (subtype) {
        case MAC_Subtype_Data:
            recvDATA(pktRx_);
            break;
        default:
            fprintf(stderr,
                    "recv_timer2:Invalid MAC Data Subtype %x\n", subtype);
            exit(1);
        }
        break;
    default:
        fprintf(stderr, "recv_timer:Invalid MAC Type %x\n", subtype);
        exit(1);
    }
  done:
    pktRx_ = 0;
    rx_resume();
}


/*
 *  Called by the recv_timer after a full RTS has been received.
 */
void Mac802_11e::recvRTS(Packet * p)
{

    int pri = LEVEL(p);
    struct rts_frame *rf =
```

```
        (struct rts_frame *) p->access(hdr_mac::offset_);
    // If TX state is not idle, the packet wouldn't have been
    // heard so discard the packet.
    if (tx_state_[pri] != MAC_IDLE) {
        discard(p, DROP_MAC_BUSY);
        return;
    }
    //  If I'm responding to someone else, discard this RTS.
    if (pktCTRL_[pri]) {
        discard(p, DROP_MAC_BUSY);
        return;
    }
    // prepare to send a CTS packet
    sendCTS(pri, ETHER_ADDR(rf->rf_ta), rf->rf_duration);
    // Restart the defer timer.
    tx_resume();
    mac_log(p);
}

/*
 *  txtime() - pluck the precomputed tx time
 *  from the packet header
 */
double Mac802_11e::txtime(Packet * p)
{
    struct hdr_cmn *ch = HDR_CMN(p);
    double t = ch->txtime();
    if (t < 0.0) {
        drop(p, "XXX");
        exit(1);
    }
    return t;
}


/*
 * txtime() - calculate tx time for packet of size
 * "psz" bytes at rate "drt" bps
 */
double Mac802_11e::txtime(double psz, double drt)
{
    double dsz = psz - phymib_.getPLCPhdrLen();
    int plcp_hdr = phymib_.getPLCPhdrLen() << 3;
    int datalen = (int) dsz << 3;

    double t =
        (((double) plcp_hdr) / phymib_.getPLCPDataRate()) +
        (((double) datalen) / drt);
    return (t);
}



void Mac802_11e::recvCTS(Packet * p)
{

    int pri = LEVEL(p);
    // If an invalid state exists drop the packet.
    if (tx_state_[pri] != MAC_RTS) {
```

```
        discard(p, DROP_MAC_INVALID_STATE);
        return;
    }
    assert(pktRTS_[pri]);
    Packet::free(pktRTS_[pri]);
    pktRTS_[pri] = 0;

    assert(pktTx_[pri]);
    mhSend_.stop();             // stop the send timer
    tx_resume();                // set the defer timer
    mac_log(p);
}


/*
 *  Called by the recv_timer after a full data packet is received.
 */
void Mac802_11e::recvDATA(Packet * p)
{

    int pri = LEVEL(p);         // Get the priority
    // get the MAC header
    struct hdr_mac802_11e *dh = HDR_MAC802_11E(p);
    u_int32_t dst, src, size;

    {
        struct hdr_cmn *ch = HDR_CMN(p);        // common header
        dst = ETHER_ADDR(dh->dh_da);    // destination address
        src = ETHER_ADDR(dh->dh_sa);    // source address
        size = ch->size();      // size

        // Adjust the MAC packet size - ie; strip
        // off the mac header
        ch->size() -= phymib_.getHdrLen11();
        ch->num_forwards() += 1;
    }

    // if the modified algorith is used update the table
    if (modified_ && dst == MAC_BROADCAST) {
        reception_table_->update(src, dh->dh_scontrol,
                                 Scheduler::instance().clock());
    }
    // If we sent a CTS, clean up...
    if (dst != MAC_BROADCAST) {
        if (size >= macmib_.RTSThreshold) {
            if (tx_state_[pri] == MAC_CTS) {
                assert(pktCTRL_[pri]);
                Packet::free(pktCTRL_[pri]);
                pktCTRL_[pri] = 0;
                mhSend_.stop();
                /*
                 * Our CTS got through.
                 */
                printf("(%d): RECVING DATA!\n", index_);
            } else {
                discard(p, DROP_MAC_BUSY);
                printf("(%d)..discard DATA\n", index_);
                return;
            }
```

```
            sendACK(pri, src);
            tx_resume();
        }
        //  We did not send a CTS and there's no
        //  room to buffer an ACK.
        else {
            if (pktCTRL_[pri]) {
                discard(p, DROP_MAC_BUSY);
                return;
            }

            sendACK(pri, src);
            if (mhSend_.busy() == 0) {
                tx_resume();
            }
        }
    }

    /* ============================================================
       Make/update an entry in our sequence number cache.
       ============================================================ */

    /* Changed by Debojyoti Dutta. This upper loop of if{}else was
       suggested by Joerg Diederich <dieder@ibr.cs.tu-bs.de>.
       Changed on 19th Oct'2000 */

    if (dst != MAC_BROADCAST) {
        if (src < (u_int32_t) cache_node_count_) {
            Host *h = &cache_[src];

            if (h->seqno && h->seqno == dh->dh_scontrol) {
                discard(p, DROP_MAC_DUPLICATE);
                return;
            }
            h->seqno = dh->dh_scontrol;

        } else {
            static int count = 0;
            if (++count <= 10) {
                if (count == 10)
                    printf
                        ("[suppressing additional MAC cache_ warnings]\n");
            }
        }
    }

    /*

     *  Pass the packet up to the link-layer.
     *  XXX - we could schedule an event to account
     *  for this processing delay.
     */

