THE EFFECTS OF MAC LAYER PARAMETERS ON QUALITY OF SERVICE PROVIDED BY 802.11E

A THESIS IN
Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment of
the requirements for the degree

MASTER OF SCIENCE

by
DANIEL TANGYI FOKUM

B.A., Park University, 2000

Kansas City, Missouri
2005

THE EFFECTS OF MAC LAYER PARAMETERS ON

QUALITY OF SERVICE PROVIDED BY 802.11E

Daniel Tangyi Fokum, Candidate for the Master of Science Degree
University of Missouri-Kansas City, 2005

ABSTRACT

The IEEE 802.11e standard was written to provide quality of service over 802.11a and 802.11b networks. In this thesis we evaluate the performance of the 802.11e protocol over an 802.11g network. We investigate the effects of the MAC parameters on the quality of service provided by the network particularly in saturated conditions. Based on our research we conclude that the arbitration interframe space and the minimum and maximum contention window sizes have the greatest effect on quality of service. Therefore, stations ought to be able to adjust these parameters, particularly the contention window sizes, in saturated conditions. In this thesis we present a mechanism called AEDCF-CW/PF to make these adjustments.

This abstract of 109 words is approved as to form and content.

Cory Beard, Ph.D.
Associate Professor
Department of Computer Science and Electrical Engineering

The undersigned, appointed by the Dean of the School of Computing and Engineering have examined a thesis titled "The Effects of MAC Layer Parameters on Quality of Service Provided by 802.11e," presented by Daniel T. Fokum, candidate for the Master of Science degree, and hereby certify that in their opinion it is worthy of acceptance.

Cory Beard, Ph.D. Department of Computer Science and Electrical Engineering	Date
Ken Mitchell, Ph.D. Department of Computer Science and Electrical Engineering	Date
Vijay Kumar, Ph.D. Department of Computer Science and Electrical Engineering	Date

CONTENTS

ABSTRACT ii

ILLUSTRATIONS vii

TABLES ix

TABLES ix

ACKNOWLEDGMENTS x

Chapter

1. INTRODUCTION 1

2. OVERVIEW OF THE CONTENTION PERIOD ACCESS FUNCTIONS 4

DCF 5

DCF Operation 5

EDCF 8

EDCF Operation 12

3. REVIEW OF PREVIOUS RESEARCH 14

4. RESEARCH MOTIVATION 30

5. RESEARCH RESULTS 33

Research Goals 33

DCF Simulation Approach 34

Results of Basic DCF Simulation 36

Results of Frame Dropping DCF Simulation 38

EDCF Simulation Approach 40

Results of Basic EDCF Simulation 44

Results of EDCF Simulation with Random Frame Dropping 49

Results of EDCF Simulation with Varying Retry Limits 50

AEDCF Simulation Approach 51

Results of Adaptive EDCF Simulation 53

Results of Adaptive EDCF Simulation with Varying Persistence Factor 55

Results of Adaptive EDCF Simulation with Varying Persistence Factor and CW_{max} 57

Impact of Changes 64

6. CONCLUSION 66

Appendix

A. DCF CODE LISTING 71

B. DCF WITH RANDOM FRAME DROPPING CODE LISTING 80

C. EDCF CODE LISTING 91

D. EDCF WITH RANDOM FRAME DROPPING CODE LISTING 105

E. EDCF WITH DIFFERENT FRAME RETRY LIMITS CODE LISTING 121

F. ADAPTIVE EDCF CODE LISTING 135

G. ADAPTIVE EDCF WITH VARYING PERSISTENCE FACTOR CODE LISTING 151

H. ADAPTIVE EDCF WITH VARYING PERSISTENCE FACTOR CODE AND CW_{MAX} CODE LISTING 168

REFERENCES 185

VITA 189

ILLUSTRATIONS

Figure Page
1. Relationships between Interframe Spaces [1] 8

2. Relationships between Interframe Spaces [17] 12

3. System Response Times for Basic DCF 37

4. Queue fill for Three Sources using DCF 37

5. Frame Drop Probability for DCF 38

6. Basic DCF versus DCF with Frame Dropping 39

7. System Response Time for Basic EDCF 45

8. Response Time Comparison for EDCF at Low Loads 46

9. Comparison of EDCF and DCF 47

10. Frame Drop Probability for EDCF 48

11. Comparison of System Response Times for Basic EDCF and EDCF with Random Frame Dropping 50

12. System Response Time basic EDCF versus EDCF with Different Frame Retry Limits 51

13. System Response Time Basic EDCF versus Adaptive EDCF 54

14. Frame Drop Probabilities for Adaptive EDCF 55

15. AEDCF with Varying Persistence Factor versus Basic AEDCF 56

16. Frame Drop Probability for AEDCF with Varying PF 57

17. System Response Time Basic EDCF versus AEDCF-CW/PF 59

18. System Response Time AEDCF versus AEDCF-CW/PF 59

19. Response Time Comparison for Traffic Classes Under AEDCF-CW/PF 60

20. Comparison of AEDCF-CW/PF and EDCF at Low Loads 61

21. Frame Drop Probabilities for AEDCF-CW/PF 62

22. System Response Time Variance for AEDCF-CW/PF and EDCF 63

23. System Response Time Variance AEDCF and AEDCF-CW/PF 64

24. System Response Time for EDCF, AEDCF and Their Variants 65

25. Collision Probability for DCF, EDCF and AEDCF 67

TABLES

Table Page
1. MAC Parameters used for DCF Simulations 35

2. MAC Parameters used for EDCF Simulations 42

ACKNOWLEDGMENTS

The author wishes to thank Dr. Cory Beard for his comments and supervision of the thesis. The author would also like to thank the members of the committee for their time in reviewing this thesis. Finally, the author would also like to thank his family and friends for encouraging him and praying for him as he undertook this work.

CHAPTER 1

INTRODUCTION

In the last few years wireless networks based on the Institute of Electrical and Electronics Engineers (IEEE) 802.11 standard have become more prevalent due to their low cost of implementation, and the IEEE 802.11 standard's interoperability with the existing Ethernet standard.

The 802.11 standard was developed by the IEEE to allow transfer of data over wireless networks. In addition to the original 802.11 standard several extensions of 802.11 exist. These extensions include 802.11a, 802.11b, and 802.11g. Each of these standards provides different data rates over different physical layers, but they all use the same medium access control (MAC) functions. In chapter 2 we will discuss some of these MAC functions in greater detail.

The 802.11 standard was originally proposed in 1997. It was designed to provide data rates of one or two megabits per second (Mbps) over the 2.4 GHz frequency band using either frequency hopping spread spectrum or direct sequence spread spectrum. Since each of these modulation methods has different physical layer characteristics each modulation method also has different MAC layer parameters.

The 802.11a standard is an extension to the 802.11 standard. It is supposed to provide data rates of up to 54 Mbps in the 5 GHz band. Unlike basic 802.11, this standard uses orthogonal frequency division multiplexing for modulating data.

The 802.11b standard is another extension to basic 802.11 that is sometimes known as Wi-Fi. Since 802.11b operates in the 2.4 GHz band, it is backward compatible with 802.11; however, it provides speeds of up to 11Mbps. Unlike basic 802.11, 802.11b uses a modulation method called complementary code keying (CCK).

In 2003 the IEEE approved yet another extension to the 802.11 standard called 802.11g. This extension provides speeds of up to 54Mbps in the 2.4GHz band. As a result, this standard is backward compatible with 802.11b, and consequently 802.11.

In spite of their ease of set-up, 802.11 networks have the following flaws:

- One of the 802.11 MAC functions, the Point Coordination Function (PCF), is inefficient.
- The MAC functions are unable to provide different levels of service to different traffic classes.
- The 802.11 MAC functions use the wireless medium inefficiently.

As a result of these flaws, the IEEE's 802.11 working group commissioned a task group, called Task Group E (TgE), to develop a new 802.11 standard called 802.11e that will provide quality of service over 802.11a and 802.11b networks.

Quality of Service (QoS) is a term used to define the different characteristics of a traffic flow. Some of these characteristics include frame delay, jitter (the amount of variation in frame delay), frame dropping probability, priority etc. In this document we will evaluate the effects of the different 802.11g parameters on the quality of service that is enjoyed by a traffic class.

Our work is motivated by a gap that we see in today's research literature -- most of the 802.11 products currently sold are 802.11 b/g compatible; however, there is little or no literature on 802.11e running over an 802.11g network. In our review of the literature we did not find any papers on the performance of 802.11e over an 802.11g network. This document will fill part of that gap.

Extending the 802.11 MAC for traffic classes to receive different levels of service will open up the opportunities for 802.11 WLANs. For example these LANs may now be used within homes to deliver video or music to a remote device. Outside of the home QoS-capable 802.11 LANs may be used to provide prioritization of emergency traffic, or to prioritize the delivery of video or audio data in a wireless hotspot.

The rest of this document is laid out as follows. In chapter 2 we discuss the contention period access functions of 802.11 and 802.11e. In chapter 3 we will review research that has already been done in the area of QoS over 802.11 networks. In chapter 4 we present our motivation for carrying out our research, and provide arguments for using simulation. In chapter 5 we present our research methodology and results. Finally we provide concluding remarks in chapter 6.

CHAPTER 2

OVERVIEW OF THE CONTENTION PERIOD ACCESS FUNCTIONS

In this chapter we provide an overview of the different functions used by the 802.11 Medium Access Control (MAC) for access to the medium during the contention period.

The 802.11 standard [1] defines the basic service set (BSS) as the building block of an 802.11 LAN. If the service set contains two stations with no access point, that service set is called an independent basic service set (IBSS). If the service set contains at least two stations with an access point, that service set is called a basic service set (BSS).

In an 802.11 network that contains an access point time is split into two types of periods -- a contention-free period (CFP) and a contention period (CP) -- that alternate at regular intervals. Together both of these periods constitute a superframe. The 802.11 MAC defines two functions for access to the medium. The Point Coordination Function (PCF) is used by the access point (AP) during the contention-free period to poll stations that associated with the AP. On the other hand the Distributed Coordination Function (DCF) is used by stations to gain access to the medium during the contention period.

The DCF is defined as the fundamental method of access to the medium, and must be implemented by all stations in a basic service set. The PCF on the other hand has not been widely implemented in many of the 802.11 products that are currently available [21]. Since this thesis focuses on medium access during the contention period we will not discuss the PCF further.

DCF

DCF is a Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) method of access to the medium.

DCF Operation

In its basic form, DCF operates as described in the following paragraphs. When a station has a frame to transmit it senses the medium until the medium has been found idle for a period of time known as the DCF Interframe Space (DIFS). Once the medium has been found to be idle for a DIFS time, the station that wishes to transmit will pick a random number, w, between 0 and the current size of the contention window, CW. After one slot time (the slot time is a time value that depends on the physical layer being used) the medium is sensed again. If the medium is found to be idle the backoff counter, w, is decremented. If the medium is found to be busy, the backoff counter is "frozen" until the medium is found to be idle for a DIFS time. At this time the backoff counter is restarted. Once the backoff counter reaches zero, the frame is transmitted. When the receiving station receives the transmitted frame, it verifies that the frame was received correctly, and that no other stations were transmitting at the same time the frame was received. If both of these conditions are true, the destination station (Under 802.11 all stations correspond with the access point. Stations cannot communicate directly.) sends an acknowledgement (ACK) frame to the sender after a period of time called a Short Interframe Space (SIFS).

If the sending station does not receive an ACK frame within an ACKTimeout time, the sending station assumes that a collision took place on the medium. Following a collision, the sending station increases its contention window as follows: $CW_{new} = ((CW_{old} + 1)*2) - 1$, as long as CW_{new} is less than or equal to the maximum contention window, CW_{max}, defined for this PHY. In addition to the contention window doubling procedure, the station also increments its retry counter by one to indicate the number of transmission attempts for this frame. After this, the medium sensing and backoff processes described above are repeated. This process continues until either an ACK is received, or the number of transmission attempts for this frame reaches the frame retry limit defined for this class of frames, or the frame exceeds its lifetime limit. This time value is called the dot11MaxTransmitMsduLifetime in the 802.11 standard [1]. This parameter refers to the amount of time from the initial transmission attempt of a frame, after which all transmission attempts for this MAC Service Delivery Unit (MSDU) shall cease. In other words, the frame shall be dropped after this period [1].

The second form of DCF is the RTS/CTS method. When a station wishes to transmit a frame under this method, it senses the medium until the medium has been found idle for a DIFS time. Next the station broadcasts a Request to Send (RTS) frame to the station that it would like to communicate with. If the Network Allocation Vector (NAV) at that station indicates that the medium is currently idle, that station will respond with a Clear to Send (CTS) frame after a SIFS time. If the station that sent the RTS frame does not receive a response it assumes that a collision has occurred, and it invokes the backoff procedure that was described above. The backoff will continue until either a CTS frame is received, or the number of transmission attempts for the RTS frame exceeds dot11ShortRetryLimit.

When the CTS frame is received, the station that sent the RTS frame will begin transmission of the data one SIFS time after the CTS frame is received. After the successful transmission of the data the receiving station shall respond with an ACK frame.

It is worth noting that the RTS/CTS procedure was developed to combat the hidden node problem[1] in a wireless network. In addition, all the control frames exchanged in the RTS/CTS method of access contain enough data such that the stations within the range of the corresponding stations know when the medium will be busy, and the duration of the transmission. As a result all the stations that are within range of the corresponding stations set their NAV to

[1] The hidden node problem occurs when a mobile station can "hear" transmissions from one station, but not another station in the same WLAN.

indicate that the medium is busy once a CTS frame is transmitted. This is done so that the neighboring stations do not interrupt the transmission.

Figure 1 is presented in the 802.11 standard [1] to illustrate the relationships that exist between the IFS times.

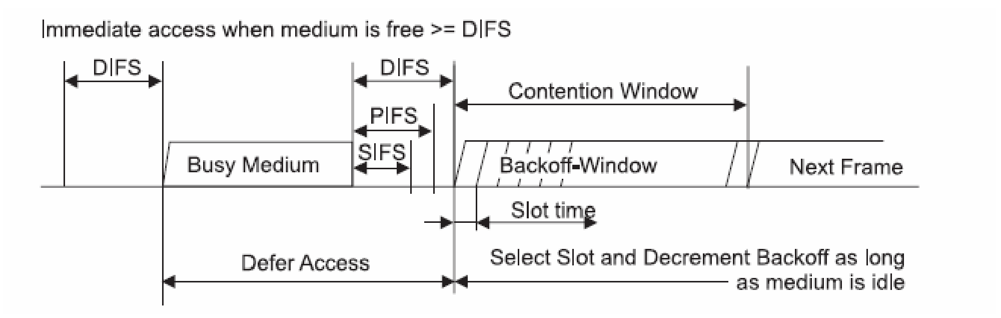
Immediate access when medium is free >= DIFS
DIFS
DIFS
Contention Window
PIFS
SIFS
Busy Medium
Backoff-Window
Next Frame
Slot time
Defer Access
Select Slot and Decrement Backoff as long as medium is idle

Figure 1: Relationships between
Interframe Spaces [1]

EDCF

As of October 24, 2005 the final 802.11e standard was not available on the IEEE's 802.11 web site. The 802.11e draft is currently with the Revisions Committee of the IEEE awaiting final approval and publication. In the absence of the standard, the information that we have about the upcoming standard was gleaned from several papers on 802.11e, as well as a document from the 802.11 Task Group E website [17].

The 802.11e MAC replaces the notion of a basic service set (BSS) with one of a QoS Basic Service Set (QBSS). The presence of a QoS-capable access point (now called the hybrid coordinator (HC)) allows for the use of the hybrid coordination function during the contention-free periods of a superframe.

In [8] the 802.11e MAC is said to provide an Enhanced Distributed Coordination Function (EDCF) for access during the contention period, and a Hybrid Coordination Function (HCF) to support station polling. HCF operates in both the contention-free and contention periods, hence it is said to be hybrid. HCF is analogous to the PCF in 802.11, while EDCF is analogous to DCF in 802.11. Unlike DCF, EDCF provides prioritization by allowing for different traffic classes called access categories (AC). Each AC has a separate queue within each station along with different contention window parameters. In addition, each access category has to sense the medium for a different interframe space time prior to beginning a transmission. This time value is defined as an Arbitration Interframe Space (AIFS). Each AIFS is equal to a DIFS time + k*aSlotTime, where k can potentially be zero. It should be observed that the lower the value of k, the higher the AC's priority, because the AC would have to sense the medium for a shorter time interval. In addition to the new access functions described above, 802.11e replaces the dot11MaxTransmitMsduLifetime parameter with a new variable called dot11MSDULifetime. This variable is defined per access category, and it refers to the amount of time that a frame from a given access category can remain queued up. Finally, [8] states that frame losses in 802.11e occur if the number of transmission attempts for a given frame exceeds either the short/long retry limit, or if the queuing delay for the frame exceeds the dot11MSDULifetime limit for this traffic class.

From [10] it is seen that the 802.11e MAC also has a new variable called TXOPlimit that defines the amount of time that an access category has access to the medium. Once an access category wins access to the medium, it may continue to transmit frames separated by SIFS without sensing the medium for AIFS[AC], as long as the station does not use the medium for a period that exceeds TXOPlimit[AC]. In addition it is seen that the eight priorities defined by the 802.1D standard are mapped to four access categories in 802.11e. Another enhancement of the 802.11e MAC is the fact that stations are now able to communicate directly with each other without necessarily involving the access point. This feature is known as the Direct Link Protocol of 802.11e.

From [17] it is seen that if multiple queues within a station count down to zero simultaneously, a virtual collision occurs. If a virtual collision occurs the transmission opportunity is handed to the AC with the highest priority, while the other AC execute the backoff algorithm. When a virtual collision occurs the transmission attempt counter for each frame in the lower priority queues is not incremented to indicate that a collision occurred.

The new 802.11e standard also implements one new feature that should help it use the medium more effectively [21]. This feature is called the group acknowledgement mechanism. When this mechanism is used, a station may send several frames without waiting for individual acknowledgement of the frames. After the station has sent a burst of frames it will send out a group acknowledgement request (GroupAckReq). The receiver will respond by sending a GroupAck frame that will list information on all the frames that were correctly received. As with the basic ACK mechanism, both the GroupAckReq and GroupAck frames are separated by an SIFS interval. If the GroupAck frame shows that any frames were not received correctly, the sender will try to resend those frames as long as it can do so (Recall that frames are dropped if the frame retry limit for those frames is exceeded, or if the frames have been in the queue for more than dot11MSDULifetime[i].)

In implementing the features listed above, TgE has solved the flaws of 802.11 listed in chapter 1. For example the EDCF function provides prioritization of traffic streams, while the direct link protocol and group ACK mechanism allow for more efficient use of the medium.

Figure 2 shows the relationships that exist between the interframe spaces under EDCF. Observe the close similarities between this figure and figure 1.

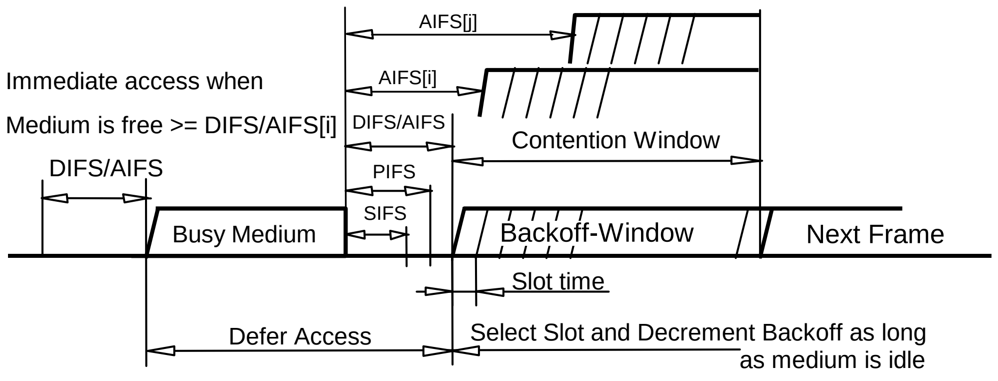

Figure 2: Relationships between
Interframe Spaces [17]

EDCF Operation

In the basic access mode a QoS station (QSTA) will have at least four access category queues that contend for the medium independently. Each AC queue will independently sense that that the medium is idle for the AIFS[i] time that is defined for that queue. Next, the AC queue will randomly pick a backoff interval, and count down to zero. As stated above, if multiple AC queues count down to zero simultaneously, the transmission opportunity is granted to the AC with the highest priority, while the other colliding queues double their contention windows and attempt to sense the medium again. In earlier versions of the 802.11e draft, the contention window was increased by a factor called the persistence factor, PF. The persistence factor was to range between 1 and 2, and increasing in steps of 1/16ths [17]; however, in [16] this factor is said to have been stricken from the draft.