    // XXXXX NOTE: use of incoming flag has been depracated;
    // In order to track direction of pkt flow, direction_
    // in hdr_cmn is used instead. see packet.h for details.
    uptarget_->recv(p, (Handler *) 0);
}
```

```
void Mac802_11e::recvACK(Packet * p)
{

    int pri = LEVEL(p);
    struct hdr_cmn *ch = HDR_CMN(p);
    if (tx_state_[pri] != MAC_SEND) {
        discard(p, DROP_MAC_INVALID_STATE);
        return;
    }

    mhSend_.stop();

    /*
     * The successful reception of this ACK packet implies
     * that our DATA transmission was successful.  Hence,
     * we can reset the Short/Long Retry Count and the CW.
     */
    if ((u_int32_t) ch->size() <= macmib_.RTSThreshold) {
        ssrc_[pri] = 0;
    } else {
        slrc_[pri] = 0;
    }
    if (rtx_[pri]) {
        rtx_[pri] = 0;
    }

    sending = 0;
    check_backoff_timer();
    /*
     * Backoff before sending again.
     */
    // need to fix this
    fprintf(stderr, "In receivedACK shouldn't reach here ch->size() %u\n",
            (unsigned int) ch->size());
    if (!cfb_ || (unsigned int) ch->size() > macmib_.RTSThreshold) {
        assert(mhBackoff_.backoff(pri) == 0);
        if (!modified_) {
            rst_cw(pri);
        }
        if (modified_ == SLIDING_CW) {
            mhBackoff_.start(pri, get_backoff(pri), is_idle());
        } else {
            mhBackoff_.start(pri, getCW(pri), is_idle());
        }
        assert(pktTx_[pri]);
        Packet::free(pktTx_[pri]);
        pktTx_[pri] = 0;
        tx_resume();

    } else {
        // if this is the first packet in cfb, we must take its
        // duration into account, too.
        if (cfb_dur == 0) {
            cfb_dur = txtime(pktTx_[pri])
                + sifs_ + txtime(phymib_.getACKlen(), basicRate_);
        }
        pktRx_ = 0;
        rx_resume();
```

```
        assert(pktTx_[pri]);
        Packet::free(pktTx_[pri]);
        pktTx_[pri] = 0;
        cfb(pri);
    }
    mac_log(p);
}

void Mac802_11e::cfb(int pri)
{
    fprintf(stderr, "*********** shouldn't be in CFB\n");
    double timeout;
    struct hdr_mac802_11e *mh;

    // next packet out of queue
    if (queue_->pri_[pri].getLen() > 0) {
        Packet *p = queue_->pri_[pri].deque();
        // framing
        sendDATA(pri, p);
        //      hdr_cmn *ch = HDR_CMN(pktTx_[pri]);
        mh = HDR_MAC802_11E(pktTx_[pri]);

        if ((u_int32_t) ETHER_ADDR(mh->dh_da) != MAC_BROADCAST) {
            cfb_dur += sifs_ + txtime(pktTx_[pri])
                + sifs_ + txtime(phymib_.getACKlen(), basicRate_);
            cfb_broadcast = 0;
        } else {
            cfb_dur += sifs_ + txtime(pktTx_[pri]);
            cfb_broadcast = 1;
        }
    } else {
        cfb_dur = txop_limit_[pri] + 1;
    }
    if (cfb_dur <= txop_limit_[pri]) {
        // send
        if ((u_int32_t) ETHER_ADDR(mh->dh_da) != MAC_BROADCAST) {
            timeout = txtime(pktTx_[pri])
                + DSSS_EDCA_MaxPropagationDelay
                + sifs_ + txtime(phymib_.getACKlen(), basicRate_)
                + DSSS_EDCA_MaxPropagationDelay;

        } else {
            timeout = txtime(pktTx_[pri]);
        }
        cfb_active = 1;
        mhSifs_.start(pri, sifs_);
    } else {
        cfb_dur = 0;
        cfb_broadcast = 0;
        assert(mhBackoff_.backoff(pri) == 0);
        if (!modified_) {
            rst_cw(pri);
        }
        if (modified_ == SLIDING_CW) {
            mhBackoff_.start(pri, get_backoff(pri), is_idle());
        } else {
            mhBackoff_.start(pri, getCW(pri), is_idle());
        }
        tx_resume();
```

```
    }
}

// request parameters for each priority from the corresponding queues
double Mac802_11e::getAIFS(int pri)
{
    // if AIFS is not set, set it
    if (!AIFSset) {
        levels = queue_->getLevels();
        for (int i = 0; i < levels; i++) {
            slotnum = queue_->pri_[i].getAIFS();
            aifs_[i] = sifs_ + (slotnum * phymib_.getSlotTime());
            txop_limit_[i] = queue_->pri_[i].getTXOPLimit();
        }
        AIFSset = 1;
    }
    return aifs_[pri];
}


int Mac802_11e::getCW(int level)
{
    // if the contention windows are not set set them
    if (!CWset) {
        levels = queue_->getLevels();
        for (int i = 0; i < levels; i++) {
            cw_[i] = queue_->pri_[i].getCW_MIN();
            cwmin_[i] = queue_->pri_[i].getCW_MIN();
            cwmax_[i] = queue_->pri_[i].getCW_MAX();
            // if the modified algorithm is used set
            // the Sliding factor and the cw upper bound.
            if (modified_ == SLIDING_CW) {
                SF_[i] = queue_->pri_[i].getPF();
                cw_UB_[i] = SF_[i] * 2;
            }
        }
        CWset = 1;
    }
    return cw_[level];
}

void Mac802_11e::setQ(PriQ * priqueue)
{
    queue_ = priqueue;
}

inline void Mac802_11e::reset_eifs_nav()
{
    if (eifs_nav_ > 0) {
        double now = Scheduler::instance().clock();

        assert(nav_ > now);
        assert(mhNav_.busy());

        mhNav_.stop();
        nav_ -= eifs_nav_;
        eifs_nav_ = 0.0;
        if (nav_ > now) {
            mhNav_.start(nav_ - now);
```