DCF and EDCF operate similarly. Apart from the presence of multiple queues within a station, and the fact that when an internal collision occurs the transmission opportunity is granted to the queue with the highest priority, the functions are essentially identical. In light of this similarity [16] and [23] view each AC as a virtual station (essentially separate stations operating DCF) with its backoff instance and contention window parameters. The reader is referred to the section on DCF operation to review how DCF operates.

CHAPTER 3

REVIEW OF PREVIOUS RESEARCH

In this section we present an overview of the research that has already been done on quality of service in 802.11 wireless LANs. Since this area is a rapidly evolving area of research most of the papers we reviewed were limited to the period from 2003 and forward.

In reference [3] the authors argue that 802.11 MAC is not as unfair as some researchers have stated. Reference [3] defines the notions of long-term[2] and short-term[3] fairness, and then develops equations to show that in the short-term 802.11 is actually fairer than slotted ALOHA, which is considered to have good fairness properties.

Bianchi's paper on the performance analysis of DCF [4] was one of the most widely referenced papers that we found in our review of previous research. In this paper Bianchi analyzes the saturation[4] throughput of an ideal 802.11 channel. By using a Markov chain Bianchi evaluates the probability, τ, that a station transmits in a given slot time. τ is then used to compute the throughput of an

[2] According to [3] a MAC layer is said to be long-term fair if the probability of successful accesses to the medium converges to 1/N for N competing hosts over an extended period of time.

[3] According to [3] a MAC layer is said to be short-term fair if the stations are able to get access to the medium relatively fairly over short periods of time.

[4] A station is said to be saturated when it always has a frame to send.

802.11 network under the basic and RTS/CTS channel access methods. Using these results, Bianchi shows that DCF achieves much better throughput using the RTS/CTS channel access method. In addition, this paper shows that 802.11 throughput -- regardless of the channel access method -- depends on the number of back-off stages, m, and the initial contention window size, W. This article concludes by saying that the RTS/CTS channel access scheme should be used in most practical cases since network performance does not appear to depend as much on the system parameters under this scheme.

Since the performance of 802.11 DCF is dependent on the number of competing wireless stations [4], Bianchi and Tinnirello provide a mechanism for estimating the number of competing stations in an 802.11 network [5]. The authors state that information on the number of competing stations can be used to help stations decide when to switch from the basic medium access method to the RTS/CTS medium access method. It is also worth noting that the results presented in this paper [5] are an extension of the work presented in [4], and that the results presented in [5] are derived with the assumption that all the stations experience ideal channels. Using the result for τ derived in [4], an expression is obtained for n, the number of competing stations. The authors then state that since each station can obtain information on p, the conditional collision probability, each station can determine the value for n. The authors contend that p, may be estimated by each station counting the number of experienced collisions, C_{coll}, and the number of busy slots, C_{busy}, and dividing the sum of these two numbers by *B*, the total number of slots that the station has observed. The authors conclude this paper by stating that a Kalman filter estimation of the number of stations in a network is better than one based on an auto-regressive moving average (ARMA) filter, since the Kalman filter provides better estimates of the number of competing stations.

In [6] Ferré et al. extend Bianchi's original work to demonstrate the throughput of the 802.11e MAC. The authors present graphs to show that 802.11a throughput is dependent on the length of the frames used by the 802.11a transmission mode, with throughput being greatly affected by frame length especially for the higher PHY modes. The authors state that the frame length affects throughput because MAC overheads tend to be much higher than data transmission delays for shorter frames. Unlike Bianchi [4], these authors argue that the RTS/CTS mode decreases network efficiency because two additional frames and two additional SIFS periods elapse for one MSDU under this scheme. However, this assumption only holds true when the number of competing stations is small i.e. less than or equal to three. The authors then argue that it is not efficient to use the RTS/CTS scheme when there is a small number of competing stations, as the scheme is wasteful under those circumstances. However, once the number of users increases it is better to use the RTS/CTS scheme, as in the event of a collision only the RTS frame is lost. Under the basic scheme on the other hand the entire frame is lost in the event of a collision. Next, Ferré et al. present graphs to show the performance of 802.11e, and show that a given AC can still achieve good throughput even when AIFS[AC] is greater than DIFS, provided that the CW_{min}[AC] is small enough to allow rapid access to the medium. Finally, the authors conclude by stating that EDCF alone cannot provide QoS guarantees. QoS guarantees can be provided only if a polling scheme such as HCF is used.

In a bid to determine whether the new MAC protocols for 802.11e provide better QoS, Garg et al. study EDCF and HCF in [7]. At the end of their paper they conclude that the new protocols are highly dependent on the protocol parameters, and that the process of choosing these parameters needs to be studied better. The other facts gleaned from this paper include the following: HCF provides better channel utilization than EDCF, and that EDCF and DCF return very similar values for medium utilization. The authors add that EDCF is better than DCF because it provides differentiation to different flows. When considering the performance of EDCF, the following observations can be made: an increase in either AIFS or CW_{min} for the low priority access category (AC) results in a decrease in the latency for higher priority ACs. The authors conclude that AIFS has a greater effect on reducing bandwidth variance and jitter. For example in their simulations, increasing AIFS resulted in reducing the jitter for the video stream. On the other hand CW_{min} did not appear to have an effect in improving the jitter for the video stream. The authors conclude their paper by stating that EDCF is an attractive channel access method because it is decentralized, and simple. They add that HCF provides better medium utilization, but it is a centralized channel access method, and therefore, it is less robust.

Other researchers [8] have focused on providing QoS via the Hybrid Coordination Function (HCF). In [8] the authors argue that using a two-state Markov model for a wireless channel is not good enough sometimes since a channel may still be able to transmit data under poor network conditions by lowering the physical bit rate. In order to provide better QoS under 802.11e, the authors suggested the use of Scheduling based on Estimated Transmission Times – Earliest Due Date (SETT-EDD). Under this algorithm the hybrid coordinator polls stations in the order of non-decreasing deadlines. This approach is very efficient since it has a time complexity of *O(n)*. Our work did not cover contention-free access to the medium, so this paper was mainly used to get an overview of the 802.11e MAC.

In reference [9], Lindgren et al. study PCF, 802.11e, distributed fair scheduling (DFS) and a channel access method called Blackburst[5] to determine which scheme provides better QoS. Based on their simulations, they conclude that EDCF and Blackburst give better performance to higher priority traffic at high loads -- with Blackburst giving the best performance. PCF and DFS on the other hand give better performance to lower priority traffic. In addition Blackburst and DFS provide very low jitter, while at high loads PCF and EDCF

[5] According to [9] Blackburst requires all high-priority stations to try to access the medium after constant intervals t_{sch}. Low priority stations get access to the medium using DCF.

do not. The authors conclude this paper by stating that while Blackburst provides the best medium utilization, it may be necessary to use EDCF if it is impossible for high priority stations to get access to the medium at regular intervals. On the other hand DFS provides better service to the high priority traffic, but it also allows the low priority traffic to get a fair share of medium access.

Mangold et al. [10] showed that under EDCF, the throughput is dramatically reduced once the number of stations becomes large. These authors argue that this is a result of the higher collision probabilities especially where CW_{min} and CW_{max} are small numbers. In [10] it is argued that the hybrid coordinator (HC) should set new values for the EDCF parameters when there are high collision rates. This paper also contributed to our understanding of the 802.11e MAC, as shown in the EDCF section in chapter 2.

Medepalli and Tobagi [11] present a paper in which they argue that stations using the 802.11 protocol all have a cycle, and that on average stations broadcast a frame in each cycle. This paper reiterates the claim made in [14], that the presence of a low rate link affects the throughput for all other stations. The paper then develops equations to compute the cycle time for an 802.11 network, as well as the average throughput per user. The paper concludes by displaying graphs to show the numerical results of their equations plotted against simulations. From the graphs it is seen that 802.11 throughput increases very slightly as the number of users increases. Once the maximum throughput is attained, adding more users to the network results in a decline in throughput. From this paper it is seen that while the average packet service time is not a useful metric under high loads, it is of great importance under finite loads and can aid in analyzing queuing delays in the system.

Another approach to providing QoS under DCF is the Distributed Deficit Round Robin (DDRR) algorithm [12]. With this algorithm, traffic is separated into different classes at each mobile station, and each class is assigned a service quantum rate to match its throughput requirement. In addition each traffic class has a deficit counter of accumulated quanta, and the class is only allowed to send data if its deficit counter is positive. Following the transmission of the frame, the deficit counter is reduced by the size of the frame. Due to the Binary Exponential Backoff (BEB) algorithm causing fluctuations in the throughput of DCF the authors of this paper reject the use of a backoff algorithm. Instead their algorithm picks a random number between 1 and $\beta > 1$ and multiplies this number by the interframe space (IFS) for this traffic class to reduce the probability of collisions. Based on this paper, DDRR has much higher throughput and much lower MAC delay than either distributed weighted fair queuing (DWFQ) or distributed fair scheduling (DFS). The authors conclude the discussion of their algorithm by observing that the combination of the IFS and backoff interval can provide QoS.

In the strictest sense EDCF only allows for prioritization of different AC; it does not allow for true QoS. True QoS can only be guaranteed if an admission control mechanism is deployed for use during the contention period. In reference [13] Moors and Pong introduce such a mechanism. These authors state that if there are no restrictions on the volume of traffic introduced from different traffic classes, medium access delays may become higher due to the increasing lengths of the backoff times. In addition, failing to restrict the amount of traffic admitted from one class might result in affecting the performance of traffic from other classes. Using Bianchi's results for the probability that a station transmits in a given slot [4], an expression is derived for the achievable throughput of each class. The flow admission algorithm then uses this expression to admit new flows only if it determines that there is sufficient bandwidth to support the new stream. The flow admission algorithm will also select the appropriate initial contention window size and transmission opportunity duration for each flow. With the algorithm presented in this reference, prior to a flow beginning a transmission, it has to request its desired bandwidth from the access point. If the access point determines that admitting the new flow will cause other flows to exceed their achievable throughput limit, the admission request is rejected. Using simulations, the authors show that their scheme is indeed effective. The authors conclude this paper by stating that if the direct link protocol proposed in the 802.11e draft is used, then the flow admission control algorithm will have to be moved from the AP to each mobile station.

Reference [14] discusses the impact of frame size on 802.11e throughput. This paper states that the values of CW_{min}[AC], CW_{max}[AC] and AIFS[AC] are announced by the AP (the hybrid coordinator) via beacon frames. Using simulations of 802.11e, the authors show that basic EDCF displays higher throughput than DCF for the highest priority traffic. EDCF throughput is even higher if the EDCF is used with a no ACK policy. In another simulation, the throughput for each AC is plotted against the number of stations in the network. From the simulation results it is seen that except for HCF and the higher priority AC, throughput decreases as the number of competing stations increases. The decrease in throughput is most marked for the low priority AC. Next, the authors present graphs to show the effect of one bad link on the other links within a network. These graphs show that one bad link penalizes all the good links in a network. The authors conclude by stating that the presence of TXOP in 802.11e networks reduces the effect of a bad link on all other links, since all stations get "their share" of the bandwidth.

In reference [15] Robinson and Randhawa extend Bianchi's original model to account for the 802.11e MAC. The authors argue that all other models to predict saturation throughput in 802.11 networks fail to account for the post-collision period. In an 802.11 network stations can infer that a collision has occurred by examining the frame check sequence (FCS) of a frame. If an incorrect FCS is found, the station assumes that a collision occurred, and so it defers access for an additional period known as the Extended Interframe space (EIFS). According to this paper all stations defer access during this period, so retransmissions have much higher probabilities of success. This paper extends Bianchi's model and then validates the accuracy of the newly developed model by comparing its analytical results to simulation results. Robinson and Randhawa claim that their model is validated by simulation. In concluding the paper, the authors state that two flows that are differentiated solely by contention window sizes are efficient just under low loads. As loads increase, low priority flows may be starved of bandwidth particularly in cases where the low priority flows have maximum contention windows that are much larger than the maximum contention windows for the high priority flows. On the other hand, AIFS differentiation does not affect the service provided to higher priority AC, in the presence of heavy low priority traffic. In the case of AIFS differentiation, low priority AC may be starved in the presence of heavy high priority traffic, since the low priority AC may rarely see idle slots. It is worth noting that the model presented in this paper does not provide a closed form solution for throughput, and that this model is limited just to saturated cases. Nonetheless, it remains a useful model.

Work has already been done on adapting 802.11e parameters based on network conditions. In [16] Romdhani et al. discuss a channel access method called Adaptive EDCF (AEDCF), that adjusts the size of the contention window based on the number of collisions that a station experiences. Standard EDCF resets the contention window to CW_{min} following a successful transmission; however, Romdhani et al. argue that this may not always be efficient. Reference [16] states that if a station has just experienced a collision, there is a very high probability that it would experience another collision. In addition the authors state that a static method[6] of decreasing the CW following a successful transmission may not be very efficient. In light of this, the authors propose that the CW be reduced using a station's history of collisions. According to the authors of [16], AEDCF is able to keep the delay low even under high loads, provide much better medium utilization, and maintain a lower collision rate than EDCF. The authors suggest that additional research needs to be done on adapting other EDCF parameters. Some of the parameters that can be adapted to network conditions include CW_{max}, the maximum number of retransmissions and the packet burst length.

In their letter [18] Xiao and Rosdahl demonstrate that a throughput upper limit, and a lower limit on the frame delay exist in 802.11 networks. Based on their findings, the authors state that any attempts to increase the throughput of 802.11 networks must also reduce the MAC overhead of 802.11.

In [19] Xiao stated that the model presented in Bianchi's paper [4] failed to meet the 802.11 standard in two respects; viz. the backoff counter continued to count down even if the current slot was busy, and the paper also assumed that a frame could be retransmitted infinitely. As a result Xiao derived a new model that rectified these two errors and also developed expressions to compute the frame dropping probability, frame dropping time, number of retransmissions and saturation delay. The paper concludes by displaying graphs of the numerical

[6] One example of a static method that is presented in [16] is an algorithm that resets CW to half of its last value as long as this value is higher than CW_{min}.

results to the equations above. From the graphs it is seen that as the number of stations increases the frame dropping probability, frame dropping time, and the average number of retransmissions per frame increase. The graphs also show that as the retry limit increases the frame dropping probability and retransmissions per frame decrease, while the frame dropping time increases. Finally, the graphs show that as CW_{min} increases, the throughput increases up to a certain maximum, and then throughput starts decreasing. The graphs also show that as CW_{min} increases, the average frame delay decreases up to a given minimum, and then frame delay starts increasing. It is interesting to observe that throughput and average frame delay decrease and increase respectively at different values of CW_{min}.

Reference [20] extends Xiao's work from [19] to the 802.11e MAC. In this paper it is assumed that either each station only has traffic from one class, or if multiple AC queues count down to zero simultaneously, the virtual collision handler will not decide which AC queue gets the TXOP. Rather, a collision will take place on the medium. Xiao then presents graphs to validate the numerical results of his model. From the graphs it is seen that increasing CW_{min} for one of the flows results in that flow getting less and less of the bandwidth, while the higher priority flow gets more bandwidth. In addition increasing CW_{min} results in the low priority flow getting higher frame delays. Increasing the backoff window-increasing factor[7] had the effect of giving more priority to the flow whose persistence factor was set to 2, once the other flow's factor exceeded 2. Xiao points out that the persistence factor may be used to minimize collisions while granting priorities. The last set of graphs indicate that differentiating the retry limits for the traffic classes has no effect on the total throughput of the system. However, class 1 traffic will have better throughput and delay than class 0 as long as $L_{1,retry} < L_{0,retry}$. As expected, the frame dropping probability for a traffic class decreases as the retry limit for that flow increases.

Xiao et al. [22] state that IEEE 802.11 networks have problems with fairness[8]. As a result they propose a backoff algorithm based on game theory instead of the normal Binary Exponential Backoff (BEB) algorithm used by standard DCF. They call their backoff algorithm the Nash Equilibrium Backoff (NEB) algorithm. Under the NEB algorithm, nodes that wish to communicate with each other are called a coalition. The members of a coalition broadcast their noise-to-signal ratio (NSR) so that members of other coalitions can compare the broadcast NSR with that from their coalition. The results of the comparison are then used to adjust the contention window (CW) as shown below:

If $NSR_{local} \leq NSR_{neighbor}$, then $CW = \lfloor CW \times \text{Random}[3, 4] \rfloor$;

[7] This was known as the persistence factor in early version of the 802.11e draft. It was later stricken from the draft

[8] The fairness problem arises because the binary exponential backoff algorithm for IEEE 802.11 tends to give an advantage to nodes that have just had a successful transmission.

Else CW = $\lfloor CW \times \text{Random}[0, 3] \rfloor$

According to [21] the NEB improves the fairness of TCP flows greatly in a network.

Xu et al. provide a method for analyzing the performance of IEEE 802.11 using a Markov model in [23]. The results in this paper are based on the assumption of having a fully connected network with ideal channels where the only packet losses are due to collisions. Using a multidimensional Markov chain this paper derives an expression for the total throughput of an 802.11e network. Once this expression is derived the authors make the following remarks:

- The smaller the CWs the greater the impact of the arbitration interframe space (AIFS).
- The combination of differentiated AIFSs and differentiated CWs introduce a more significant effect on the degree of differentiation.
- It is the difference in AIFS values rather than the absolute AIFS values that determine the degree of QoS differentiation.

It is interesting to note that while the authors of this article state that collisions are relatively infrequent in a wireless network, they state that the ratio of collisions to useful data transmissions ranges between 0.094 and 0.139 for a persistence factor of 1.

Reference [24] provides an overview of the QoS that will be provided by 802.11e. Most of the points from this paper have already been covered in the other references that were reviewed. However, Xu points out that the contention free burst feature of EDCF has not fully been investigated; therefore, some concerns exist about its unfairness. This paper also discusses the group ACK feature of 802.11e, which allows several frames to be sent prior to receiving an ACK from the recipient. This feature, he indicates, will make the 802.11e protocol much more efficient. Finally, the paper points out that admission control for contention-based channel access is a non-trivial task, and adds that the 802.11e draft calls for distributed admission control (DAC). Under this method stations would self-regulate themselves to protect the traffic that is already present in the QBSS. The paper concludes by stating that the QoS enhancement for the legacy 802.11 standard will be a great benefit.

In [25] Zhu et al. indicate that existing "QoS mechanisms for 802.11 can be classified into three categories: service differentiation, admission control and bandwidth reservation, and link adaptation." Service differentiation mechanisms such as EDCF and distributed fair scheduling do not provide any QoS guarantees; they just provide better than best effort services. The authors observe that while fair scheduling algorithms will share the bandwidth fairly between stations, they would also require a substantial rewrite of the 802.11 protocol. As a result EDCF is the only service differentiation mechanism that currently garners a lot of interest. Admission control and bandwidth reservation schemes for 802.11 include token-bucket based algorithms that may be used by a station to help determine when it is approaching heavy loads. This information may then be used to adjust the contention window parameters. Link adaptation algorithms in 802.11 networks include those that allow stations to maximize their throughput based on the current physical conditions. The authors conclude that future work on QoS in wireless networks should focus on integrating DiffServ and IntServ with 802.11e.

From our review of research, we have seen that the DCF function is a fair channel access method, but more research needs to be conducted on how to adjust EDCF parameters to maximize throughput. In addition, any attempt to increase the throughput of the EDCF or DCF functions will have to concentrate on reducing the MAC overhead of these functions.

CHAPTER 4

RESEARCH MOTIVATION

In the preceding chapter we provided an overview of the research that has been conducted in the area of QoS in 802.11 networks. As a result we set out to study different methods of handling the contention window, the persistence factor and the retry limit while minimizing the average frame delay. Our goal was to develop a mechanism that would allow stations to determine the values for these parameters independently.

Another goal for our research was to use knowledge of past experience i.e. collision history in determining the values for contention window parameters. It should be recalled that Romdhani et al. [16] had already developed an algorithm to use collision history, so in our work we extend their algorithm.

There were two reasons for setting out with these goals. First, we argued that since EDCF is a distributed function, stations ought to be able to determine some of their MAC layer parameters independently without waiting for broadcasts from the HC. Secondly, in a previous class the author was a member of a group that studied 802.11e. This group discovered that at high loads the high priority access category had several transmission attempts per frame. This group concluded that the high number of transmission attempts per frame was a result of the high priority AC choosing its backoff interval from a smaller range. The author concluded that if the backoff interval was chosen from a larger range, the number of collisions per station might be reduced. In this thesis we develop a mechanism for adapting the maximum contention window – and thus the backoff interval – for an access category based on collision history.