```
        } else {
            nav_ = now;
            check_backoff_timer();
        }
    }
}


bool Mac802_11e::inc_retryCounter(int pri)
{
    u_int32_t *rcount, *thresh;
    struct hdr_cmn *ch = HDR_CMN(pktTx_[pri]);
    if ((u_int32_t) ch->size() <= macmib_.RTSThreshold) {
        ssrc_[pri]++;
        rcount = &ssrc_[pri];
        thresh = &macmib_.ShortRetryLimit;
    } else {
        slrc_[pri]++;
        rcount = &slrc_[pri];
        thresh = &macmib_.LongRetryLimit;
    }
    if (*rcount > *thresh) {
        rtx_[pri] = 0;
        macmib_.FailedCount++;
        if (!modified_) {
            rst_cw(pri);
        }
        discard(pktTx_[pri], DROP_MAC_RETRY_COUNT_EXCEEDED);
        pktTx_[pri] = 0;
        *rcount = 0;
        return 1;
    } else {
        return 0;
    }
}
void Mac802_11e::updateTableHandler()
{
    double avg =
        reception_table_->avg_reception_rate(Scheduler::instance().
                                             clock());
    double prev_avg = previous_recp_rate_;
    double diff_avg = avg - prev_avg;
    previous_recp_rate_ = avg;

    // adjust the windows based on the difference in avg
    if (diff_avg >= sliding_threshold_) {
        decrease_window();
    } else if (-diff_avg >= sliding_threshold_) {
        increase_window();
    }

    // if the the threashold is not exceded maintain the current window
}

/*
 *  increase the size of the CW for the MODIFIED_CW
 */
void Mac802_11e::increase_window()
{
```

```
    for (int level = 0; level < MAX_PRI; level++) {
        cw_old = cw_[level];
        //calculate new cw_[pri]
        double new_window = cw_old * scaling_factor_;
        int win_size = (int) floor(new_window) + 1;
        cw_[level] = win_size;
        if (cw_[level] > cwmax_[level]) {
            cw_[level] = cwmax_[level];
        }
    }
}

/*
 *  decrease the size of the CW for the MODIFIED_CW
 */
void Mac802_11e::decrease_window()
{
    for (int level = 0; level < MAX_PRI; level++) {
        cw_old = cw_[level];
        double new_window = cw_old / scaling_factor_;
        int win_size = (int) ceil(new_window) - 1;

        //calculate new cw_[pri]
        cw_[level] = win_size;
        if (cw_[level] < cwmin_[level]) {
            cw_[level] = cwmin_[level];
        }
    }
}

/*
 *  get_backoff returns a backoff value to be used for the class based on
 *  the sliding window algorithm.  A value is randomly selected between
 *  the cw lower bound and the cw upper bound of the window.
 */
int Mac802_11e::get_backoff(int prio)
{
    if (!CWset) {
        getCW(prio);            // call to initialize the CW
    }

    int new_backoff =
        (Random::random() % (cw_UB_[prio] - cw_[prio] + 1)) + cw_[prio];
// ********  used for debugging ***********
//      fprintf(stderr, "******random %d level %d\n", Random::random(), prio);
//      fprintf(stderr, "new CW = %d\n", new_backoff);
//      fprintf(stderr, "CW %d, cw_UB %d, SF %d\n", cw_[prio], cw_UB_[prio], SF_[prio]);
    return new_backoff;
}


/*
 *  Slide the CW up for the SLIDING_CW.
 */
void Mac802_11e::slide_window_up()
{
    // cw_[i] is the lower bound of the window
    // cw_UB_ is the upper bound of the window
    for (int i = 0; i < MAX_PRI; i++) {
```

```
        // if the maximum is not reached slide the window
        if (cw_UB_[i] + SF_[i] <= cwmax_[i]) {
            cw_[i] = cw_[i] + SF_[i];
            cw_UB_[i] = cw_UB_[i] + SF_[i];
        } else {
            // if the upper bound of the window is teached
            // set the cw_UB_[i] to the max value
            cw_[i] = cwmax_[i] - 2 * SF_[i];
            cw_UB_[i] = cwmax_[i];
        }
    }
}

/*
 *  Slide the CW down for the SLIDING_CW
 */
void Mac802_11e::slide_window_down()
{
    // cw_[i] is the lower bound of the window
    // cw_UB_ is the upper bound of the window
    for (int i = 0; i < MAX_PRI; i++) {
        // If the minimum is not exceeded slide the
        // window down.
        if (cw_[i] - SF_[i] >= cwmin_[i]) {
            cw_[i] = cw_[i] - SF_[i];
            cw_UB_[i] = cw_UB_[i] - SF_[i];
        } else {
            cw_[i] = cwmin_[i];
            cw_UB_[i] = cwmin_[i] + 2 * SF_[i];
        }
    }
}
```

E.2.3 mac-timers_802_11e.h

```
//* -*- Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 * This product includes software developed by the Computer Systems
 * Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
```

```
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */
/* Ported from CMU/Monarch's code, nov'98 -Padma.*/

#ifndef __mac_timers_802_11e_h__
#define __mac_timers_802_11e_h__

#define MAX_PRI 4
/* ======================================================================
   Timers
   ====================================================================== */
class Mac802_11e;

class MacTimer_802_11e:public Handler {
  public:
  MacTimer_802_11e(Mac802_11e * m, double s = 0):mac(m), slottime(s) {
        busy_ = paused_ = 0;
        stime = rtime = 0.0;
    } virtual void handle(Event * e) = 0;

    virtual void start(double time);
    virtual void stop(void);
    virtual void pause(void) {
        assert(0);
    }
    virtual void resume(void) {
        assert(0);
    }

    inline int busy(void) {
        return busy_;
    }
    inline int paused(void) {
        return paused_;
    }
    inline double expire(void) {
        return ((stime + rtime) - Scheduler::instance().clock());
    }

  protected:
    Mac802_11e * mac;
    int busy_;
    int paused_;
    Event intr;
    double stime;               // start time
    double rtime;               // remaining time
    double slottime;
};

// BackoffTimer_802_11e counts down the residual time
// of a backoff.
class BackoffTimer_802_11e:public MacTimer_802_11e {
```

```
  public:
    BackoffTimer_802_11e(Mac802_11e * m, double s):MacTimer_802_11e(m, s) {
        for (int pri = 0; pri < MAX_PRI; pri++) {
            AIFSwait_[pri] = 0.0;
            stime_[pri] = rtime_[pri] = 0;
            backoff_[pri] = 0;
            decremented_[pri] = 0.0;
        } levels = 0;
    }

    void start(int pri, int cw, int idle);
    void handle(Event * e);
    void pause(void);
    void resume();
    int backoff(int pri);

  private:
    inline void round_time(int pri);
    inline bool rounding(double x, double y);
    int backoff_[MAX_PRI];      //for handler
    void restart();

    // AIFSwait indicates whether a timer was paused and
    // resumed or paused and restarted due to the start or
    // handle of another backoff while others are active.
    double AIFSwait_[MAX_PRI];
    double rtime_[MAX_PRI];

    double stime_[MAX_PRI];
    double decremented_[MAX_PRI];
    int levels;                 // get number of levels out of priq.cc or mac-802_11e.cc
    bool pause_restart;
};

//  DeferTimer_802_11e is used when the MAC has to sense the
//  the medium being idle for the defer period
class DeferTimer_802_11e:public MacTimer_802_11e {
  public:
    DeferTimer_802_11e(Mac802_11e * m, double s):MacTimer_802_11e(m, s) {
        for (int i = 0; i < MAX_PRI; i++) {
            defer_[i] = 0;
            rtime_[i] = stime_[i] = 0;
        } prio = -1;
    }

    void start(int pri, double time);
    void pause(void);
    void stop(void);
    void handle(Event * e);
    int defer(int pri);
  private:
    int prio;
    int defer_[MAX_PRI];
    double rtime_[MAX_PRI];
    double stime_[MAX_PRI];
};

//  SIFSTimer is used when the MAC has to sense the medium
//  being idle for period of a SIFS.
```

```
class SIFSTimer_802_11e:public MacTimer_802_11e {
  public:
    SIFSTimer_802_11e(Mac802_11e * m, double s):MacTimer_802_11e(m, s) {
        for (int i = 0; i < MAX_PRI; i++) {
            sifs_[i] = 0;
            rtime_[i] = stime_[i] = 0;
        } prio = -1;
    }

    void start(int pri, double time);
    void handle(Event * e);
  private:
    int prio;
    int sifs_[MAX_PRI];
    double rtime_[MAX_PRI];
    double stime_[MAX_PRI];
};

//  IFTimer indicates how long the interface will be
//  in transmit mode when sending a packet.
class IFTimer_802_11e:public MacTimer_802_11e {
  public:
    IFTimer_802_11e(Mac802_11e * m):MacTimer_802_11e(m) {
    } void handle(Event * e);
};


// The NavTimer is started if:
//   - for the EIFS if collision is detected
//   - for the period contained in the duration field of
//     a succefully received frame
//   - for the duration of the RTS/CTS/DATA/ACK exchange
class NavTimer_802_11e:public MacTimer_802_11e {
  public:
    NavTimer_802_11e(Mac802_11e * m):MacTimer_802_11e(m) {
    } void handle(Event * e);
};

class RxTimer_802_11e:public MacTimer_802_11e {
  public:
    RxTimer_802_11e(Mac802_11e * m):MacTimer_802_11e(m) {
    } void handle(Event * e);
};

class TxTimer_802_11e:public MacTimer_802_11e {
  public:
    TxTimer_802_11e(Mac802_11e * m):MacTimer_802_11e(m) {
    } void handle(Event * e);
};

class UpdateTableTimer:public MacTimer_802_11e {
  public:
    UpdateTableTimer(Mac802_11e * m):MacTimer_802_11e(m) {
    }
    void handle(Event * e);
};


#endif                          /* __mac_timers_802_11e_h__ */
```

E.2.4 mac-timers_802_11e.cc

```
//* -*- Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 * This product includes software developed by the Computer Systems
 * Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

#undef NDEBUG
#include <assert.h>
#include <delay.h>
#include <connector.h>
#include <packet.h>
#include <random.h>
#include <iostream>
#define BDEBUG 0

//#include <debug.h>
#include <arp.h>
#include <ll.h>
#include <mac.h>
#include <mac/802_11e/mac-timers_802_11e.h>
#include <mac/802_11e/mac-802_11e.h>

// set to 1 for EDCA, 0 for older EDCF
#define EDCA 1
#define INF   (1e12)
#define ROUND (1e-12)
#define MU(x)    x * (1e+6)
```