Once the goals above were identified, a choice had to be made between building an analytical model, or a simulation model. Analytical models are flexible, and would allow investigation of more relationships between MAC parameters. Ultimately the choice was made in favor of simulation for the following reasons:

- Analytical models for 802.11 are currently either too complex, or inaccurate.
- Analytical models may not necessarily yield a closed form solution to the problem. Recall that reference [15] did not have a closed form solution. This point makes a strong case for using simulation i.e. simulation models are used because we cannot completely solve a problem using an analytical model.
- Analytical models may not completely capture all the times within a model. For example, an analytical model cannot model the different amounts of time used to process the frame.

Having chosen to use simulation, our next goal was to write a series of CSIM programs to simulate the 802.11 functions. CSIM was chosen because of its ease of use, and also so that the author could build and validate an accurate simulation model. Most of the simulation currently done on 802.11 is based on models built in ns-2 (network simulator-2). There are few colloquial reports about these models not being accurate; therefore, we set out to build a valid model for ourselves. In the next chapter we will discuss the results of our simulations.

CHAPTER 5

RESEARCH RESULTS

In the preceding chapter we advanced reasons for choosing a simulation model over an analytical model for this thesis. Having chosen the simulation model we set out with the research goals below.

Research Goals

Our goals included the following:

- Investigating the effects of random frame dropping from queues on the average frame delay
- Studying the effects of starting a frame transmission with a contention window other than CW_{min}
- Studying the effects of varying the persistence factor based on collision rates
- Studying the effects of varying the retry limit for each class.

With the exception of random frame dropping, and varying the persistence factor, all these ideas had been tried in [16] and [20].

In choosing these goals we wanted to propose changes that would not require significant modification of the 802.11 MAC. The 802.11 standard is very successful; therefore, any changes that are suggested to the MAC should be able to interoperate with existing hardware very easily. In addition, any changes that are proposed ought to have an easy implementation, and not require a significant modification of the 802.11 standard.

DCF Simulation Approach

Our simulations consisted of several CSIM programs that matched the DCF and EDCF standards very closely. During our simulations we held the number of competing stations at 3, 5, 10, 15, 20, 25 and 30, while varying the mean frame interarrival time from 100 msec to 10 msec in descending steps of 2 msec[9]. All the stations were assumed to be within range of each other; therefore, we did not have to account for the hidden terminal problem. Each simulation was run for 300,000 frame arrivals, following a warm-up period in which 12,500 frames were processed. Our frame sizes were uniformly distributed between 1 and 4095 octets, with no fragmentation allowed. In addition the frames were entered into queues of infinite lengths. The simulations used the 802.11g MAC, and it was assumed that all control and data frames were sent over ideal channels at the maximum link rate of 54 Mbps. Please refer to the appendices for more detailed descriptions of our code. The MAC parameters shown in table 1 were used in all of our DCF simulations. These parameters were all drawn from the 802.11g standard.

[9] The more frequently frames arrived, the heavier the traffic load on the network.

Table 1: MAC Parameters used for
DCF Simulations

802.11 Parameter Name	Value
CW_{min}	31
CW_{max}	1023
aSlotTime	20 μs
aSIFSTime	10 μs
aDIFSTime	50 μs
aMPDUMaxLength	4095 octets
dot11LongRetryLimit	7
dot11MaxTransmitMSDULifetime	512 μs
MAC Header Length	34 octets
PHY Header Length	24 octets
ACK Length	14 octets

Results of Basic DCF Simulation

The goal of our first simulation exercise was to validate our DCF model by studying the queue fill for the model, as well as validating that system response times increased exponentially as the network became saturated. It is worth noting that in early versions of our simulations, the frame response times tended to approach infinity (over 25 seconds) under heavy loads when frames did not have a lifetime limit. Once the dot11MaxTransmitMSDULifetime parameter was introduced, frame delays were limited to about 0.005 seconds. In our simulations frames could only be dropped if the number of transmission attempts for that frame exceeded the frame retry limit, or if the frame in question had been in the queue for longer than dot11MaxTransmitMSDULifetime. Figures 3 and 4 display the results of our DCF simulations.

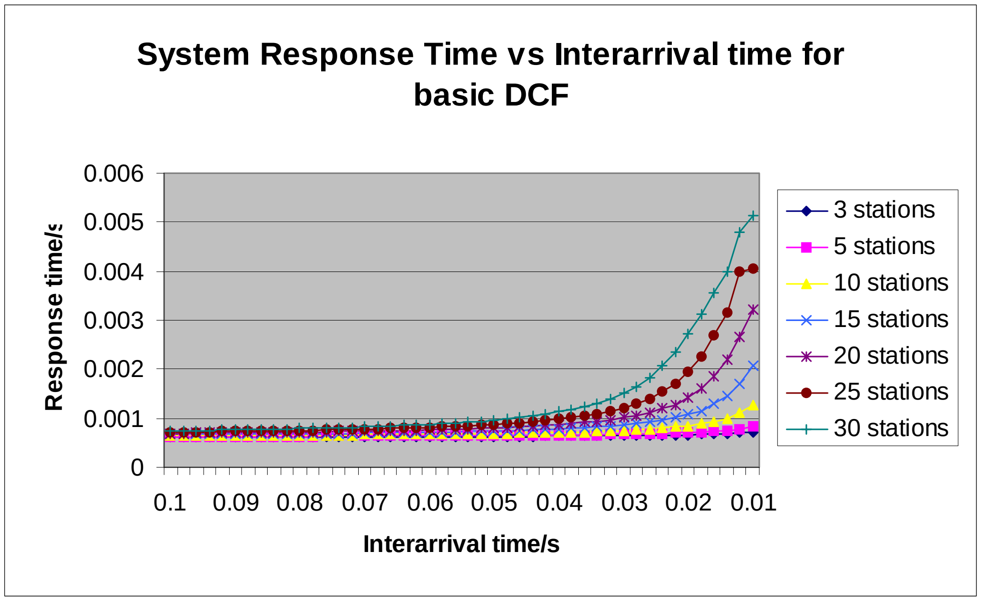

Figure 3: System Response Times for Basic DCF

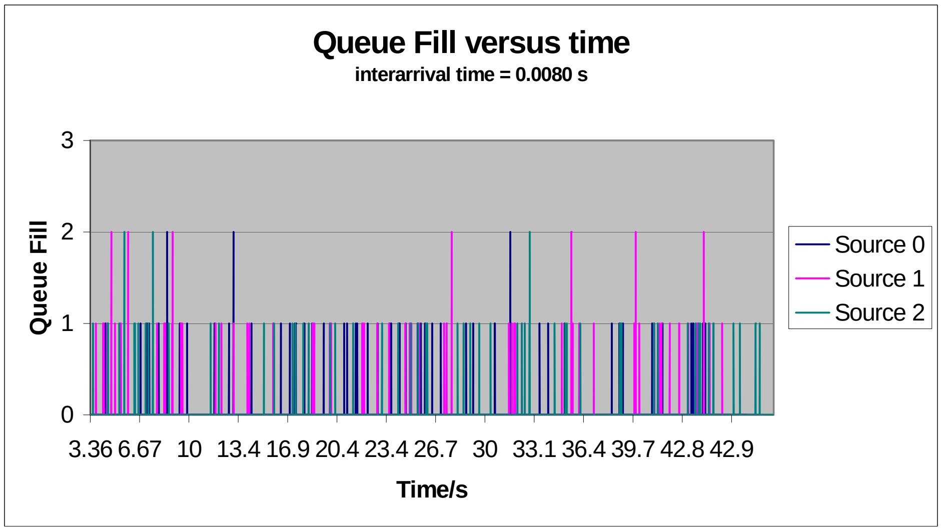

Figure 4: Queue fill for Three Sources using DCF

Since DCF does not provide any differentiation between traffic classes, one would expect that frames should be dropped with equally likely probability from all classes. As figure 5 indicates, this assumption is indeed correct. Recall that frames are dropped in our model only when the frame exceeds its maximum transmit lifetime or when the transmission attempts for that frame exceed seven.

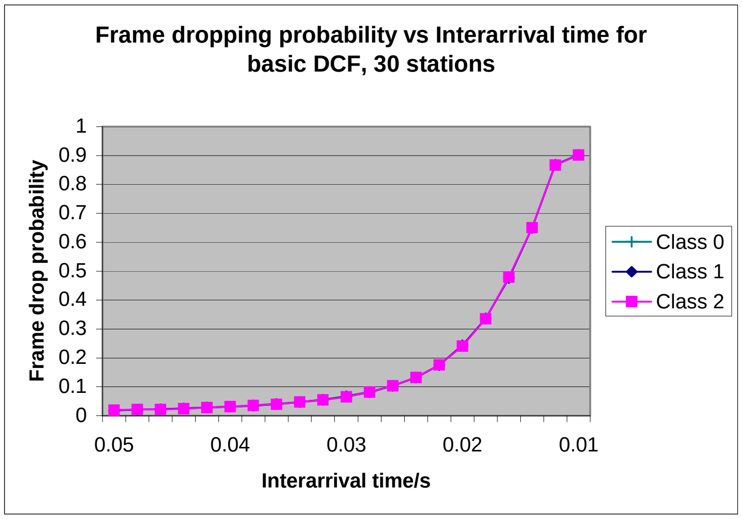

Figure 5: Frame Drop Probability for DCF

Results of Frame Dropping DCF Simulation

The goal of our next simulation exercise was to determine if random dropping of frames from each source's queue would help minimize the frame delay for each source. Our simulation program was written to drop 1 in 100 frames when the source's queue contained less than 5, 3, or 10 frames for sources with class 0, 1 or 2 traffic respectively. If the queue contained more than 5, 3 or 10 frames respectively the frame dropping probability was raised to 1 in 10. It was our opinion that a random number generator for frame dropping could be easily incorporated in each station since 802.11 stations already implemented a random number generator for computing the backoff interval. Based on our findings frame dropping appears to have little effect in lowering the frame delay for each source under heavy loading, as shown in figure 6.

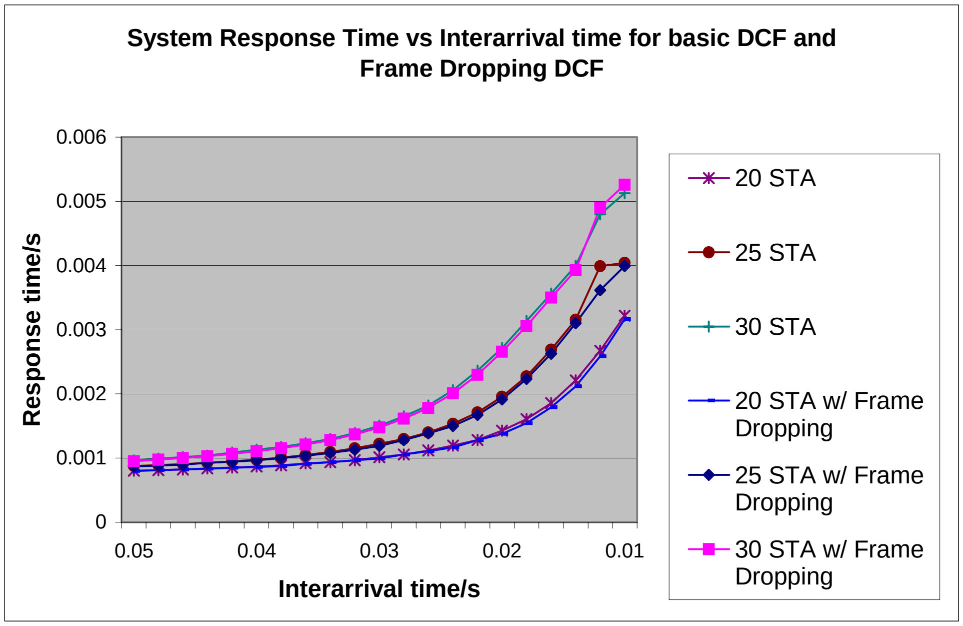

Figure 6: Basic DCF versus DCF with
Frame Dropping

EDCF Simulation Approach

Our next simulation exercise was to simulate the EDCF function. As in the case of the DCF simulations, we held the number of competing stations at 3, 5, 10, 15, 20, 25 and 30, while varying the mean frame interarrival time from 100 msec to 10 msec in descending steps of 2 msec. In each of our simulation exercises, at least 25% of the competing stations were marked as having high priority traffic i.e. having traffic from class 0, at least 50% of the stations were marked as having medium priority traffic, and the rest of the stations were marked as having low priority traffic. All the stations were assumed to be within range of each other; therefore, we did not have to account for the hidden terminal problem. Each simulation was run for 300,000 frame arrivals, following a warm-up period in which 12,500 frames were processed. Our frame sizes were uniformly distributed between 1 and 4095 octets, with no fragmentation allowed. In addition the frames were entered into queues of infinite lengths at each station. The simulations used the 802.11g MAC, and it was assumed that all control and data frames were sent over ideal channels at the maximum link rate of 54 Mbps. Please refer to the appendices for more detailed descriptions of our code.

The MAC parameters shown in table 2 were used in all of our simulations. All the parameters marked with an asterisk (*) were drawn from the 802.11 standard [1]; those with two asterisks came from [1] and the HCF group's recommendation [17]. Except for the contention window sizes, the rest of the variables came from [8]. The contention window sizes were chosen such that the highest priority access category had priority over DCF. In addition the contention window sizes were chosen such that each access category had at least two opportunities to back-off prior to attaining its maximum contention window size.

Table 2: MAC Parameters used for
EDCF Simulations

802.11 Parameter Name	Value
CW_{min}[0]	15
CW_{max}[0]	63
CW_{min}[1]	31
CW_{max}[1]	127
CW_{min}[2]	63
CW_{max}[2]	1023
aSlotTime*	20 μs
aSIFSTime*	10 μs
aDIFSTime*	50 μs
aAIFSTime[0]**	50 μs
aAIFSTime[1]**	70 μs
aAIFSTime[2]**	90 μs
aMPDUMaxLength*	4095 octets
dot11LongRetryLimit*	7
dot11MSDULifeTime[0]	60 ms
dot11MSDULifeTime[1]	100 ms
dot11MSDULifeTime[2]	200 ms

Table 2. -- Continued

802.11 Parameter Name	Value
MAC Header Length*	34 octets
PHY Header Length*	24 octets
ACK Length*	14 octets

Results of Basic EDCF Simulation

The goal of our next simulation exercise was to validate our EDCF model by verifying that system response times increased dramatically as the network became saturated. It is worth noting that in early versions of our simulations, the frame response times tended to approach infinity (over 25 seconds) when frames did not have a lifetime limit. Once this parameter was introduced, frame delays were limited. In our simulations frames could only be dropped if the number of transmission attempts for that frame exceeded the frame retry limit, or if the frame in question had been in the queue for longer than dot11MSDULifetime defined for that source. Figures 7, 8 and 9 display the results of our EDCF simulations.

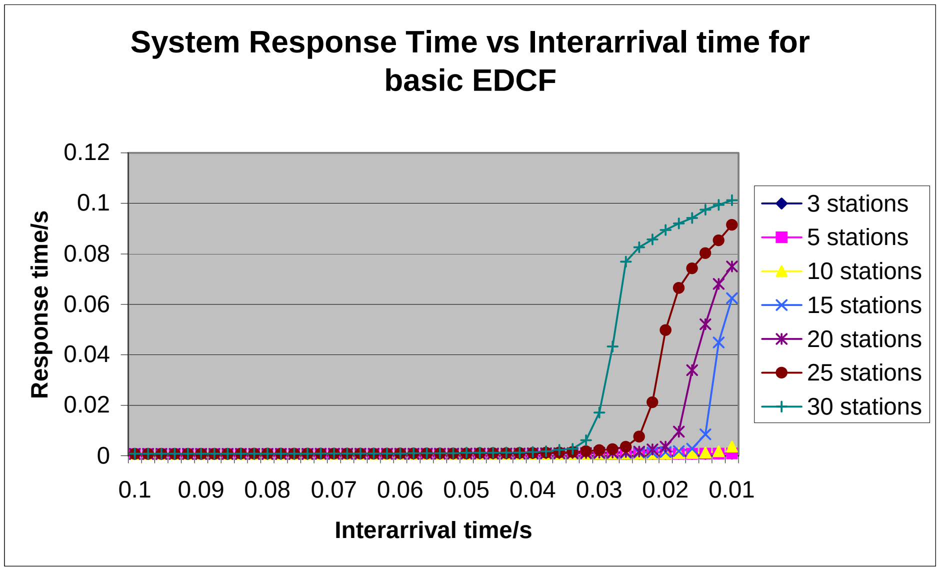

Figure 7: System Response Time for
Basic EDCF

Figure 8 shows how EDCF provides different service to different traffic classes. The response times under EDCF are in stark contrast to the response times under DCF where the class response times were identical across classes.

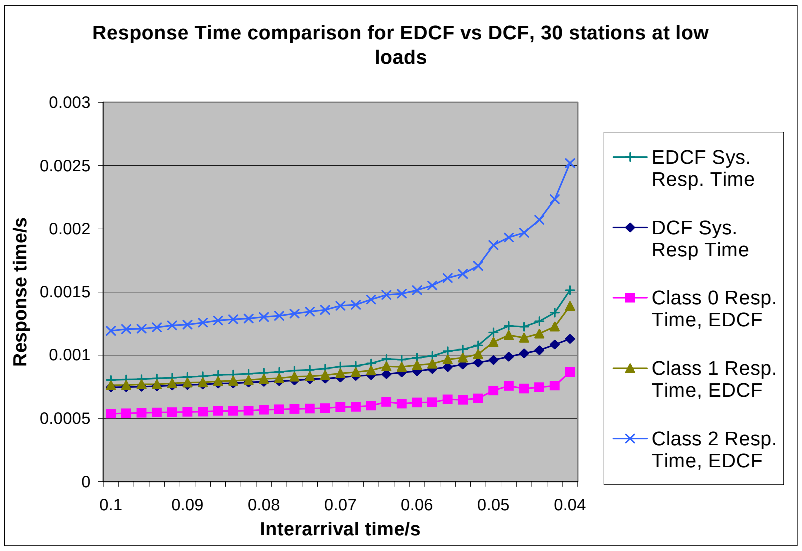

Figure 8: Response Time Comparison
for EDCF at Low Loads

It should be observed that EDCF actually has worse system response time than DCF. This is because EDCF defines separate AIFS times for each AC, and since each AIFS is at least DIFS + k*aSlotTime; each frame is expected to wait for the medium to be idle for a longer interval under EDCF. The smaller values of CW_{min} may appear to offset this difference; however, under heavy loads high priority stations will tend to collide a lot more since these stations have a much smaller interval from which to select the backoff counter.

Figure 9 displays EDCF's poor performance under heavy loads. When studying this figure, one ought to recall that the maximum frame lifetime under DCF was 512 μs, while under EDCF the frames had much longer lifetimes. This difference partly explains the variation in the graphs, but the figure still shows that EDCF performs poorly under heavy loads.

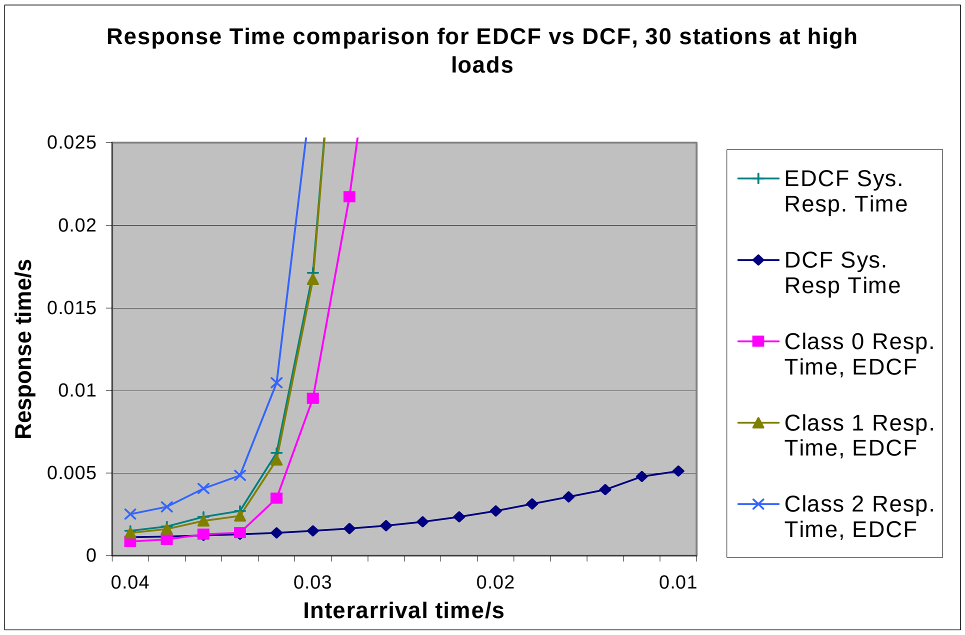

Figure 9: Comparison of EDCF and DCF

Our simulations also showed that the frame drop probabilities were highest for our lowest priority access category. This was in spite of the fact that this AC had the longest MSDU lifetime, and the largest values of CW_{min} and $CW_{max.}$ From figure 10, we can conclude that the frame dropping probability is affected by the probability that a frame can receive access to the medium. Since the higher priority AC has a much higher probability of getting access to the medium, there is also a lower probability that its frames will be dropped, even though the higher priority AC also has a much higher probability of collision at high loads.