```
/* ======================================================================
   Timers
   ====================================================================== */

void
 MacTimer_802_11e::start(double time)
{
    Scheduler & s = Scheduler::instance();
    assert(busy_ == 0);

    busy_ = 1;
    paused_ = 0;
    stime = s.clock();
    rtime = time;
    assert(rtime >= 0.0);
    s.schedule(this, &intr, rtime);
}



void MacTimer_802_11e::stop(void)
{
    Scheduler & s = Scheduler::instance();

    assert(busy_);

    if (paused_ == 0)
        s.cancel(&intr);

    busy_ = 0;
    paused_ = 0;
    stime = 0.0;
    rtime = 0.0;
}



/* ======================================================================
   Defer Timer
   ====================================================================== */

/*
 *  Start the defer timer
 */
void DeferTimer_802_11e::start(int pri, double time)
{
    Scheduler & s = Scheduler::instance();
    bool another = 0;
    assert(defer_[pri] == 0);
    for (int i = 0; i < MAX_PRI; i++) {
        if (defer_[i] && busy_) {
            another = 1;
        }
    }

    if (another) {
        pause();
    }
```

```
    busy_ = 1;

    defer_[pri] = 1;
    stime_[pri] = s.clock();
    rtime_[pri] = time;
    double delay = INF;
    int prio = MAX_PRI + 1;
    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (defer_[pri]) {
            stime_[pri] = s.clock();
            double delay_ = rtime_[pri];
            if ((delay_ < delay)) {
                delay = delay_;
                prio = pri;
            }
        }
    }
    if (prio < MAX_PRI + 1) {
        assert(rtime_[prio] >= 0);
        s.schedule(this, &intr, rtime_[prio]);
        paused_ = 0;
    } else {
        exit(0);

    }
}

void DeferTimer_802_11e::stop()
{
    busy_ = 0;
    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (defer_[pri])
            mac->defer_stop(pri);
        defer_[pri] = 0;
        stime_[pri] = 0;
        rtime_[pri] = 0;

    }
    Scheduler & s = Scheduler::instance();
    s.cancel(&intr);
}

void DeferTimer_802_11e::pause()
{
    Scheduler & s = Scheduler::instance();
    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (defer_[pri] && busy_) {
            double st = s.clock();      //now
            double rt = stime_[pri];    // Start-Time of defer
            double sr = st - rt;
            assert(busy_ && !paused_);
            if (rtime_[pri] - sr >= 0.0) {
                rtime_[pri] -= sr;
                assert(rtime_[pri] >= 0.0);
            } else {
                if (rtime_[pri] + ROUND - sr >= 0.0) {
                    rtime_[pri] = 0;
                } else {
                    cerr <<
```

```
                        "ERROR in DeferTimer::pause(), rtime_["
                        << pri << "]  is " << rtime_[pri] <<
                        ", sr:" << sr << "  \n";
                    exit(0);
                }
            }
        }
    }
    s.cancel(&intr);
    paused_ = 1;
}


void DeferTimer_802_11e::handle(Event *)
{
    double delay = INF;
    int prio = MAX_PRI + 1;

    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (rtime_[pri] >= 0 && defer_[pri]) {
            double delay_ = rtime_[pri];
            if ((delay_ < delay) && defer_[pri]) {
                delay = delay_;
                prio = pri;
            }
        }
    }
    if (prio < MAX_PRI + 1) {
        busy_ = 0;
        paused_ = 0;
        defer_[prio] = 0;
        stime_[prio] = 0.0;
        rtime_[prio] = 0.0;
        mac->deferHandler(prio);
    } else {
        cout << "handling ERROR in DeferTimer::handler \n";
        exit(0);
    }

    //check if there is another DeferTimer active
    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (rtime_[pri] >= 0 && defer_[pri]) {
            rtime_[pri] = 0.0;
            stime_[pri] = 0.0;
            defer_[pri] = 0;
            mac->defer_stop(pri);
        }
    }
}

int DeferTimer_802_11e::defer(int pri)
{
    return defer_[pri];
}

/* ======================================================================
   NAV Timer
   ====================================================================== */
void NavTimer_802_11e::handle(Event *)
```

```
{
    busy_ = 0;
    paused_ = 0;
    stime = 0.0;
    rtime = 0.0;

    mac->navHandler();
}


/* ======================================================================
   Receive Timer
   ====================================================================== */
void RxTimer_802_11e::handle(Event *)
{
    busy_ = 0;
    paused_ = 0;
    stime = 0.0;
    rtime = 0.0;

    mac->recvHandler();
}


/* ======================================================================
   Send Timer
   ====================================================================== */
void TxTimer_802_11e::handle(Event *)
{
    busy_ = 0;
    paused_ = 0;
    stime = 0.0;
    rtime = 0.0;

    mac->sendHandler();
}


/* ======================================================================
   Interface Timer
   ====================================================================== */
void IFTimer_802_11e::handle(Event *)
{
    busy_ = 0;
    paused_ = 0;
    stime = 0.0;
    rtime = 0.0;

    mac->txHandler();
}

/* ======================================================================
 Defer Timer for SIFS
   ====================================================================== */
void SIFSTimer_802_11e::start(int pri, double time)
{

    Scheduler & s = Scheduler::instance();
    if (busy_ == 1) {
```

```
        cerr << "Mac " << mac->index_ << ", ERROR in SIFSTimer!";
        exit(0);
    }
    assert(sifs_[pri] == 0);

    busy_ = 1;

    sifs_[pri] = 1;
    stime_[pri] = s.clock();
    rtime_[pri] = time;
    s.schedule(this, &intr, rtime_[pri]);
}

void SIFSTimer_802_11e::handle(Event *)
{
    busy_ = 0;
    prio = 0;

    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (sifs_[pri]) {
            prio = pri;
            break;
        }
    }

    for (int i = 0; i < MAX_PRI; i++) {
        sifs_[i] = 0;
        stime_[i] = 0.0;
        rtime_[i] = 0.0;
    }
    mac->deferHandler(prio);
}

/* ======================================================================
   Backoff Timer
   ====================================================================== */

/*
 * round to the next slot,
 * this is needed if a station initiates a transmission
 * and has not been synchronized by the end of last frame on channel
 *
 * round_time rounds the residule time, rtime, to a multiple
 * of a slot time,
 */

inline void BackoffTimer_802_11e::round_time(int pri)
{
    double rmd = remainder(rtime_[pri], slottime);
    if (rmd + ROUND < 0) {
        exit(1);
    }
    if (rmd > ROUND) {
        rtime_[pri] = ceil(rtime_[pri] / slottime) * slottime;
    } else {
        rtime_[pri] = floor(rtime_[pri] / slottime) * slottime;
    }
}
```

```
// handle is called at the expiration of the timer.
// the method determines which priority invoked
// the timer expiration.
void BackoffTimer_802_11e::handle(Event *)
{
    Scheduler & s = Scheduler::instance();
    paused_ = 0;
    double delay = INF;
    int prio = MAX_PRI + 1;

    // determine the smallest delay
    for (int pri = 0; pri < MAX_PRI; pri++) {
        double delay_ = rtime_[pri] + AIFSwait_[pri];
        if ((delay_ < delay) && backoff_[pri]
            && !rounding(delay, delay_)) {
            delay = delay_;
            prio = pri;
        }
    }
    // check for errors
    if (!EDCA
        && !rounding(s.clock(),
                     stime_[prio] + rtime_[prio] + AIFSwait_[prio])) {
        exit(1);
    }
    // reset the values of the priority with the smallest delay
    if (prio < MAX_PRI + 1) {
        busy_ = 0;
        stime_[prio] = 0;
        rtime_[prio] = 0;
        backoff_[prio] = 0;
        AIFSwait_[prio] = 0.0;
        decremented_[prio] = 0;
    } else {
        cerr << "ERROR: wrong priority in BackoffTimer::handler()";
        exit(0);
    }

    busy_ = 0;
    bool another = 0;

    //check if there is another backoff process
    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (backoff_[pri]) {
            if (rounding(rtime_[pri] + AIFSwait_[pri], delay)) {
                // check if other priority is not a postbackoff,
                // i.e., packet to tx is available
                if (mac->pktTx_[pri]) {
                    // if not using the modified algorithm increase
                    // the CW.
                    if (!mac->modified_) {
                        mac->inc_cw(pri);
                    }
                    if (EDCA) {
                        if (mac->inc_retryCounter(pri)) {
                            // maximum retry limit exceeded,
                            // deletion of packet done in inc_retryCounter(),
                            // now reset Backoff variables
                            stime_[pri] = 0;
```

```
                            rtime_[pri] = 0;
                            backoff_[pri] = 0;
                            AIFSwait_[pri] = 0.0;
                        } else {
                            // if virtual collision has occured and the retry
                            // limit is not reached restart the backoff timer
                            AIFSwait_[pri] = 0;
                            stime_[pri] = s.clock();    //Start-Time
// may need to change here
                            if (mac->modified_ == SLIDING_CW) {
                                rtime_[pri] =
                                    mac->get_backoff(pri) * slottime;
                            } else {
                                rtime_[pri] =
                                    (Random::random() %
                                     (mac->getCW(pri) + 1)) * slottime;
                            }
                            backoff_[pri] = 1;
                            another = 1;
                        }
                    } else {
                        // if not using EDCA
                        // if virtual collision has occured and the retry
                        // limit is not reached restart the backoff timer

                        AIFSwait_[pri] = 0;
                        stime_[pri] = s.clock();        //Start-Time
// may need to change
                        if (mac->modified_ == SLIDING_CW) {
                            rtime_[pri] = mac->get_backoff(pri) * slottime;
                            fprintf(stderr,
                  "backoffhadler handle virtual collision !EDCA rtime reset to %f\n",
                   rtime_[pri]);
                        } else {
                            rtime_[pri] =
                                (Random::random() %
                                 (mac->getCW(pri) + 1)) * slottime;
                        }
                        backoff_[pri] = 1;
                        another = 1;
                    }
                } else {
                    // end of postbackoff: reset stime_, rtime_, ...
                    stime_[pri] = 0;
                    rtime_[pri] = 0;
                    backoff_[pri] = 0;
                    AIFSwait_[pri] = 0.0;
                }
            } else {
                another = 1;
            }
        }
    }
    if (another) {
        busy_ = 1;
    }
    if (busy_ && !paused_) {
        //pause + restart because the finished
        //backoff could have been a postbackoff!
```

```
        pause();
        restart();
    }
    mac->backoffHandler(prio);
}


// start schedules the backoff
void BackoffTimer_802_11e::start(int pri, int cw, int idle)
{
    Scheduler & s = Scheduler::instance();

    if (busy_) {                //already a backoff running!
        stime_[pri] = s.clock();        //Start-Time
        if (mac->modified_ == SLIDING_CW) {
            rtime_[pri] = cw * slottime;
//          fprintf(stderr, "BackoffTimer::start a backoff already running CW = %d\n", cw);
        } else {
            rtime_[pri] = (Random::random() % (cw + 1)) * slottime;
        }
        AIFSwait_[pri] = 0.0;
        decremented_[pri] = 0;
        backoff_[pri] = 1;