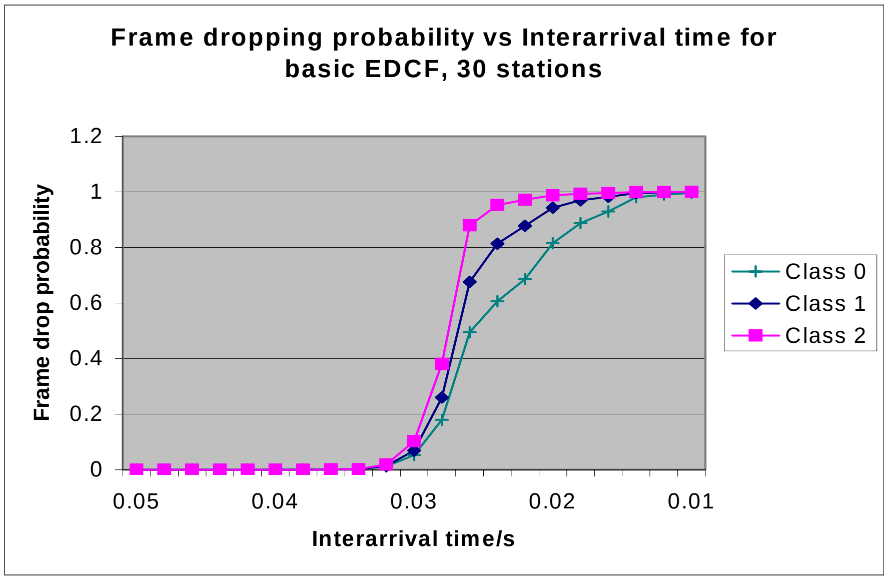

Figure 10: Frame Drop Probability for EDCF

Results of EDCF Simulation with Random Frame Dropping

In our next set of simulations we studied the effect of random frame dropping on EDCF performance. As was done for DCF, our frame dropping function had different frame dropping probabilities based on the number of frames in the queue. Our simulation program was written to drop 1 in 100 frames when the source's queue contained less than 5, 3, or 10 frames for sources with class 0, 1 or 2 traffic respectively. If the queue contained more than 5, 3 or 10 frames respectively the frame dropping probability was raised to 1 in 10. Figure 11 shows that, as was the case for DCF, random frame dropping does not appear to have any noticeable effect in reducing the delay for frames at low loads. However, at high loads with more than 30 competing stations random frame dropping reduces the system response time.

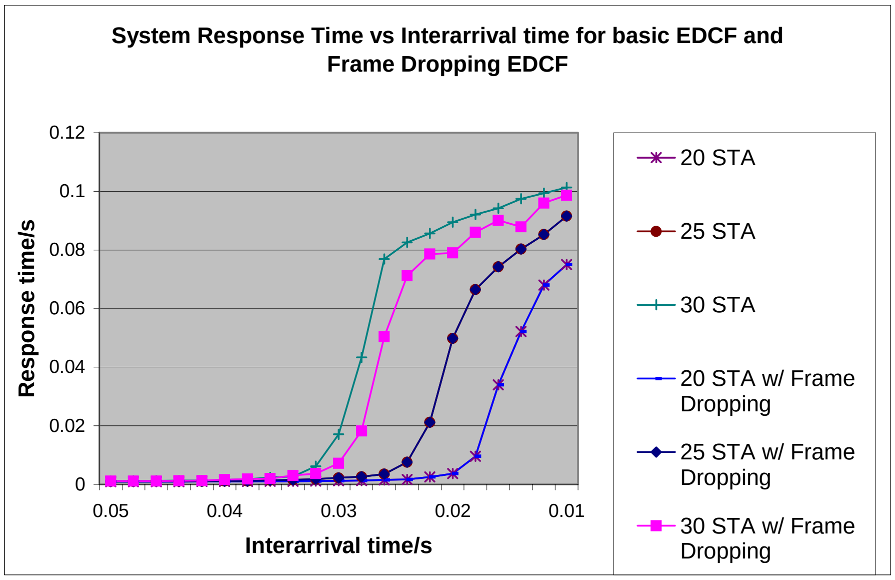

Figure 11: Comparison of System Response Times for Basic EDCF and EDCF with Random Frame Dropping

Results of EDCF Simulation with Varying Retry Limits

In our next set of simulations we studied the effect of different frame retry limits for the different sources on overall EDCF performance. All class 0 frames had a frame retry limit of three, all class 1 frames had a frame retry limit of five, while all class 2 frames had a frame retry limit of seven. Based on figure 12, we conclude that changing the frame retry limits for the different sources has no significant effect on the overall system response time for EDCF under saturation.

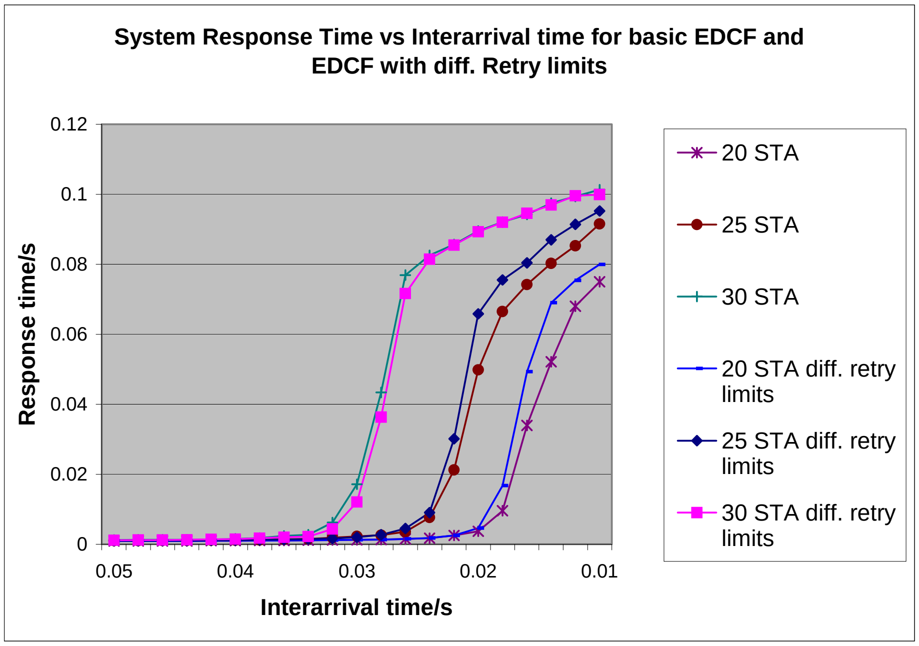

Figure 12: System Response Time
basic EDCF versus EDCF with
Different Frame Retry Limits

<u>AEDCF Simulation Approach</u>

In reference [16] Romdhani et al. stated that it was inefficient to reset the contention window to CW_{min} following the successful transmission of a frame. They argued that it was inefficient to reset the contention window as such because if a collision occurred, then there was a very high probability of another collision occurring soon afterwards. As a result, they proposed a scheme called Adaptive EDCF (AEDCF) that resets the contention window based on network conditions. In their scheme the CW is reduced using a station's history of collisions, denoted here as f_{avg}. In order to guard against transient changes in the collision probability, the authors propose using an exponentially weighted moving average (EWMA) to compute f_{avg}. f_{avg} is updated at regular intervals, T_{update}, as follows:

$$f_{avg}(j) = (1-\alpha) * f_{curr}(j) + \alpha * f_{avg}(j-1)$$

α is defined as the smoothing factor, and it determines the memory of the averaging process. Following the successful transmission of a frame under AEDCF, the new contention window, CW_{new}, is given by

$$MF[i] = \min ((1 + (i*2))* f_{avg} , 0.8)$$
$$CW_{new}[i] = \max (CW_{min}[i], CW_{old}[i] *MF[i])$$

MF[i], above, refers to the multiplicative factor used to reduce the size of the contention window, while 0.8 is a factor that was derived through simulation.

The authors of [16] state that their scheme requires four additional registers to store the values for α, MF[i], $f_{avg}(j-1)$ and T_{update}, the period that must elapse before the collision probability is computed.

In the next few sections we will present the research that was done on AEDCF, concluding with our extensions to AEDCF. Unlike Romdhani et al. our initial MAC parameters were as shown as defined in table 2. On the other hand our model was similar to theirs in that we updated the exponentially weighted collision probabilities at one-second intervals as in reference [16].

Results of Adaptive EDCF Simulation

In this set of simulations we studied the performance of adaptive EDCF (AEDCF) [16] compared with basic EDCF. Since the concept of the Persistence Factor was stricken from the EDCF draft, we also eliminated that factor from this set of simulations. Our simulation model also differed from reference [16] in that all our frame sources generated frames with roughly equal frame sizes and frame interarrival times. From figure 13 it is seen that adaptive EDCF does indeed outperform basic EDCF at heavy loads. This gain in performance shows that at heavy loads, the frame delay is actually dominated by the starting value of CW. Another way of reinterpreting this data might be to say that at heavy loads stations ought to have the right to pick a higher value for $CW_{min}[i]$, or perhaps the AP ought to broadcast new values of $CW_{min}[i]$to all stations once the collision probability exceeds a certain threshold.

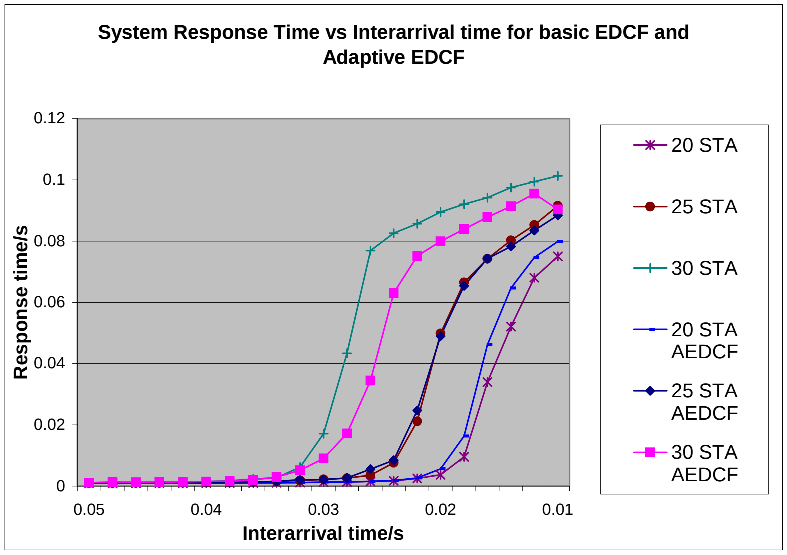

Figure 13: System Response Time
Basic EDCF versus Adaptive EDCF

In addition to reducing the frame delay under saturated conditions. Adaptive EDCF has the added benefit of reducing the frame drop probabilities throughout the system. Figure 14 shows how the frame drop probabilities were affected by AEDCF.

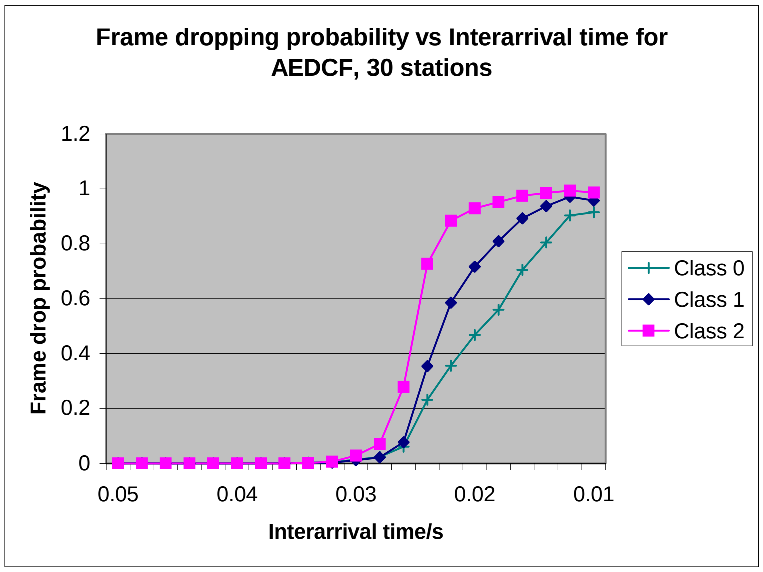

Figure 14: Frame Drop Probabilities for Adaptive EDCF

Results of Adaptive EDCF Simulation with Varying Persistence Factor

In our next set of simulations we studied the performance of adaptive EDCF (AEDCF) [16] while varying the persistence factor. We called this scheme AEDCF-PF. Recall that the EDCF draft specification document [17] indicated that the persistence factor was stricken from the 802.11e EDCF specification. However, in our simulations we reintroduced this parameter and allowed it to be either two or four, depending on the station's collision probability. The persistence factor, PF[i], was reset as follows:

$$f_{avg}(j) = (1-\alpha) * f_{curr}(j) + \alpha * f_{avg}(j-1)$$

```
If ($f_{avg}$ >β) then PF[i] = 4
Else PF[i]= 2
```

In these simulations, if any station experienced more than one collision in sixteen transmission attempts i.e. β was 0.0625, that station's persistence factor was set to 4. From figure 15, we see that this change does not result in a noticeable change in throughput, when the scheme was compared with AEDCF. Our research also indicates that this change did not affect the frame drop probabilities in a significant manner. This finding is reflected in figure 16.

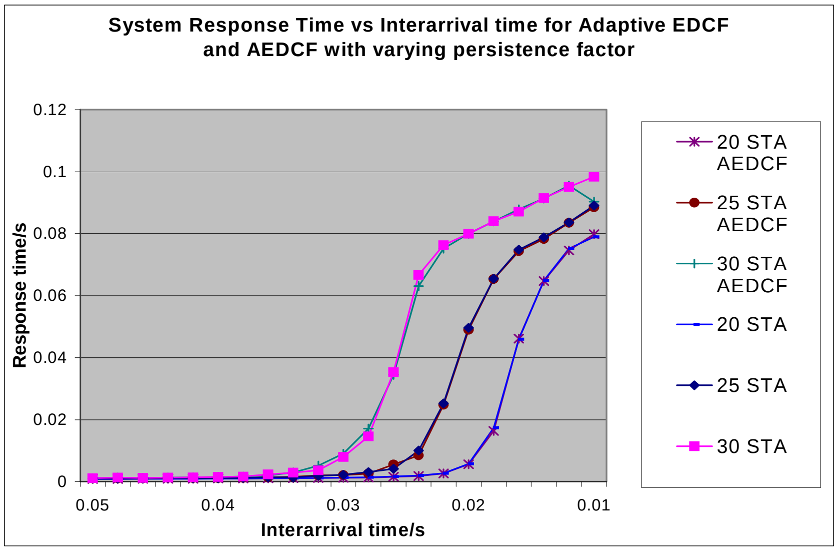

Figure 15: AEDCF with Varying Persistence Factor versus Basic AEDCF

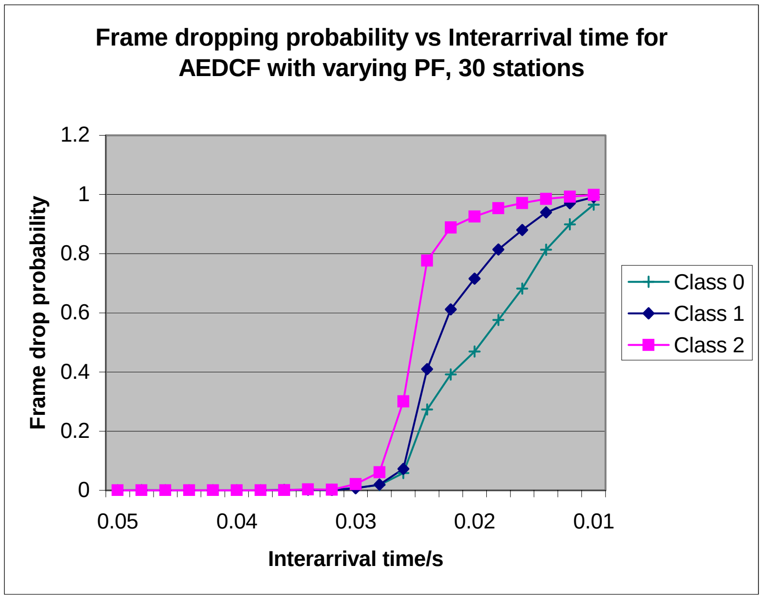

Figure 16: Frame Drop Probability for AEDCF with Varying PF

Results of Adaptive EDCF Simulation with Varying Persistence Factor and CW_{max}

In this section we present our AEDCF-CW/PF scheme that varies both the maximum contention window and the persistence factor. In simulating this scheme, each station was to check its average collision probability, $f_{avg}(j)$, prior to executing the exponential backoff algorithm. If $f_{avg}(j)$ was greater than some threshold value, γ, the station was allowed to increase $CW_{max}[i]$ temporarily, while the persistence factor was also increased as described for the AEDCF-PF scheme. Our algorithm operated as shown below:

```
$f_{avg}(j) = (1-\alpha) * f_{curr}(j) + \alpha * f_{avg}(j-1)$
$MF[i] = \min((1 + (i*2))* f_{avg}, 0.8)$
$CW_{new}[i] = \max(CW_{min}[i], CW_{old}[i] * MF[i])$
If ($f_{avg} > \beta$) then PF[i] = 4
Else PF[i]= 2
If ((a collision occurred) AND (($f_{avg} > \gamma$)) then
   If ((CW[i]+1) * PF [i]) <= (2 * ($CW_{max}$[i] + 1)) then
      $CW_{new}$[i] = (CW[i]+1) * PF [i]
   Endif
Else if ((a collision occurred) AND (($f_{avg} < \gamma$)) then
   If ((CW[i]+1) * PF [i]) <= $CW_{max}$[i]) then
      $CW_{new}$[i] = (CW[i]+1) * PF [i]
   Endif
```

In our simulations if a station experienced more than one collision in 32 transmission attempts i.e. γ was 0.03125 the maximum contention window was doubled. β, on the other hand, was set to 2* γ. After implementing these changes we found that our new scheme outperformed both basic EDCF and adaptive EDCF at high loads. Figures 17, 18 and 19 display these findings.