        if (idle == 0) {
            if (!paused_) {
                pause();
            }
        } else {
            assert(rtime_[pri] >= 0.0);
            if (!paused_) {
                pause();
                AIFSwait_[pri] = mac->getAIFS(pri);
                restart();
            }
        }
    } else {                    // no other backoffs
        backoff_[pri] = 1;      // active flag for the priority
        busy_ = 1;              // active timer flag
        stime_[pri] = s.clock();        //Start-Time
        if (mac->modified_ == SLIDING_CW) {
            rtime_[pri] = cw * slottime;
        } else {
            rtime_[pri] = (Random::random() % (cw + 1)) * slottime;
        }
        AIFSwait_[pri] = mac->getAIFS(pri);

        if (idle == 0) {
            paused_ = 1;
        } else {
            assert(rtime_[pri] >= 0.0);
            s.schedule(this, &intr, rtime_[pri] + AIFSwait_[pri]);
        }
    }
}

inline bool BackoffTimer_802_11e::rounding(double x, double y)
{
    /*
```

```
     * check whether x is within y +/- 1e-12
     */
    if (x > y && x < y + ROUND)
        return 1;
    else if (x < y && x > y - ROUND)
        return 1;
    else if (x == y)
        return 1;
    else
        return 0;
}


// pause temporarily stops the backoff timer.
// Called when the medium changes from idle to busy
// or when a backoff is started while there are other
// active backoffs.
void BackoffTimer_802_11e::pause()
{
    Scheduler & s = Scheduler::instance();

    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (backoff_[pri]
            && !rounding(stime_[pri], s.clock())
            && !rounding(decremented_[pri], s.clock())) {
            double st = s.clock();      //now
            double rt = stime_[pri] + AIFSwait_[pri];
            // start-Time of backoff + waiting time
            double sr = st - rt;
            // is < 0,  if stime is back for less than AIFSwait, thus a slot
            // decrement must not appear

            int slots = 0;

            if (rounding(sr, 0.0)) {    // now - stime is equal to AIFSWAIT
                // include EDCA rule: slot decrement due to full AIFS period
                if (EDCA && (rtime_[pri] > slottime)) {
                    rtime_[pri] -= slottime;
                    /* P802.11e/D13.0
                     * Section 9.9.1.3:
                     * one and only one of the following actions is allowed at
                     * slot boundaries:
                     * - decrement of backoff timer, or
                     * - initiate transmission, or
                     * - backoff procedure due to internal collision, or
                     * - do nothing.
                     * Thus we need to assure that a station does not
                     * initiate a transmission
                     * immediately after decreasing rtime.The check is
                     * done in restart()/resume()
                     * by comparing the time of decrement and simulated time.
                     */
                    decremented_[pri] = st;     // time of decrement
                } else if (!rounding(stime_[pri], st)) {
                    // this backoff has just been started in this moment,
                    // while others are busy, do nothing

                } else {
                    fprintf(stderr,
```

```
                    "BackoffTimer::pause new EDCA rule, but rtime < slottime!\n");
                    exit(1);
                }
            } else if (sr > 0 + ROUND) {
                slots = int (floor((sr + ROUND) / slottime));
            }

            assert(busy_ && !paused_);
            if ((rtime_[pri] - (slots * slottime) > 0.0)
                || (rtime_[pri] - (slots * slottime) > ROUND)) {
                if (EDCA) {
                    rtime_[pri] -= ((slots + 1) * slottime);
                    // +1 slot for new EDCA rule
                } else {
                    rtime_[pri] -= (slots * slottime);
                }
                decremented_[pri] = st;
                assert(rtime_[pri] >= 0.0);
            } else {
                if (rounding(rtime_[pri] - (slots * slottime), 0.0)) {
                    // difference within rounding errors
                    rtime_[pri] = 0;
                    decremented_[pri] = st;
                } else {
                    printf("ERROR in BackoffTimer\n");
                    exit(0);
                }
            }
            /*
             * decrease of AIFSwait is required because pause() and restart() can
             * be called immediately afterwards (in case of postbackoff), so that
             * normal backoff operation must proceed
             */
            if (st - stime_[pri] >= mac->getAIFS(pri)) {
                AIFSwait_[pri] = 0.0;
            } else {
                AIFSwait_[pri] -= st - stime_[pri];
                if (AIFSwait_[pri] < 0)
                    AIFSwait_[pri] = 0;
            }
        }
    }
    s.cancel(&intr);
    paused_ = 1;
}

// resume finds the smallest residual time and
// reschedules the event.
void BackoffTimer_802_11e::resume()
{
    double delay = INF;
    int prio = MAX_PRI + 1;
    Scheduler & s = Scheduler::instance();

    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (backoff_[pri]) {
            double delay_ = 0;
            round_time(pri);
            AIFSwait_[pri] = mac->getAIFS(pri);
```

```
            stime_[pri] = s.clock();
            if (EDCA && rounding(decremented_[pri], s.clock()))
                delay_ = rtime_[pri] + AIFSwait_[pri] + slottime;
            else
                delay_ = rtime_[pri] + AIFSwait_[pri];
            //decremented_[pri]=0;
            // find the smallest time
            if ((delay_ < delay) && backoff_[pri]) {
                delay = delay_;
                prio = pri;
            }
        }
    }
    // scehdule the smallest time
    if (prio < MAX_PRI + 1) {
        assert(rtime_[prio] + AIFSwait_[prio] >= 0);
        s.schedule(this, &intr, rtime_[prio] + AIFSwait_[prio]);
        paused_ = 0;
    } else {
        cout << "ERROR: wrong priority in BackoffTimer::resume() \n";
        exit(0);
    }
}

void BackoffTimer_802_11e::restart()
{
    busy_ = 1;
    double delay = INF;
    int prio = MAX_PRI + 1;
    Scheduler & s = Scheduler::instance();