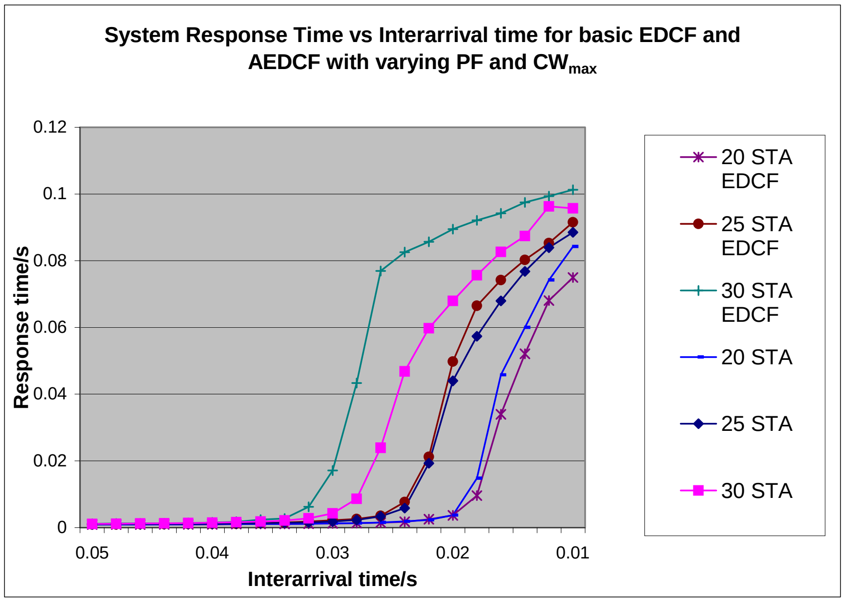

Figure 17: System Response Time
Basic EDCF versus AEDCF-CW/PF

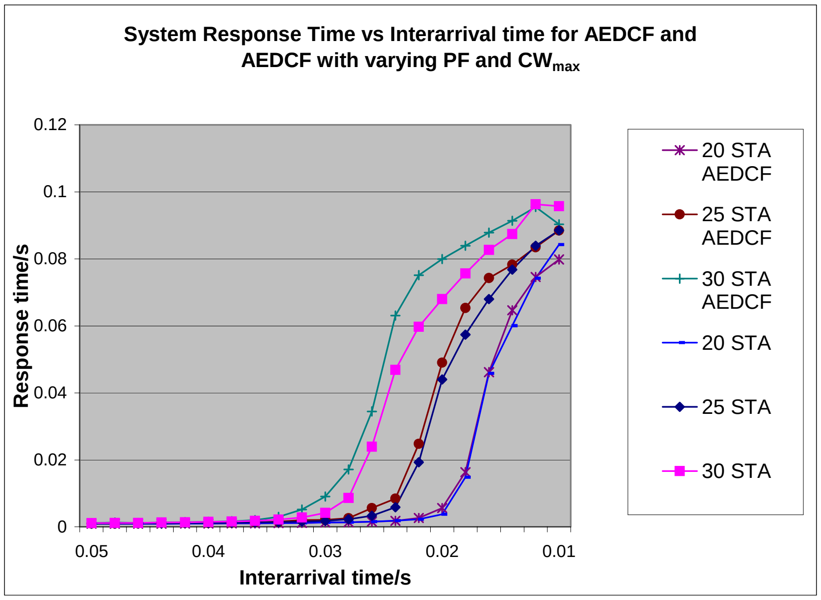

Figure 18: System Response Time
AEDCF versus AEDCF-CW/PF

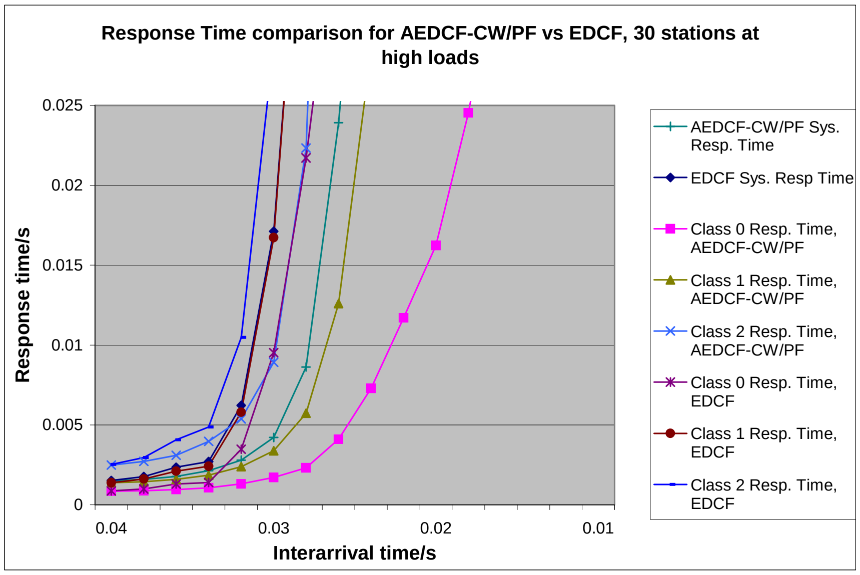

Figure 19: Response Time Comparison for Traffic Classes Under AEDCF-CW/PF

From figure 19 we conclude that AEDCF-CW/PF provides excellent response time to all traffic classes. This figure also shows that class 0 and class 1 traffic have a response time that is lower than the system response time for basic EDCF. This gain in performance is due to the combination of slowly reducing the contention windows, as well as the temporary increases in $CW_{max}[i]$.

At low loads AEDCF-CW/PF performs very similarly to EDCF, as shown in figure 20.

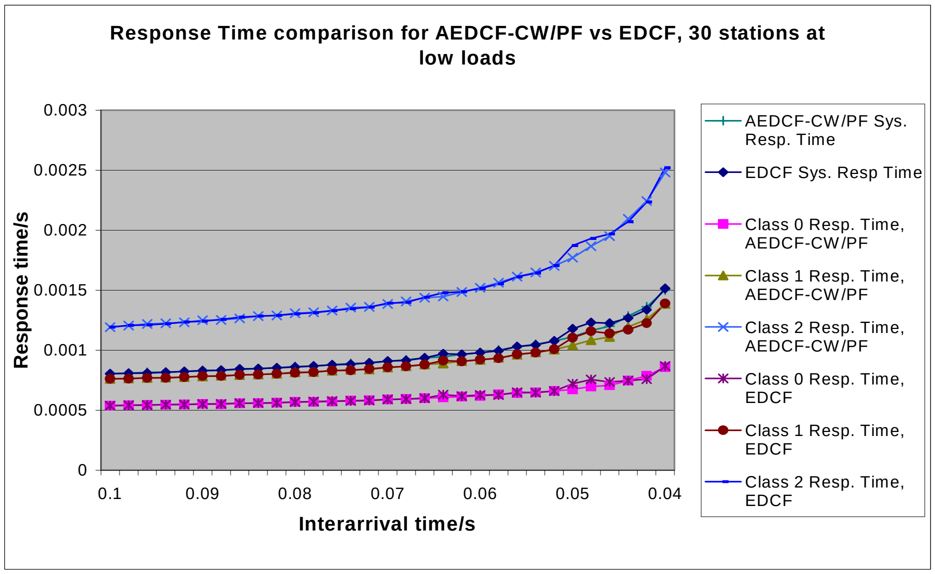

Figure 20: Comparison of AEDCF-CW/PF and EDCF at Low Loads

In addition to the new scheme's excellent response time performance under heavy loads, AEDCF-CW/PF also drops frames at a lower rate than both EDCF and AEDCF, as illustrated in figure 21. From this figure we see that at the highest load (0.01 ms between frame arrivals) 83% of the class 0 frames are dropped. This value was 99.6% for basic EDCF and 91.5% for AEDCF. Based on this, we can conclude that the highest priority traffic actually gains the most from the new scheme. It is worth noting, though, that the other traffic classes also gain from lower frame drop probabilities at the very high loads.

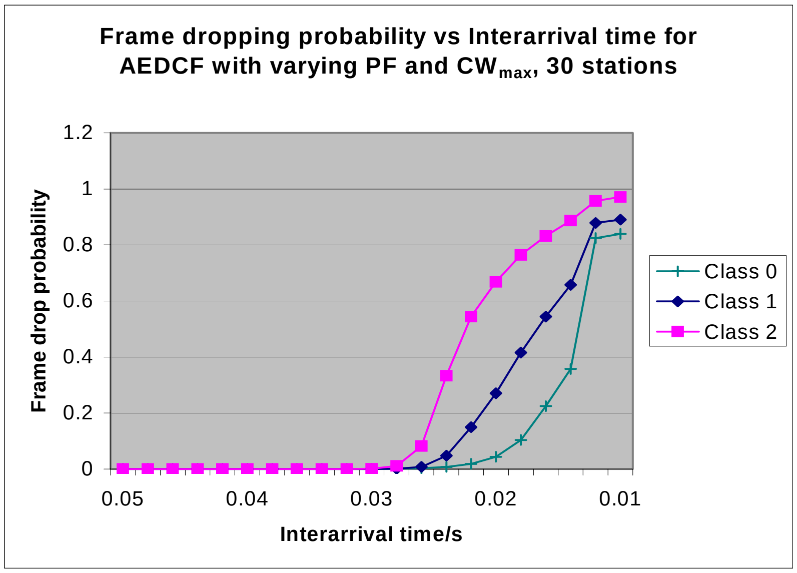

Figure 21: Frame Drop Probabilities
for AEDCF-CW/PF

AEDCF-CW/PF does have one flaw – it introduces higher jitter values for the traffic under heavy loads, as illustrated in figures 22 and 23. In our simulations, we measured jitter by collecting the variance of the system response time. Our data indicates that AEDCF-CW/PF starts out having comparable, or lower jitter values than both EDCF and AEDCF. Once each of these medium access functions attains its maximum throughput, AEDCF-CW/PF begins to display poor jitter performance. This degradation in performance is a result of the lower tendency of AEDCF-CW/PF to drop frames. Since more frames get transmitted, these frames then adversely affect the variance of the system response time. In spite of this flaw, we are of the opinion that AEDCF-CW/PF is superior to both EDCF and AEDCF since it has much better response times, and lower frame dropping probabilities.

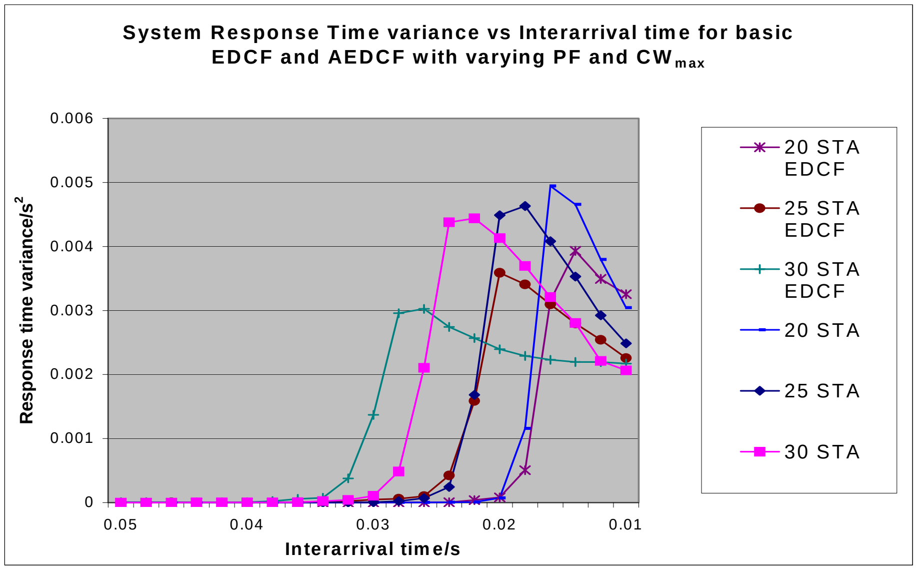

Figure 22: System Response Time Variance for AEDCF-CW/PF and EDCF

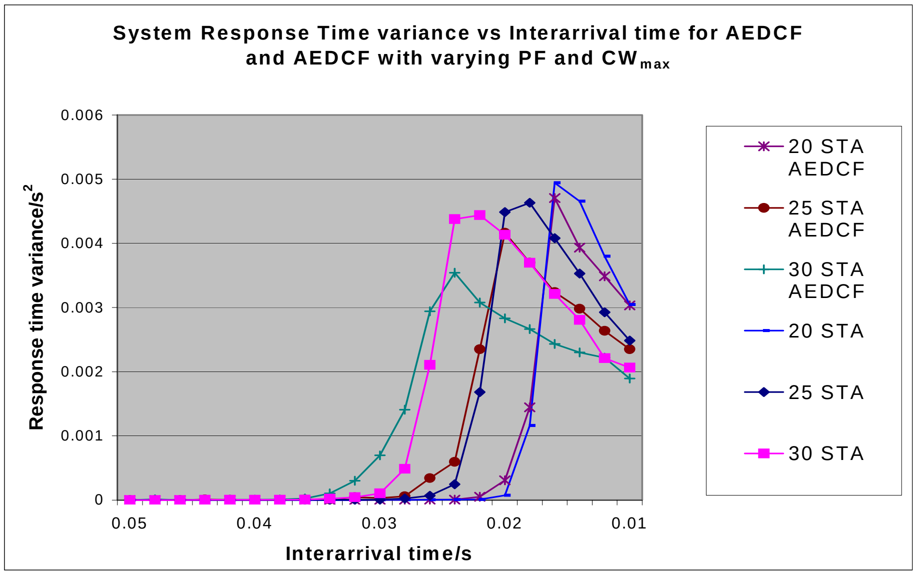

Figure 23: System Response Time Variance AEDCF and AEDCF-CW/PF

Impact of Changes

Adaptive EDCF with varying CW and PF can be implemented at each station with minimal impact. In addition to the four additional registers mentioned in [16] to store the values for α, PF[i], f^{j-1}_{avg} and T_{update}, our scheme will require two additional registers to store the values of β and γ. Computing the new persistence factor will take one comparison operation. In the event of a collision, determining whether or not to double CW_{max} will take two comparisons. Choosing to double CW_{max} will take one addition and one multiplication operation. The other multiplication and addition operations are a normal part of the standard EDCF procedure, and therefore do not add to the complexity of the function.

In this chapter we studied DCF and EDCF culminating in the introduction of a scheme called AEDCF-CW/PF that is a marked improvement over EDCF under heavy loads. Figure 24 shows the improvement in system response time provided by AEDCF-CW/PF.

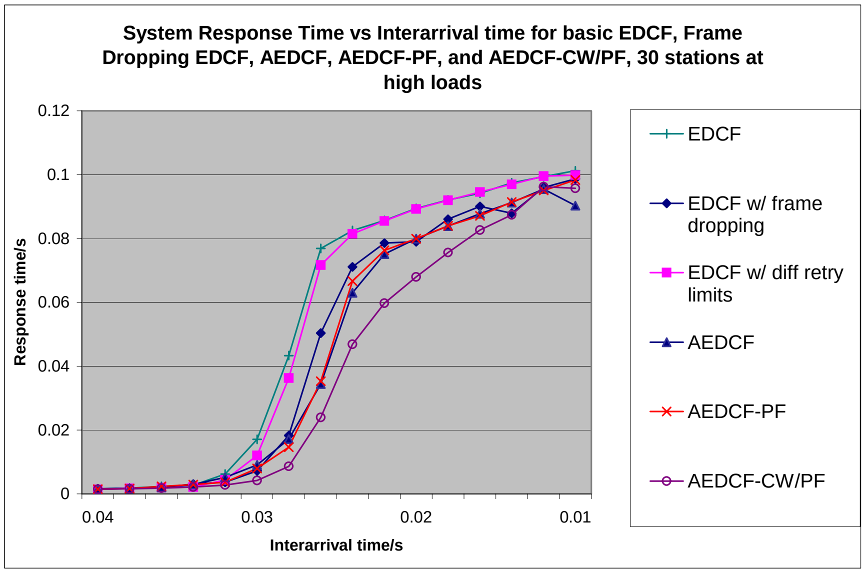

Figure 24: System Response Time for EDCF, AEDCF and Their Variants

CHAPTER 6

CONCLUSION

In the last chapter we presented a variety of techniques to provide quality of service in 802.11e LANs. Based on our findings random frame dropping from DCF queues does not appear to affect the frame delay for different traffic classes significantly. As a result we concluded that random frame dropping from DCF queues would have no effect on quality of service.

Next, we investigated the EDCF function and many of its variants. In our research we discovered that while EDCF provided prioritization for traffic classes, this came at the cost of a higher collision probability for stations as figure 25 shows.

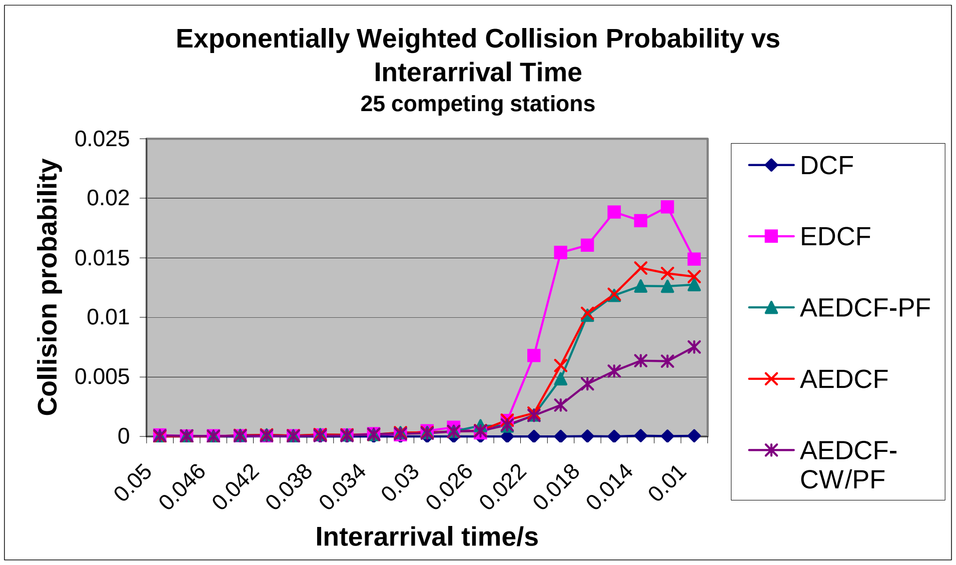

Figure 25: Collision Probability for DCF, EDCF and AEDCF

From figure 25 we conclude that the collision probability in an 802.11 WLAN is independent of the frame interarrival time as long as the WLAN is unsaturated. As the network approaches the saturation point, the probability of collision increases. This is because as we approach the saturation point more stations always have a frame to transmit; therefore, the network has a higher number of competing stations at any given instant.

It should be observed from figure 25 that EDCF has a much higher collision probability than both DCF and all the AEDCF variants. The difference in the curves can be attributed to the following factors. First, under DCF all the stations have the same starting values for CW_{min} and DIFS, therefore the stations all have a roughly equal chance of colliding with each other. EDCF on the other hand has at least four values of CW_{min} each with an associated AIFS[i] time. Since the AIFS times are longer for the lower priority AC, there is a much higher probability that these AC will collide with a higher priority AC that has a shorter AIFS time and smaller value for its contention window. Please note the drop in the collision probability for EDCF when the frame interarrival time is 0.01 s. This drop is a result of the much higher frame dropping probability at that rate. At that interarrival time several frames are being dropped from the queues because the frames have exceeded their MSDU lifetime limits. As a result the stations are no longer operating in saturated conditions. Finally we observe that AEDCF has a much lower collision probability than EDCF. This is a result of AEDCF's tendency to reset the contention window to CW_{min} at a lower rate than EDCF. In [16] Romdhani et al. stated that whenever a collision occurred in a network, another collision was very likely to occur in the near future. By resetting the CW slowly, one reduces the probability of two successive collisions.

As was the case for DCF our research indicated that random frame dropping from EDCF queues did not make a significant contribution to minimizing the frame delay, particularly when the number of stations is less than 30. As a result, random frame dropping from EDCF queues is not an attractive option for providing quality of service.

We also saw that setting up different frame retry limits for the different AC sometimes resulted in lowering the system response time. However, differentiating between frame retry limits for the different AC is not a viable method for providing quality of service, especially under saturated conditions.

The best method of providing quality of service in an 802.11e LAN would be to implement Adaptive EDCF (AEDCF), or one of its variants i.e. AEDCF with a varying persistence factor (AEDCF-PF) or AEDCF with a varying persistence factor and CW_{max} (AEDCF-PF/CW). Any of these algorithms can help a station deal with temporary instability in the WLAN caused by saturation, plus they have the benefit of being fully distributed. As a result QSTAs can compute values for $CW_{min}[i]$ and AIFS[i] without having to rely on a HC to broadcast these values.

In light of the advantages offered by AEDCF and its variants, it is our recommendation that any attempts to offer quality of service in 802.11e LANs focus on CW_{min}, and CW_{max}. From chapter 3, we found that 802.11e had a well-documented method of setting the arbitration interframe space i.e. aDIFSTime + k*aSlotTime, $k \geq 0$; therefore, we cannot modify these values at each station. Based on our simulations it is our recommendation that the persistence factor be reinstated in the 802.11e standard. This factor can aid in providing differentiation under heavy loads.

Our work in this thesis may be extended to investigate the effects of starting the AEDCF algorithm with different values for CW_{min} and CW_{max}. In addition, AEDCF's performance needs to be investigated when the group ACK mechanism is used. The AEDCF algorithm may be extended to adapt the packet burst length based on the number of acknowledgements that are received [16]. Finally, some research needs to be done in making the 802.11e MAC fairer. By incorporating the use of different sizes for CW_{min}, CW_{max} and AIFS the 802.11e MAC has become less fair than DCF. One approach to alleviating the unfairness of the 802.11e MAC might be to allow stations to use a given TXOP to transmit data from the queue that won access to the medium as well as another queue other than the queue that originally won access to the medium. The second frame that is transmitted may be chosen by choosing a frame from a queue that has the largest queue length to AIFS length ratio.

APPENDIX A

DCF CODE LISTING

```
/* Code to simulate DCF function */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin      31          //Number of slots
#define aCWmax    1023          //Number of slots
#define aSlotTime   0.00002     //time in seconds. 20 microsecs
#define aSIFSTime   0.00001     //time in seconds. 10 microsecs
#define aPIFSTime   0.00004     //time in seconds. 40 microsecs
#define aDIFSTime   0.00005     /*time in seconds. DIFS = SIFS
+ (2*aSlotTime) */
#define aMPDUMaxLength 4095     /*MPDU in octets.  According to
the standard 1 <= x <= 4095 */
#define dot11RetryLimit 7       /*Number of times a frame may
be retransmitted */
#define dot11MaxTransmitMsduLifetime 0.000512 /*Time after
initial tx when frame is dropped */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0  // 54 Mbps
#define NARS  100000            //Number of arrivals
#define STNS    20

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;   //Event that the medium is idle after aDIFSTime
EVENT busy;   //Event that the medium is currently busy
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;     //Event that a function can continue execution

int cnt;
int xmitting; //Nbr of stations transmitting simultaneously
int j;
double IATM;  //Frame inter-arrival times

void init();
float GenerateFrame();    //Generates frames
int ExpBackoff(int CW, int PF); /*Exponential Backoff
algorithm */
void BackOffCtr(int w);   //Counts down idle slots
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k); //Processes frames
void cust(int i);  /*Generates and processes the frame */
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void ACKgenerator(); /*Generates ACK iff only one station is
transmitting */

void sim()                      //1st process - named sim
{
 int stdone;
 FILE *out;
 char filename[16];

 sprintf(filename, "outdcf%d.txt", STNS);
 set_model_name("Simulation of DCF");
 out = fopen(filename, "wt");
 for (IATM = 0.1000; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");                  //required create statement

  max_processes(100000000);
  max_events(1000000);
  init();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
  clear(ACK);
  clear(idle);
  clear(busy);
  clear(finxmission);
  clear(go);
  xmitting = 0;
  reset();

  cnt = 3*NARS;
  MediumSensing();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                           //wait until all done
  fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
               %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
             ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
             ,table_var(tbl),table_mean(rtxtbl)
             ,table_mean(cwtbl),table_mean(drptbl)
             ,table_mean(tbl1),table_var(tbl1)
             ,table_mean(rtxtbl1),table_mean(cwtbl1)
             ,table_mean(drptbl1),table_mean(tbl2)
             ,table_var(tbl2),table_mean(rtxtbl2)
             ,table_mean(cwtbl2),table_mean(drptbl2)
             ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
   report();                              //print report
   mdlstat();
}

void init()
{
   s = facility("medium");            //initialize facility
 facility_set(queues,"src_qs",STNS);

 done = event("done");                //initialize event
 ACK = event("ACK");
 idle = event("idle");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");

 qtbl = qhistogram("num from source0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source1", 10l);
 qtbl2 = qhistogram("num from source2", 10l);
 xmitting = 0;
}

float GenerateFrame()    //Generates frames
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}

int ExpBackoff(int CW, int PF) /*Exponential Back-off
algorithm */
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= aCWmax)
  return(tempCW);
 else
  return (CW);
}

void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)    /*Processes
frames */
{
    double t1;
    int    st;
 int    retx=0;     /*keeps track of how many times a
retransmission occurs */
 int    w;          //used to keep track of number of slots
 int    CW;         /*used to keep track of the size of the
contention window*/

 CW = aCWmin;
 t1 = clock;        //time of request
 reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock-t1)<dot11MaxTransmitMsduLifetime)
  {
   wait(idle);    //Wait until the medium is idle for
aDIFSTime */
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;     /*Increment the number of stations
transmitting */
   set(busy);
   use(s,svc);     //reserve medium
   clear(busy);
   record(clock-t1, tbl);      /*record response time for this
source */
   record(clock-t1, tbl3);     /*record response time for all
sources */
   set(finxmission);   /*Indicate that the transmission is
done */
   wait(go);     /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;     /*Decrement the number of stations
transmitting */
   clear(go);      /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //If an ACK has not been received
   while (st!=1 && retx<dot11RetryLimit && (clock-
t1)<dot11MaxTransmitMsduLifetime)
   {
    CW = ExpBackoff(CW, 2);
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;    /*Increment the number of stations
transmitting */
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
```

```
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;    /*Decrement the number of stations
transmitting */
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl);  //record number of retransmissions
   record(w, cwtbl);   //Record size of contention window
   if (st !=1 && (retx == dot11RetryLimit || (clock-
t1)>=dot11MaxTransmitMsduLifetime))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that this frame was dropped
 release(q[k]);
}

void cust(int i)     //process customer
{
 char procname[32];
 sprintf(procname, "cust %d", i);
 create(procname);               //required create statement
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);           //note arrival
  ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  ProcessFrame(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i); //Processes frames
  note_exit(qtbl1);
 }
 else
 {
  note_entry(qtbl2);
  ProcessFrame(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
        note_exit(qtbl2);   //note departure
 }
    cnt--;
    if(cnt == 0)
        set(done);     //if last arrival, signal
}

void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
        cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}
void MediumSensing()    //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
            hold(aDIFSTime);  //Wait for a DIFS interval
   st = state(busy);
   if (st==2)
    set(idle);   /*If the medium is free for a DIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}
```

```
void ACKgenerator()     /*Determines whether or not an ACK can
be sent */
{
 create("ACKgenerator");
 set_priority(2);
 int stdone;
 double ACKsize = ACKLgth+ PHYHdr;
 double svc = ACKsize/LinkSpeed;
 record(ACKsize,frmtbl);
 do
 {
  wait(finxmission);
  if (xmitting==1 || xmitting==0)  /*Generate an ACK iff one
station is transmitting */
  {
   hold(aSIFSTime);
   set(busy);
   use(s, svc);   //Transmit the ACK frame
   clear(busy);
   set(ACK);
   set(go);
  }
  else
  {
   clear(ACK);
   hold(aSIFSTime);
   set(go);
  }
  stdone = state(done);
 }
 while (stdone == 2);
}
```

APPENDIX B

DCF WITH RANDOM FRAME DROPPING

CODE LISTING

```
/* Code to simulate DCF with random frame dropping function */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin      31         //Number of slots
#define aCWmax    1023         //Number of slots
#define aSlotTime   0.00002    //time in seconds. 20 microsecs
#define aSIFSTime   0.00001    //time in seconds. 10 microsecs
#define aPIFSTime   0.00004    //time in seconds. 40 microsecs
#define aDIFSTime   0.00005    /*time in seconds. DIFS = SIFS
+ (2*aSlotTime) */
#define aMPDUMaxLength 4095    /*MPDU in octets.  According to
the standard 1 <= x <= 4095 */
#define dot11RetryLimit 7      /*Number of times a frame may
be retransmitted */
#define dot11MaxTransmitMsduLifetime 0.000512 /*Time after
initial tx when frame is dropped */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0  // 54 Mbps
#define NARS  100000           //Number of arrivals
#define STNS    20

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;    //Event that the medium is idle after aDIFSTime
EVENT busy;    //Event that the medium is currently busy
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;      //Event that a function can continue execution

int cnt;
int xmitting;  /*Nbr of stations transmitting simultaneously
int j;
double IATM;   //Frame inter-arrival times

void init();
float GenerateFrame();    //Generates frames
int ExpBackoff(int CW, int PF); /*Exponential Back-off
algorithm */
void BackOffCtr(int w);   //Counts down idle slots
void DropFrame(FACILITY q[], TABLE drptbl, MBOX mb, int i);
 //Drops frames
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void cust(int i);  /*Generates and processes the frame */
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void ACKgenerator(); /*Generates ACK iff only one station is
transmitting*/

void sim()                //1st process - named sim
{
 int stdone;
 FILE *out;
 char filename[16];

 sprintf(filename, "outdcfdrp%d.txt", STNS);
 set_model_name("Simulation of DCF w/ random frame dropping");
 out = fopen(filename, "wt");
 for (IATM = 0.1000; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");              //required create statement

  max_processes(100000000);
  max_events(1000000);
  max_mailboxes(1000000);
  init();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
  clear(ACK);
  clear(idle);
  clear(busy);
  clear(finxmission);
  clear(go);
  xmitting = 0;
  reset();

  cnt = 3*NARS;
  MediumSensing();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                      //wait until all done
  fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
               %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
              ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
              ,table_var(tbl),table_mean(rtxtbl)
              ,table_mean(cwtbl),table_mean(drptbl)
              ,table_mean(tbl1),table_var(tbl1)
              ,table_mean(rtxtbl1),table_mean(cwtbl1)
              ,table_mean(drptbl1),table_mean(tbl2)
              ,table_var(tbl2),table_mean(rtxtbl2)
              ,table_mean(cwtbl2),table_mean(drptbl2)
              ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
 report();                              //print report
 mdlstat();
}

void init()
{
 s = facility("medium");             //initialize facility
 facility_set(queues,"src_qs",STNS);

 done = event("done");                //initialize event
 ACK = event("ACK");
 idle = event("idle");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");

 qtbl = qhistogram("num from source0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source1", 10l);
 qtbl2 = qhistogram("num from source2", 10l);
 xmitting = 0;
}

float GenerateFrame()    //Generates frames
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}

int ExpBackoff(int CW, int PF) /*Exponential Back-off
algorithm*/
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= aCWmax)
  return(tempCW);
 else
  return (CW);
}

void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}

void DropFrame(FACILITY q[], TABLE drptbl, MBOX mb, int i)
 //Drops frames
{
 MBOX recv;
 MBOX temp;
 long buffer=0;
 long* msg;

 recv = mailbox("rcv");
 temp = mailbox("tmp");
 reserve(q[i]);     /*Enter this frame in the queue for the
source being processed */
  record(1.0, drptbl); //Record that this frame was dropped
  send(mb, buffer);   /*Indicate to the source the message was
dropped */
  receive(mb, (long *) &msg);
 release(q[i]);
 delete_mailbox(temp);
 delete_mailbox(recv);
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)   //Processes frames
{
   double t1;
   int    st;
   int    retx=0;     /*keeps track of how many times a
retransmission occurs */
   int    w;          //used to keep track of number of slots
   int    CW;      /*used to keep track of the size of the
contention window */

    CW = aCWmin;
    t1 = clock;           //time of request
    reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock-t1)<dot11MaxTransmitMsduLifetime)
  {
   wait(idle);     /*Wait until the medium is idle for
aDIFSTime */
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;
   set(busy);
   use(s,svc);     //reserve medium
   clear(busy);
   record(clock-t1, tbl);      /*record response time for this
source */
   record(clock-t1, tbl3);     /*record response time for all
sources */
   set(finxmission);   /*Indicate that the transmission is
done */
   wait(go);     /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);     /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //If an ACK has not been received
   while (st!=1 && retx<dot11RetryLimit && (clock-
t1)<dot11MaxTransmitMsduLifetime)
   {
    CW = ExpBackoff(CW, 2);
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); /*record response time for all
sources */
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl);  //record number of retransmissions
   record(w, cwtbl);   //Record size of contention window
   if (st !=1 && (retx == dot11RetryLimit || (clock-
t1)>=dot11MaxTransmitMsduLifetime))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); /*Record that this frame wasn't
dropped */
   }
  }
  else
   record(1.0, drptbl); //Record that this frame was dropped
 release(q[k]);
}

void cust(int i)     //process customer
{
 char procname[32];
 MBOX mb;
 sprintf(procname, "cust %d", i);
 create(procname);    //required create statement
 mb = mailbox("mb");    /*Mailbox to receive messages on
dropped frames */
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);           //note arrival
  if (qlength(queues[i]) < 5)
  {
   if (random(0,999)%501 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
  else
  {
   if (random(0,99)%51 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  if (qlength(queues[i]) < 3)
  {
   if (random(0,999)%502 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
  else
  {
   if (random(0,99)%52 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
  note_exit(qtbl1);
 }
 else
 {
  note_entry(qtbl2);
  if (qlength(queues[i]) < 10)
  {
   if (random(0,999)%503 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
  else
  {
   if (random(0,99)%53 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
        note_exit(qtbl2);   //note departure
 }
 delete_mailbox(mb);
    if(cnt == 0)
        set(done);     //if last arrival, signal
}

void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
        cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}
void MediumSensing()    //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aDIFSTime);  //Wait for a DIFS interval
   st = state(busy);
   if (st==2)
    set(idle);   /*If the medium is free for a DIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void ACKgenerator()     /*Determines whether or not an ACK can
be sent */
{
 create("ACKgenerator");
 set_priority(2);
 int stdone;
 double ACKsize = ACKLgth+ PHYHdr;
 double svc = ACKsize/LinkSpeed;
 record(ACKsize,frmtbl);
 do
 {
  wait(finxmission);
  if (xmitting==1 || xmitting==0)   /*Generate an ACK iff one
station is transmitting */
  {
   hold(aSIFSTime);
   set(busy);
   use(s, svc);   //Transmit the ACK frame
   clear(busy);
   set(ACK);
   set(go);
  }
  else
  {
   clear(ACK);
   hold(aSIFSTime);
   set(go);
  }
  stdone = state(done);
 }
 while (stdone == 2);
}
```

APPENDIX C

EDCF CODE LISTING

```
/* Code to simulate EDCF function */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin       15      //Number of slots
#define aCWmax       63      //Number of slots
#define aCWmin1      31      //Number of slots
#define aCWmax1     127      //Number of slots
#define aCWmin2      63      //Number of slots
#define aCWmax2    1023      //Number of slots
#define aSlotTime   0.00002  //time in seconds. 20 microsecs
#define aSIFSTime   0.00001  //time in seconds. 10 microsecs
#define aPIFSTime   0.00004  //time in seconds. 40 microsecs
#define aDIFSTime   0.00005  /*time in seconds. DIFS = SIFS +
(2*aSlotTime) */
#define aAIFSTime   0.00005  //time in seconds. 50 microsecs
#define aAIFSTime1  0.00007  //time in seconds. 70 microsecs
#define aAIFSTime2  0.00009  //time in seconds. 90 microsecs
#define aMPDUMaxLength 4095  /*MPDU in bits.  According to the
standard 1 <= x <= 4095 */
#define dot11RetryLimit 7    /*Number of times a frame may be
retransmitted */
#define dot11MSDULifeTime 0.06  /*Amount of time that a source
0 frame can be alive */
#define dot11MSDULifeTime1 0.1  /*Amount of time that a source
1 frame can be alive */
#define dot11MSDULifeTime2 0.2  /*Amount of time that a source
2 frame can be alive */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0   // 54 Mbps
#define NARS  100000            //Number of arrivals
#define STNS    20

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;   //Event that the medium is idle after aAIFSTime
EVENT idle1;  //Event that the medium is idle after aAIFSTime1
EVENT idle2;  //Event that the medium is idle after aAIFSTime2
EVENT busy;   //Event that the medium is currently busy
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;     //Event that a function can continue execution

int cnt;
int xmitting;  //Nbr of stations transmitting simultaneously
int j;
double IATM;   //Frame inter-arrival times

void init();   //Initializes structures used in the simulation
float GenerateFrame();  //Generates frames
int ExpBackoff(int CW, int PF, int amaxCW); /*Exponential
Back-off algorithm */
void BackOffCtr(int w);  /*Counts down the number of slots in
the backoff interval */
void BackOffCtr1(int w); /*Counts down the number of slots in
the backoff interval */
void BackOffCtr2(int w); /*Counts down the number of slots in
the backoff interval */
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void cust(int i);  /*Generates and processes the frame */
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void MediumSensing1(); //Medium Sensing function
void MediumSensing2(); //Medium Sensing function
void ACKgenerator(); /*Generates ACK iff only one station is
transmitting.*/

void sim()                    //1st process - named sim
```

```
{
 int stdone;
 FILE *out;
 char filename[16];

    sprintf(filename, "outedcf%d.txt", STNS);
    set_model_name("Simulation of EDCF");
 out = fopen(filename, "wt");
 for (IATM = 0.1000; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");                //required create statement

  max_processes(100000000);
  max_events(10000000);
  init();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
        clear(ACK);
  clear(idle);
  clear(idle1);
  clear(idle2);
  clear(busy);
  clear(finxmission);
  clear(go);
  xmitting = 0;
  reset();

  cnt = 3*NARS;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                        //wait until all done
  fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
               %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
              ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
              ,table_var(tbl),table_mean(rtxtbl)
              ,table_mean(cwtbl),table_mean(drptbl)
              ,table_mean(tbl1),table_var(tbl1)
              ,table_mean(rtxtbl1),table_mean(cwtbl1)
              ,table_mean(drptbl1),table_mean(tbl2)
              ,table_var(tbl2),table_mean(rtxtbl2)
              ,table_mean(cwtbl2),table_mean(drptbl2)
              ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
   report();                                //print report
   mdlstat();
}

void init()
{
 s = facility("medium");               //initialize facility
 facility_set(queues,"src_queues",STNS);

 done = event("done");                 //initialize event
 ACK = event("ACK");
 idle = event("idle");
 idle1 = event("idle1");
 idle2 = event("idle2");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");

 qtbl = qhistogram("num from source 0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source 1", 10l);
 qtbl2 = qhistogram("num from source 2", 10l);
 xmitting = 0;
}

float GenerateFrame()    //Generates frames
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}

int ExpBackoff(int CW, int PF, int amaxCW) /*Exponential Back-
off algorithm */
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= amaxCW)
  return(tempCW);
 else
  return (CW);
}

void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}

void BackOffCtr1(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle1);
 }
}

void BackOffCtr2(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle2);
 }
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin;
    t1 = clock;        //time of request
    reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime)
  {
   wait(idle);   //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;    /*Increment the number of stations
transmitting */
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);  /*record response time for all
sources */
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;    /*Decrement the number of stations
transmitting */
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime)
   {
    CW = ExpBackoff(CW, 2, aCWmax);
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;    /*Increment the number of stations 
transmitting */
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this 
source */
    record(clock-t1, tbl3); /*record response time for all 
sources */
    set(finxmission);  /*Indicate that the transmission is 
done */
    wait(go);    /*Wait for the ACKgenerator function to check 
xmitting */
    xmitting--;    /*Decrement the number of stations 
transmitting */
    clear(go);    /*Once go has been received continue 
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >= 
dot11MSDULifeTime && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE 
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes 
frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a 
retransmission occurs */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of contention 
window size */

    CW = aCWmin1;
    t1 = clock;        //time of request
    reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime1)
  {
   wait(idle1);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr1(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);  /*record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime)
   {
    CW = ExpBackoff(CW, 2, aCWmax1);
    wait(idle1);
    w = rand()%CW;
    BackOffCtr1(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); /*record response time for all
sources */
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime1 && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that this frame was dropped
 release(q[k]);
}

void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin2;
    t1 = clock;        //time of request
    reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime)
  {
   wait(idle2);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr2(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);  /*record system response time */
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime2)
   {
    CW = ExpBackoff(CW, 2, aCWmax2);
    wait(idle);
    w = rand()%CW;
    BackOffCtr2(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); /*record response time for all
sources */
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime2 && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that this frame was dropped
 release(q[k]);
}

void cust(int i)     //process customer
{
 char procname[32];
 sprintf(procname, "cust %d", i);
 create(procname);    //required create statement
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);   //note arrival
  ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  ProcessFrame1(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
  note_exit(qtbl1);
 }
 else
 {
  note_entry(qtbl2);
  ProcessFrame2(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
        note_exit(qtbl2);   //note departure
 }
    cnt--;
    if(cnt == 0)
        set(done);     //if last arrival, signal
}

void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
        cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing()    //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
```

```
            hold(aAIFSTime);  //Wait for an AIFS interval
   st = state(busy);
   if (st==2)
    set(idle);   /*If the medium is free for an AIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing1()    //Check state of medium
{
  create("MediumSensing1");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime1);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle1);   /*If the medium is free for a AIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing2()    //Check state of medium
{
  create("MediumSensing2");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime2);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle2);   /*If the medium is free for a AIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void ACKgenerator()     /*Determines whether or not an ACK can
be sent */
{
 create("ACKgenerator");
 set_priority(2);
 int stdone;
 double ACKsize = ACKLgth+ PHYHdr;
 double svc = ACKsize/LinkSpeed;
 record(ACKsize,frmtbl);
 do
 {
  wait(finxmission);
  if (xmitting==1 || xmitting==0)  /*Generate an ACK iff one
station is transmitting */
  {
   hold(aSIFSTime);
   set(busy);
   use(s, svc);   //Transmit the ACK frame
   clear(busy);
   set(ACK);
   set(go);
  }
  else
  {
   clear(ACK);
   hold(aSIFSTime);
   set(go);
  }
  stdone = state(done);
 }
 while (stdone == 2);
}
```