    for (int pri = 0; pri < MAX_PRI; pri++) {
        if (backoff_[pri]) {
            double delay_ = 0;
            round_time(pri);
            stime_[pri] = s.clock();
            if (EDCA && rounding(decremented_[pri], s.clock()))
                delay_ = rtime_[pri] + AIFSwait_[pri] + slottime;
            else
                delay_ = rtime_[pri] + AIFSwait_[pri];
            //decremented_[pri]=0;
            if (delay_ < delay) {
                delay = delay_;
                prio = pri;
            }
        }
    }
    if (prio < MAX_PRI + 1) {
        assert(rtime_[prio] + AIFSwait_[prio] >= 0);
        s.schedule(this, &intr, rtime_[prio] + AIFSwait_[prio]);
        paused_ = 0;
    } else {
        cout << "ERROR: wrong priority in BackoffTimer::restart() \n";
        exit(0);
    }
}


int BackoffTimer_802_11e::backoff(int pri)
```

```
{
    return backoff_[pri];
}

void UpdateTableTimer::handle(Event * e)
{
    // reschedule the timer
    Scheduler & s = Scheduler::instance();
    s.schedule(this, &intr, mac->update_interval_);

    // update the CW for broadcast messages
    mac->updateTableHandler();
}
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] The cmu monarch wireless and mobility extensions to ns-2.

[2] The ns-2 network simulator.

[3] The ns-users mailing list.

[4] The ns manual (formerly ns notes and documentation), 2006.

[5] Lee Armstrong. Dedicated short-range communications project.

[6] F. Bai, N. Sadagopan, and A. Helmy. User manual for important mobility tool generators in ns-2 simulator, 2004.

[7] J. J. Blum, A. Eskandarian, and L. J. Hoffman. Challenges of intervehicle ad hoc networks. *IEEE Trans. Intelligent Transportation Systems*, 5(4):347–351, December 2004.

[8] A. Trivino Cabrera and E. Casilari. Network simulator: A learning tool for wireless technologies, 2006.

[9] Jae Chung and Mark Claypool. Ns by example.

[10] Federal Communications Commission. Amendment of the commission's rules regarding dedicated short-range communication service in the 5.850-5.925 ghz band, fcc 02-302. Technical report, FCC, November 2002.

[11] Javier Gomez and Andrew T. Campbell. A case for variable-range transmission power control in wireless multihop networks. In *INFOCOM*, 2004.

[12] Marc Greis. Tutorial for the network simulator ns.

[13] Dr jiunn Deng and Ruay shiung Chang Nonmembers. A priority scheme for IEEE 802.11 DCF access method, August 09 1999.

[14] Gökhan Korkmaz, Eylem Ekici, Füsun Özgüner, and Ümit Özgüner. Urban multi-hop broadcast protocol for inter-vehicle communication systems. In Kenneth P. Laberteaux, Raja Sengupta, Chen-Nee Chuah, and Daniel Jiang, editors, *Proceedings of the First International Workshop on Vehicular Ad Hoc Networks, 2004, Philadelphia, PA, USA, October 1, 2004*, pages 76–85. ACM, 2004.

[15] Stefan Mangold, Sunghyun Choi, Ole Klein, and Guido Hiertz. IEEE 802.11e wireless LAN for quality of service, August 03 2002.

[16] Matthew S. Gast. *802.11 Wireless Networks: The Definative Guide*. O'Reily, 1st edition, April 2002.

[17] C. Siva Ram Murthy and B. S. Manoj. *Ad Hoc Wireless Networks: Architectures and Protocols*, chapter Wireless LANs and PANs, pages 74–75. Prentice Hall, 2004.

[18] NHTSA. Intellegent transportation systems, 2006.

[19] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *MOBICOM*, pages 151–162, 1999.

[20] R. Ramanathan and R. Rosales-Hain. Topology control of multihop wireless networks using transmit power adjustment. In *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'00)*, pages 404–413, March 2000.

[21] Thomas J. Schriber and Daniel T. Brunner. Inside discrete-event simulation software: how it works and why it matters. In *Winter Simulation Conference*, pages 72–80, 1999.

[22] Marc Torrent-Moreno, Daniel Jiang, and Hannes Hartenstein. Broadcast reception rates and effects of priority access in 802.11-based vehicular ad-hoc networks. In Kenneth P. Laberteaux, Raja Sengupta, Chen-Nee Chuah, and Daniel Jiang, editors, *Proceedings of the First International Workshop on Vehicular Ad Hoc Networks, 2004, Philadelphia, PA, USA, October 1, 2004*, pages 10–18. ACM, 2004.

[23] Vehicle Safety Communications Consortium consisting of, BMW, DaimlerChrysler, Ford, GM, Nissian, Toyota, and VW. Vehicle safety communications project task 3 final report: Identify intellegent vehicle safety applications enabled by DSRC, March 2005.

[24] Sven Wiethlter and Christian Hoene. Design and verification of an IEEE 802.11e EDCF simulation model in ns-2.26. Technical report, Telecommunication Networks Group, Technische Universitt Berlin, November 2004.

[25] Qing Xu, Raja Sengupta, and Daniel Jiang. Design and analysis of highway safety communication protocol in 5.9 ghz dedicated short range communication spectrum, Spring 2003.

[26] Y. Xaio. Enhanced DCF of 802.11e to Support QOS. *IEEE Wireless Communications and Networking*, pages 16–20, March 2003.

[27] Xue Yang, Jie Liu, Feng Zhao, and Nitin H. Vaidya. A vehicle-to-vehicle communication protocol for cooperative collision warning. In *MobiQuitous*, pages 114–123. IEEE Computer Society, 2004.