APPENDIX D

EDCF WITH RANDOM FRAME DROPPING CODE LISTING

```
/* Code to simulate EDCF function */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin       15      //Number of slots
#define aCWmax       63      //Number of slots
#define aCWmin1      31      //Number of slots
#define aCWmax1     127      //Number of slots
#define aCWmin2      63      //Number of slots
#define aCWmax2    1023      //Number of slots
#define aSlotTime   0.00002  //time in seconds. 20 microsecs
#define aSIFSTime   0.00001  //time in seconds. 10 microsecs
#define aPIFSTime   0.00004  //time in seconds. 40 microsecs
#define aDIFSTime   0.00005  /*time in seconds. DIFS = SIFS +
(2*aSlotTime) */
#define aAIFSTime   0.00005  //time in seconds. 50 microsecs
#define aAIFSTime1  0.00007  //time in seconds. 70 microsecs
#define aAIFSTime2  0.00009  //time in seconds. 90 microsecs
#define aMPDUMaxLength 4095  /*MPDU in bits.  According to the
standard 1 <= x <= 4095 */
#define dot11RetryLimit 7    /*Number of times a frame may be
retransmitted */
#define dot11MSDULifeTime 0.06  /*Amount of time that a source
0 frame can be alive */
#define dot11MSDULifeTime1 0.1  /*Amount of time that a source
1 frame can be alive */
#define dot11MSDULifeTime2 0.2  /*Amount of time that a source
2 frame can be alive */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0   // 54 Mbps
#define NARS  100000            //Number of arrivals
#define STNS    20

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;   //Event that the medium is idle after aAIFSTime
EVENT idle1;  //Event that the medium is idle after aAIFSTime1
EVENT idle2;  //Event that the medium is idle after aAIFSTime2
EVENT busy;   //Event that the medium is currently busy
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;     //Event that a function can continue execution

int cnt;
int xmitting; //Number of stations transmitting simultaneously
int j;
double IATM;  //Frame inter-arrival times

void init();   //Initializes structures used in the simulation
float GenerateFrame();  //Generates frames
int ExpBackoff(int CW, int PF, int amaxCW); /*Exponential
Back-off algorithm */
void BackOffCtr(int w);  /*Counts down the number of slots in
the backoff interval */
void BackOffCtr1(int w); /*Counts down the number of slots in
the backoff interval */
void BackOffCtr2(int w); /*Counts down the number of slots in
the backoff interval */
void DropFrame(FACILITY q[], TABLE drptbl, MBOX mb, int i);
 //Drops frames
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void cust(int i);  //Generates and processes the frame
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void MediumSensing1(); //Medium Sensing function
void MediumSensing2(); //Medium Sensing function
void ACKgenerator(); /*Generates ACK iff only one station is
transmitting.*/

void sim()                   //1st process - named sim
```

```
{
 int stdone;
 FILE *out;
 char filename[16];

 sprintf(filename, "outedcfdrp%d.txt", STNS);
 set_model_name("Simulation of EDCF w/ random frame
dropping");
 out = fopen(filename, "wt");
 for (IATM = 0.1000; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");        //required create statement

  max_processes(100000000);
  max_events(10000000);
  max_mailboxes(1000000);
  init();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
        clear(ACK);
  clear(idle);
  clear(idle1);
  clear(idle2);
  clear(busy);
  clear(finxmission);
  clear(go);
  xmitting = 0;
  reset();

  cnt = 3*NARS;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                    //wait until all done
  fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
                %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
              ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
              ,table_var(tbl),table_mean(rtxtbl)
              ,table_mean(cwtbl),table_mean(drptbl)
```

```
             ,table_mean(tbl1),table_var(tbl1)
             ,table_mean(rtxtbl1),table_mean(cwtbl1)
             ,table_mean(drptbl1),table_mean(tbl2)
             ,table_var(tbl2),table_mean(rtxtbl2)
             ,table_mean(cwtbl2),table_mean(drptbl2)
             ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
   report();                              //print report
   mdlstat();
}

void init()
{
 s = facility("medium");              //initialize facility
 facility_set(queues,"src_queues",STNS);

 done = event("done");                //initialize event
 ACK = event("ACK");
 idle = event("idle");
 idle1 = event("idle1");
 idle2 = event("idle2");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");

 qtbl = qhistogram("num from source 0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source 1", 10l);
 qtbl2 = qhistogram("num from source 2", 10l);
 xmitting = 0;
}

float GenerateFrame()    //Generates frames
```

```
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}

int ExpBackoff(int CW, int PF, int amaxCW) /*Exponential Back-
off algorithm */
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= amaxCW)
  return(tempCW);
 else
  return (CW);
}

void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}

void BackOffCtr1(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle1);
 }
}

void BackOffCtr2(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
```

```
   wait(idle2);
 }
}

void DropFrame(FACILITY q[], TABLE drptbl, MBOX mb, int i)
 //Drops frames
{
 MBOX recv;
 MBOX temp;
 long buffer=0;
 long* msg;

 recv = mailbox("rcv");
 temp = mailbox("tmp");
 reserve(q[i]);     //Enter this frame in the queue for the
source being processed
  record(1.0, drptbl);  //Record that this frame was dropped
  send(mb, buffer);   //Indicate to the source the message was
dropped
  receive(mb, (long *) &msg);
 release(q[i]);
 delete_mailbox(temp);
 delete_mailbox(recv);
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin;
    t1 = clock;                  //time of request
    reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime)
  {
   wait(idle);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3); /*record response time for all
sources */
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution*/
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime)
   {
    CW = ExpBackoff(CW, 2, aCWmax);
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime && st!=1))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); /*Record that frame wasn't dropped */
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
```

```
 release(q[k]);
}

void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs. */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin1;
    t1 = clock;        //time of request
 reserve(q[k]);        /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime1)
  {
   wait(idle1);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr1(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3); /*record system response time */
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    //Once go has been received continue
execution
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime1)
   {
    CW = ExpBackoff(CW, 2, aCWmax1);
    wait(idle1);
    w = rand()%CW;
    BackOffCtr1(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime1 && st!=1))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs. */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin2;
    t1 = clock;     //time of request
 reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime2)
  {
   wait(idle2); //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr2(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime2)
   {
    CW = ExpBackoff(CW, 2, aCWmax2);
    wait(idle2);
    w = rand()%CW;
    BackOffCtr2(w);
    retx++;
    xmitting++;    /*Increment the number of stations
transmitting */
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);   /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime2 && st!=1))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
    }
   }
   else
    record(1.0, drptbl); //Record that frame was dropped
  release(q[k]);
 }

void cust(int i)     //process customer
{
 char procname[32];
 MBOX mb;
 sprintf(procname, "cust %d", i);
 create(procname);    //required create statement
 mb = mailbox("mb");    /*Mailbox to receive messages on
dropped frames */
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);   //note arrival
  if (qlength(queues[i]) < 5)
  {
   if (random(0,999)%501 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
  else
  {
   if (random(0,99)%51 != 0)
   {
    double svc = frameSize/LinkSpeed;
    ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
    cnt--;
   }
   else
    DropFrame(queues, drptbl, mb, i);
  }
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  if (qlength(queues[i]) < 3)
```

```
    {
     if (random(0,999)%502 != 0)
     {
      double svc = frameSize/LinkSpeed;
      ProcessFrame1(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
      cnt--;
     }
     else
      DropFrame(queues, drptbl, mb, i);
    }
    else
    {
     if (random(0,99)%52 != 0)
     {
      double svc = frameSize/LinkSpeed;
      ProcessFrame1(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
      cnt--;
     }
     else
      DropFrame(queues, drptbl, mb, i);
    }
    note_exit(qtbl1);
   }
   else
   {
    note_entry(qtbl2);
    if (qlength(queues[i]) < 10)
    {
     if (random(0,999)%503 != 0)
     {
      double svc = frameSize/LinkSpeed;
      ProcessFrame2(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
      cnt--;
     }
     else
      DropFrame(queues, drptbl, mb, i);
    }
    else
    {
     if (random(0,99)%53 != 0)
     {
      double svc = frameSize/LinkSpeed;
      ProcessFrame2(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
      cnt--;
     }
     else
      DropFrame(queues, drptbl, mb, i);
```

```
  }
        note_exit(qtbl2);   //note departure
 }
 delete_mailbox(mb);
    if(cnt == 0)
        set(done);     //if last arrival, signal
}

void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
  cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing()                //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime);  //Wait for an AIFS interval
   st = state(busy);
   if (st==2)
    set(idle);   /*If the medium is free for an AIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing1()               //Check state of medium
{
  create("MediumSensing1");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime1);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle1);   //If the medium is free for a AIFS interval
make it idle
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing2()    //Check state of medium
{
  create("MediumSensing2");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime2);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle2);   /*If the medium is free for a AIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void ACKgenerator()     /*Determines whether or not an ACK can
be sent */
{
 create("ACKgenerator");
 set_priority(2);
 int stdone;
 double ACKsize = ACKLgth+ PHYHdr;
 double svc = ACKsize/LinkSpeed;
 record(ACKsize,frmtbl);
 do
 {
  wait(finxmission);
  if (xmitting==1 || xmitting==0)   /*Generate an ACK iff one
station is transmitting */
  {
   hold(aSIFSTime);
   set(busy);
   use(s, svc);   //Transmit the ACK frame
   clear(busy);
   set(ACK);
   set(go);
  }
  else
  {
   clear(ACK);
   hold(aSIFSTime);
   set(go);
  }
  stdone = state(done);
 }
 while (stdone == 2);
}
```

APPENDIX E

EDCF WITH DIFFERENT FRAME

RETRY LIMITS CODE LISTING

```
/* Code to simulate EDCF function */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin       15      //Number of slots
#define aCWmax       63      //Number of slots
#define aCWmin1      31      //Number of slots
#define aCWmax1     127      //Number of slots
#define aCWmin2      63      //Number of slots
#define aCWmax2    1023      //Number of slots
#define aSlotTime   0.00002  //time in seconds. 20 microsecs
#define aSIFSTime   0.00001  //time in seconds. 10 microsecs
#define aPIFSTime   0.00004  //time in seconds. 40 microsecs
#define aDIFSTime   0.00005  /*time in seconds. DIFS = SIFS +
(2*aSlotTime) */
#define aAIFSTime   0.00005  //time in seconds. 50 microsecs
#define aAIFSTime1  0.00007  //time in seconds. 70 microsecs
#define aAIFSTime2  0.00009  //time in seconds. 90 microsecs
#define aMPDUMaxLength 4095  /*MPDU in bits.  According to the
standard 1 <= x <= 4095 */
#define dot11RetryLimit 7    /*Number of times a frame may be
retransmitted */
#define dot11MSDULifeTime 0.06  /*Amount of time that a source
0 frame can be alive */
#define dot11MSDULifeTime1 0.1  /*Amount of time that a source
1 frame can be alive */
#define dot11MSDULifeTime2 0.2  /*Amount of time that a source
2 frame can be alive */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0   // 54 Mbps
#define NARS  100000            //Number of arrivals
#define STNS    20

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;   //Event that the medium is idle after aAIFSTime
EVENT idle1;  //Event that the medium is idle after aAIFSTime1
EVENT idle2;  //Event that the medium is idle after aAIFSTime2
EVENT busy;   //Event that the medium is currently busy
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;     /Event that a function can continue execution

int cnt;
int xmitting;  //Nbr of stations transmitting simultaneously
int j;
double IATM;   //Frame inter-arrival times

void init();   //Initializes structures used in the simulation
float GenerateFrame();  //Generates frames
int ExpBackoff(int CW, int PF, int amaxCW); /*Exponential
Back-off algorithm */
void BackOffCtr(int w);  /*Counts down the number of slots in
the backoff interval */
void BackOffCtr1(int w); /*Counts down the number of slots in
the backoff interval */
void BackOffCtr2(int w); /*Counts down the number of slots in
the backoff interval */
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i, int limtry);
 //Processes frames
void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i, int limtry);
 //Processes frames
void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i, int limtry);
 //Processes frames
void cust(int i);  //Generates and processes the frame
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void MediumSensing1(); //Medium Sensing function
void MediumSensing2(); //Medium Sensing function
void ACKgenerator(); /*Generates ACK iff only one station is
transmitting. */
```

```
void sim()                //1st process - named sim
{
 int stdone;
 FILE *out;
 char filename[16];

    sprintf(filename, "outrledcf%d.txt", STNS);
    set_model_name("Simulation of EDCF");
 out = fopen(filename, "wt");
 for (IATM = 0.1000; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");             //required create statement

  max_processes(100000000);
  max_events(10000000);
  init();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
        clear(ACK);
  clear(idle);
  clear(idle1);
  clear(idle2);
  clear(busy);
  clear(finxmission);
  clear(go);
  xmitting = 0;
  reset();

  cnt = 3*NARS;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                   //wait until all done
    fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
                %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
              ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
              ,table_var(tbl),table_mean(rtxtbl)
              ,table_mean(cwtbl),table_mean(drptbl)
              ,table_mean(tbl1),table_var(tbl1)
              ,table_mean(rtxtbl1),table_mean(cwtbl1)
              ,table_mean(drptbl1),table_mean(tbl2)
              ,table_var(tbl2),table_mean(rtxtbl2)
              ,table_mean(cwtbl2),table_mean(drptbl2)
              ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
   report();                              //print report
   mdlstat();
}

void init()
{
 s = facility("medium");               //initialize facility
 facility_set(queues,"src_queues",STNS);

 done = event("done");                 //initialize event
 ACK = event("ACK");
 idle = event("idle");
 idle1 = event("idle1");
 idle2 = event("idle2");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");

 qtbl = qhistogram("num from source 0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source 1", 10l);
 qtbl2 = qhistogram("num from source 2", 10l);
 xmitting = 0;
}

float GenerateFrame()    //Generates frames
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}

int ExpBackoff(int CW, int PF, int amaxCW) /*Exponential Back-
off algorithm */
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= amaxCW)
  return(tempCW);
 else
  return (CW);
}

void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}

void BackOffCtr1(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle1);
 }
}

void BackOffCtr2(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle2);
 }
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k, int limtry)
 //Processes frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin;
    t1 = clock;        //time of request
 reserve(q[k]);        //Enter this frame in the queue for the
source being processed
  if ((clock - t1) < dot11MSDULifeTime)
  {
   wait(idle);   //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;    /*Increment the number of stations
transmitting */
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;    /*Decrement the number of stations
transmitting */
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<limtry && (clock - t1) <
dot11MSDULifeTime)
   {
    CW = ExpBackoff(CW, 2, aCWmax);
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;    /*Increment the number of stations
transmitting */
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); /*record response time for all
sources */
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == limtry || ((clock - t1) >= dot11MSDULifeTime &&
st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k, int limtry)
 //Processes frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs.*/
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin1;
    t1 = clock;     //time of request
 reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
```

```
  if ((clock - t1) < dot11MSDULifeTime1)
  {
   wait(idle1);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr1(w);
   xmitting++;    /*Increment the number of stations
transmitting */
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  //record response time for this
source
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<limtry && (clock - t1) <
dot11MSDULifeTime1)
   {
    CW = ExpBackoff(CW, 2, aCWmax1);
    wait(idle1);
    w = rand()%CW;
    BackOffCtr1(w);
    retx++;
    xmitting++;    /*Increment the number of stations
transmitting */
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == limtry || ((clock - t1) >= dot11MSDULifeTime1
&& st!=1))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k, int limtry)
 //Processes frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs. */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

    CW = aCWmin2;
    t1 = clock;        //time of request
 reserve(q[k]);        /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime2)
  {
   wait(idle2);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr2(w);
   xmitting++;    /*Increment the number of stations
transmitting */
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     /*record response time for all
sources*/
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<limtry && (clock - t1) <
dot11MSDULifeTime2)
   {
    CW = ExpBackoff(CW, 2, aCWmax2);
    wait(idle2);
    w = rand()%CW;
    BackOffCtr2(w);
    retx++;
    xmitting++;    /*Increment the number of stations
transmitting */
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;    /*Decrement the number of stations
transmitting */
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == limtry || ((clock - t1) >= dot11MSDULifeTime2
&& st!=1))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void cust(int i)     //process customer
{
 char procname[32];
 sprintf(procname, "cust %d", i);
 create(procname);    //required create statement
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);   //note arrival
  ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i, 3);
 //Processes frames
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  ProcessFrame1(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i, 5);    //Processes frames
  note_exit(qtbl1);
 }
 else
 {
  note_entry(qtbl2);
  ProcessFrame2(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i, 7); //Processes frames
        note_exit(qtbl2);   //note departure
 }
    cnt--;
    if(cnt == 0)
        set(done);     //if last arrival, signal
}

void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
  cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing()    //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime);  //Wait for an AIFS interval
   st = state(busy);
   if (st==2)
    set(idle);
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing1()    //Check state of medium
{
  create("MediumSensing1");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime1);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle1);
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing2()    //Check state of medium
{
  create("MediumSensing2");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime2);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle2);
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void ACKgenerator()     /*Determines whether or not an ACK can
be sent */
{
 create("ACKgenerator");
 set_priority(2);
 int stdone;
 double ACKsize = ACKLgth+ PHYHdr;
 double svc = ACKsize/LinkSpeed;
 record(ACKsize,frmtbl);
 do
 {
  wait(finxmission);
  if (xmitting==1 || xmitting==0)  /*Generate an ACK iff one
station is transmitting */
  {
   hold(aSIFSTime);
   set(busy);
   use(s, svc);   //Transmit the ACK frame
   clear(busy);
   set(ACK);
   set(go);
  }
  else
  {
   clear(ACK);
   hold(aSIFSTime);
   set(go);
  }
  stdone = state(done);
 }
 while (stdone == 2);
}
```

APPENDIX F

ADAPTIVE EDCF CODE LISTING

```
/* Code to simulate Adaptive EDCF (AEDCF) function */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin        15      //Number of slots
#define aCWmax        63      //Number of slots
#define aCWmin1       31      //Number of slots
#define aCWmax1      127      //Number of slots
#define aCWmin2       63      //Number of slots
#define aCWmax2     1023      //Number of slots
#define aSlotTime   0.00002  //time in seconds. 20 microsecs
#define aSIFSTime   0.00001  //time in seconds. 10 microsecs
#define aPIFSTime   0.00004  //time in seconds. 40 microsecs
#define aDIFSTime   0.00005  /*time in seconds. DIFS = SIFS +
(2*aSlotTime) */
#define aAIFSTime   0.00005  //time in seconds. 50 microsecs
#define aAIFSTime1  0.00007  //time in seconds. 70 microsecs
#define aAIFSTime2  0.00009  //time in seconds. 90 microsecs
#define aMPDUMaxLength 4095  /*MPDU in bits.  According to the
standard 1 <= x <= 4095 */
#define dot11RetryLimit 7    /*Number of times a frame may be
retransmitted */
#define dot11MSDULifeTime 0.06  /*Amount of time that a source
0 frame can be alive */
#define dot11MSDULifeTime1 0.1  /*Amount of time that a source
1 frame can be alive */
#define dot11MSDULifeTime2 0.2  /*Amount of time that a source
2 frame can be alive */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0   // 54 Mbps
#define NARS  100000            //Number of arrivals
#define STNS    10
#define ObsvSlots 5000    /*Number of slots to observe b/4
computing collision probability */

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;   //Event that the medium is idle after aAIFSTime
EVENT idle1;  //Event that the medium is idle after aAIFSTime1
EVENT idle2;  //Event that the medium is idle after aAIFSTime2
EVENT busy;   //Event that the medium is busy after DIFS
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;     //Event that a function can continue execution

int cnt;
int xmitting;  //Nbr of stations transmitting simultaneously
int j;
int collisions[STNS]; /*holds number of collisions that occur
in a cycle */
double PktsSent[STNS]; /*holds number of frames sent in a
cycle */
double CollRate[STNS]; /*Holds instantaneous collision rate
per station */
double AvgCollRate[STNS]; /*Holds average collision rate per
station */
int lstCW[STNS];  /*Holds the last contention window for each
class */
int currCW[STNS]; /*Holds the current contention window for
each class */
double IATM;        //Frame inter-arrival times

void init();   //Initializes structures used in the simulation
void ResetCollArrays();  /*Resets the arrays used in computing
collision rates */
void ResetArrays();  //Resets the arrays that keep track of
history
float GenerateFrame();  //Generates frames
int ExpBackoff(int CW, int PF, int amaxCW); /*Exponential
Back-off algorithm */
void ResetCW(int k, int aminCW); /*Resets the contention
window using history */
```

```
void BackOffCtr(int w);  /*Counts down the number of slots in
the backoff interval */
void BackOffCtr1(int w); /*Counts down the number of slots in
the backoff interval */
void BackOffCtr2(int w); /*Counts down the number of slots in
the backoff interval */
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void cust(int i);  //Generates and processes the frame
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void MediumSensing1(); //Medium Sensing function
void MediumSensing2(); //Medium Sensing function
void ACKgenerator(); /*Generates ACK iff only one station is
transmitting. */
void CollRateEst(int i); //Estimates the collision rate per
station

void sim()        //1st process - named sim
{
 int stdone;
 FILE *out;
 char filename[16];

 sprintf(filename, "outaedcf%d.txt", STNS);
 set_model_name("Simulation of AEDCF");
 out = fopen(filename, "wt");
 for (IATM = 0.1000; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");              //required create statement

  max_processes(100000000);
  max_events(10000000);
  init();
  ResetCollArrays();
  ResetArrays();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
```

```
        clear(ACK);
  clear(idle);
  clear(idle1);
  clear(idle2);
  clear(busy);
  clear(finxmission);
  clear(go);
  ResetCollArrays();
  xmitting = 0;
  reset();
  ResetArrays();
  cnt = 3*NARS;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                     //wait until all done
  fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
                %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
              ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
              ,table_var(tbl),table_mean(rtxtbl)
              ,table_mean(cwtbl),table_mean(drptbl)
              ,table_mean(tbl1),table_var(tbl1)
              ,table_mean(rtxtbl1),table_mean(cwtbl1)
              ,table_mean(drptbl1),table_mean(tbl2)
              ,table_var(tbl2),table_mean(rtxtbl2)
              ,table_mean(cwtbl2),table_mean(drptbl2)
              ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
    report();                                 //print report
    mdlstat();
}

void ResetCollArrays()
{
 int i;
 for (i=0; i<STNS; i++)
 {
  collisions[i]=0;
  PktsSent[i]=0;
  CollRate[i]=0;
 }
}

void ResetArrays()
{
 int i;
 for (i=0; i<STNS; i++)
 {
  AvgCollRate[i]=0;
  lstCW[i]=0;
  currCW[i]=0;
 }
}

void init()
{
 s = facility("medium");               //initialize facility
 facility_set(queues,"src_queues",STNS);

 done = event("done");                 //initialize event
 ACK = event("ACK");
 idle = event("idle");
 idle1 = event("idle1");
 idle2 = event("idle2");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");

 qtbl = qhistogram("num from source 0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source 1", 10l);
 qtbl2 = qhistogram("num from source 2", 10l);
 xmitting = 0;
}

float GenerateFrame()    //Generates frames
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}
```

```
int ExpBackoff(int CW, int PF, int amaxCW) /*Exponential Back-
off algorithm */
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= amaxCW)
  return(tempCW);
 else
  return (CW);
}

void ResetCW(int k, int aminCW)  /*Resets the contention
window using history */
{
 double MF;  //Multiplicative factor
 if (k%4 == 0) //Indicates it is of class 0
 {
  MF = AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 else if (k%2 == 1) //Indicates it is of class 1
 {
  MF = 3*AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 else    //indicates it is of class 2
 {
  MF = 5*AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 int tempCW = lstCW[k]*MF;
 if (tempCW < aminCW)
  currCW[k] = aminCW;
 else
  currCW[k] = tempCW;
}
void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}
```

```
void BackOffCtr1(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle1);
 }
}

void BackOffCtr2(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle2);
 }
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     /*Processes
frames */
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of contention
window size*/

 ResetCW(k, aCWmin);
 CW = currCW[k];
 t1 = clock;                         //time of request
 reserve(q[k]);      /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime)
  {
   wait(idle);   //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;    /*Increment the number of stations
transmitting */
```

```
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    //Wait for the ACKgenerator function to check
xmitting
   xmitting--;    //Decrement the number of stations
transmitting
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime)
   {
    CW = ExpBackoff(CW, 2, aCWmax);
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);  /*record response time for this
source */
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    //Wait for the ACKgenerator function to check
xmitting
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;    /*keeps track of how many times a
retransmission occurs. */
    int    w;         //used to keep track of number of slots
    int    CW;        /*used to keep track of the size of the
contention window */

    ResetCW(k, aCWmin1);
    CW = currCW[k];
    t1 = clock;       //time of request
    reserve(q[k]);    /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime1)
  {
   wait(idle1);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr1(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime1)
   {
    CW = ExpBackoff(CW, 2, aCWmax1);
    wait(idle1);
    w = rand()%CW;
    BackOffCtr1(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime1 && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs. */
```

```
    int    w;          //used to keep track of number of slots
    int    CW;  //used to keep track of contention window size

 ResetCW(k, aCWmin2);
 CW = currCW[k];
 t1 = clock;                    //time of request
 reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime2)
  {
   wait(idle2);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr2(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);
   record(clock-t1, tbl3);
   set(finxmission);  //Indicate that the transmission is done
   wait(go);
   xmitting--;
   clear(go);
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime2)
   {
    CW = ExpBackoff(CW, 2, aCWmax2);
    wait(idle2);
    w = rand()%CW;
    BackOffCtr2(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3);
    set(finxmission);
    wait(go);
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime2 && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void cust(int i)     //process customer
{
 char procname[32];
 sprintf(procname, "cust %d", i);
 create(procname);    //required create statement
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);   //note arrival
  ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  ProcessFrame1(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
  note_exit(qtbl1);
 }
 else
 {
  note_entry(qtbl2);
  ProcessFrame2(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
        note_exit(qtbl2);   //note departure
 }
    cnt--;
    if(cnt == 0)
        set(done);     //if last arrival, signal
}
```

```
void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 CollRateEst(i);    /*Start the collision rate estimator for
this source */
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
        cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing()    //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime);  //Wait for an AIFS interval
   st = state(busy);
   if (st==2)
    set(idle);   /*If the medium is free for an AIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing1()    //Check state of medium
{
  create("MediumSensing1");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime1);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle1);
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing2()    //Check state of medium
{
  create("MediumSensing2");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime2);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle2);
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void ACKgenerator()     /*Determines whether or not an ACK can
be sent */
{
 create("ACKgenerator");
 set_priority(2);
 int stdone;
 double ACKsize = ACKLgth+ PHYHdr;
 double svc = ACKsize/LinkSpeed;
 record(ACKsize,frmtbl);
 do
 {
  wait(finxmission);
  if (xmitting==1 || xmitting==0)  /*Generate an ACK iff one
station is transmitting */
  {
   hold(aSIFSTime);
   set(busy);
   use(s, svc);   //Transmit the ACK frame
   clear(busy);
   set(ACK);
   set(go);
  }
  else
  {
   clear(ACK);
   hold(aSIFSTime);
   set(go);
  }
  stdone = state(done);
 }
 while (stdone == 2);
}

void CollRateEst(int k)  /*Estimates the collision rate per
station */
{
 char procname[32];
 sprintf(procname, "CollRateEst %d", k);
 create(procname);
 int stdone;
 int i;
 do
 {
  for (i=0; i<ObsvSlots; i++)
   hold(aSlotTime);
  if (PktsSent[k]!= 0)
   CollRate[k] = (collisions[k]*1.0)/PktsSent[k];
  else
   CollRate[k] = 0;
  /*The equation below is based on page 3 of the AEDCF paper.
alpha = 0.8 */
  AvgCollRate[k] = (0.2* CollRate[k]) + (0.8* AvgCollRate[k]);
  ResetCollArrays();
  stdone = state(done);
 }
 while (stdone == 2);
}
```

APPENDIX G

ADAPTIVE EDCF WITH VARYING PERSISTENCE FACTOR CODE LISTING

```
/* Code to simulate Adaptive EDCF (AEDCF) function with
varying PF */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin       15      //Number of slots
#define aCWmax       63      //Number of slots
#define aCWmin1      31      //Number of slots
#define aCWmax1     127      //Number of slots
#define aCWmin2      63      //Number of slots
#define aCWmax2    1023      //Number of slots
#define aSlotTime   0.00002  //time in seconds. 20 microsecs
#define aSIFSTime   0.00001  //time in seconds. 10 microsecs
#define aPIFSTime   0.00004  //time in seconds. 40 microsecs
#define aDIFSTime   0.00005  /*time in seconds. DIFS = SIFS +
(2*aSlotTime) */
#define aAIFSTime   0.00005  //time in seconds. 50 microsecs
#define aAIFSTime1  0.00007  //time in seconds. 70 microsecs
#define aAIFSTime2  0.00009  //time in seconds. 90 microsecs
#define aMPDUMaxLength 4095  /*MPDU in bits.  According to the
standard 1 <= x <= 4095 */
#define dot11RetryLimit 7    /*Number of times a frame may be
retransmitted */
#define dot11MSDULifeTime 0.06  /*Amount of time that a source
0 frame can be alive */
#define dot11MSDULifeTime1 0.1  /*Amount of time that a source
1 frame can be alive */
#define dot11MSDULifeTime2 0.2  /*Amount of time that a source
2 frame can be alive */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0   // 54 Mbps
#define NARS  100000             //Number of arrivals
#define STNS    30
#define ObsvSlots 5000    /*Number of slots to observe b/4
computing collision probability */

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes
TABLE prbtbl;  /*Table with collision probabilities for the
simulation */

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;   //Event that the medium is idle after aAIFSTime
EVENT idle1;  //Event that the medium is idle after aAIFSTime1
EVENT idle2;  //Event that the medium is idle after aAIFSTime2
EVENT busy;   //Event that the medium is busy after DIFS
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;     //Event that a function can continue execution

int cnt;
int xmitting;  //Nbr of stations transmitting simultaneously
int j;
int collisions[STNS]; //holds number of collisions that occur
in a cycle
double PktsSent[STNS]; //holds amount of data sent in a cycle
double CollRate[STNS]; //Holds instantaneous collision rate
per station
double AvgCollRate[STNS]; //Holds average collision rate per
station
int lstCW[STNS];  //Holds the last contention window for each
class
int currCW[STNS]; //Holds the current contention window for
each class
int PFactor[STNS]; //Holds the current persistence factor for
each class
double IATM;        //Frame inter-arrival times

void init();   //Initializes structures used in the simulation
void ResetCollArrays();  //Resets the arrays used in computing
collision rates
void ResetArrays();  //Resets the arrays that keep track of
history
float GenerateFrame();  //Generates frames
int ExpBackoff(int CW, int PF, int amaxCW); //Exponential
Back-off algorithm
```

```
void ResetCW(int k, int aminCW); //Resets the contention
window using history
void BackOffCtr(int w);  //Counts down the number of slots in
the backoff interval
void BackOffCtr1(int w); //Counts down the number of slots in
the backoff interval
void BackOffCtr2(int w); //Counts down the number of slots in
the backoff interval
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void cust(int i);  //Generates and processes the frame
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void MediumSensing1(); //Medium Sensing function
void MediumSensing2(); //Medium Sensing function
void ACKgenerator(); //Generates ACK iff only one station is
transmitting.
void CollRateEst(int i); //Estimates the collision rate per
station

void sim()                             //1st process - named
sim
{
 int stdone;
 FILE *out;
 char filename[16];

    sprintf(filename, "outpaedcf%d.txt", STNS);
    set_model_name("Simulation of AEDCF, w/ varying PF");
 out = fopen(filename, "wt");
 for (IATM = 0.1000; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");                       //required create
statement

  max_processes(100000000);
  max_events(10000000);
  init();
  ResetCollArrays();
  ResetArrays();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
        clear(ACK);
  clear(idle);
  clear(idle1);
  clear(idle2);
  clear(busy);
  clear(finxmission);
  clear(go);
  ResetCollArrays();
  xmitting = 0;
  reset();
  ResetArrays();
  cnt = 3*NARS;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                   //wait until all done
    fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
                %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
              ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
              ,table_var(tbl),table_mean(rtxtbl)
              ,table_mean(cwtbl),table_mean(drptbl)
              ,table_mean(tbl1),table_var(tbl1)
              ,table_mean(rtxtbl1),table_mean(cwtbl1)
              ,table_mean(drptbl1),table_mean(tbl2)
              ,table_var(tbl2),table_mean(rtxtbl2)
              ,table_mean(cwtbl2),table_mean(drptbl2)
              ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
    report();                               //print report
    mdlstat();
}

void ResetCollArrays()
{
 int i;
 for (i=0; i<STNS; i++)
 {
  collisions[i]=0;
  PktsSent[i]=0;
  CollRate[i]=0;
 }
```

```
}

void ResetArrays()
{
 int i;
 for (i=0; i<STNS; i++)
 {
  AvgCollRate[i]=0;
  lstCW[i]=0;
  currCW[i]=0;
  PFactor[i]=2;
 }
}

void init()
{
 s = facility("medium");               //initialize facility
 facility_set(queues,"src_queues",STNS);

 done = event("done");                 //initialize event
 ACK = event("ACK");
 idle = event("idle");
 idle1 = event("idle1");
 idle2 = event("idle2");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");
 prbtbl = table("Collision probabilities");

 qtbl = qhistogram("num from source 0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source 1", 10l);
 qtbl2 = qhistogram("num from source 2", 10l);
 xmitting = 0;
}
```

```
float GenerateFrame()    //Generates frames
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}

int ExpBackoff(int CW, int PF, int amaxCW) /*Exponential Back-
off algorithm */
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= amaxCW)
  return(tempCW);
 else
  return (CW);
}

void ResetCW(int k, int aminCW)  /*Resets the contention
window using history */
{
 double MF;  //Multiplicative factor
 if (k%4 == 0) //Indicates it is of class 0
 {
  MF = AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 else if (k%2 == 1) //Indicates it is of class 1
 {
  MF = 3*AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 else    //indicates it is of class 2
 {
  MF = 5*AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 int tempCW = lstCW[k]*MF;
 if (tempCW < aminCW)
  currCW[k] = aminCW;
 else
  currCW[k] = tempCW;
 if (AvgCollRate[k]> 0.0625)
  PFactor[k] = 4;
 else
  PFactor[k] = 2;
}
void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}

void BackOffCtr1(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle1);
 }
}

void BackOffCtr2(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle2);
 }
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     /*Processes
frames */
{
    double t1;
    int    st;
    int    retx=0;    /*keeps track of how many times a
retransmission occurs */
    int    w;         //used to keep track of number of slots
    int    CW;        //used to keep track of contention
window size */

 ResetCW(k, aCWmin);
 CW = currCW[k];
 t1 = clock;                         //time of request
 reserve(q[k]);      /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime)
  {
   wait(idle);   //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;    /*Increment the number of stations
transmitting */
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     /*record system response time*/
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;    /*Decrement the number of stations
transmitting */
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime)
   {
    CW = ExpBackoff(CW, PFactor[k], aCWmax);
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3); /*record system response time*/
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     /*Processes
frames */
{
    double t1;
    int    st;
    int    retx=0;     /*keeps track of how many times a
retransmission occurs. */
    int    w;          //used to keep track of number of slots
    int    CW;         /*used to keep track of the size of the
contention window */

 ResetCW(k, aCWmin1);
 CW = currCW[k];
 t1 = clock;                     //time of request
 reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime1)
  {
   wait(idle1);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr1(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);
   record(clock-t1, tbl3);
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
```

```
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime1)
   {
    CW = ExpBackoff(CW, PFactor[k], aCWmax1);
    wait(idle1);
    w = rand()%CW;
    BackOffCtr1(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3);
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    //Once go has been received continue
execution
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime1 && st!=1))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}
```

```
void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     /*Processes
frames */
{
    double t1;
    int    st;
    int    retx=0;   /*keeps track of how many times a
retransmission occurs.*/
    int    w;        //used to keep track of number of slots
    int    CW;       /*used to keep track of the size of the
contention window */

 ResetCW(k, aCWmin2);
 CW = currCW[k];
 t1 = clock;                      //time of request
 reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime2)
  {
   wait(idle2);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr2(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     /*record response time for all
sources */
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    //Once go has been received continue
execution
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime2)
   {
    CW = ExpBackoff(CW, PFactor[k], aCWmax2);
    wait(idle2);
    w = rand()%CW;
    BackOffCtr2(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3);
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime2 && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void cust(int i)     //process customer
{
 char procname[32];
 sprintf(procname, "cust %d", i);
 create(procname);    //required create statement
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);   //note arrival
  ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  ProcessFrame1(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
  note_exit(qtbl1);
 }
 else
 {
  note_entry(qtbl2);
  ProcessFrame2(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
        note_exit(qtbl2);   //note departure
 }
    cnt--;
    if(cnt == 0)
        set(done);     //if last arrival, signal
}

void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 CollRateEst(i);    /*Start the collision rate estimator for
this source */
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
        cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing()    //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime);  //Wait for an AIFS interval
   st = state(busy);
   if (st==2)
    set(idle);   /*If the medium is free for an AIFS interval
make it idle */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing1()    //Check state of medium
{
  create("MediumSensing1");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime1);  //Wait for a AIFS interval
   st = state(busy);
   if (st==2)
    set(idle1);
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing2()    //Check state of medium
{
  create("MediumSensing2");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime2);
   st = state(busy);
   if (st==2)
    set(idle2);
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void ACKgenerator()     /*Determines whether or not an ACK can
be sent */
{
 create("ACKgenerator");
 set_priority(2);
 int stdone;
 double ACKsize = ACKLgth+ PHYHdr;
 double svc = ACKsize/LinkSpeed;
 record(ACKsize,frmtbl);
 do
 {
  wait(finxmission);
  if (xmitting==1 || xmitting==0)  /*Generate an ACK iff one
station is transmitting */
  {
   hold(aSIFSTime);
   set(busy);
   use(s, svc);   //Transmit the ACK frame
   clear(busy);
   set(ACK);
   set(go);
  }
  else
  {
   clear(ACK);
   hold(aSIFSTime);
   set(go);
  }
  stdone = state(done);
 }
 while (stdone == 2);
}

void CollRateEst(int k)  /*Estimates the collision rate per
station */
{
 char procname[32];
 sprintf(procname, "CollRateEst %d", k);
 create(procname);
 int stdone;
 int i;
 do
 {
  for (i=0; i<ObsvSlots; i++)
   hold(aSlotTime);
  if (PktsSent[k]!= 0)
```

```
   CollRate[k] = (collisions[k]*1.0)/PktsSent[k];
  else
   CollRate[k] = 0;
  /*The equation below is based on page 3 of the AEDCF paper.
alpha = 0.8 */
  AvgCollRate[k] = (0.2* CollRate[k]) + (0.8* AvgCollRate[k]);
  record(AvgCollRate[k],prbtbl);
  ResetCollArrays();
  stdone = state(done);
 }
 while (stdone == 2);
}
```

APPENDIX H

ADAPTIVE EDCF WITH VARYING PERSISTENCE FACTOR CODE AND CW_{MAX} CODE LISTING

```
/* Code to simulate Adaptive EDCF (AEDCF) function with
varying PF and CWmax */
/* All times below are in seconds */

#include "csim.h"
#include <stdio.h>

#define aCWmin       15      //Number of slots
#define aCWmax       63      //Number of slots
#define aCWmin1      31      //Number of slots
#define aCWmax1     127      //Number of slots
#define aCWmin2      63      //Number of slots
#define aCWmax2    1023      //Number of slots
#define aSlotTime   0.00002  //time in seconds. 20 microsecs
#define aSIFSTime   0.00001  //time in seconds. 10 microsecs
#define aPIFSTime   0.00004  //time in seconds. 40 microsecs
#define aDIFSTime   0.00005  /*time in seconds. DIFS = SIFS +
(2*aSlotTime) */
#define aAIFSTime   0.00005  //time in seconds. 50 microsecs
#define aAIFSTime1  0.00007  //time in seconds. 70 microsecs
#define aAIFSTime2  0.00009  //time in seconds. 90 microsecs
#define aMPDUMaxLength 4095  /*MPDU in bits.  According to the
standard 1 <= x <= 4095 */
#define dot11RetryLimit 7    /*Number of times a frame may be
retransmitted */
#define dot11MSDULifeTime 0.06  /*Amount of time that a source
0 frame can be alive */
#define dot11MSDULifeTime1 0.1  /*Amount of time that a source
1 frame can be alive */
#define dot11MSDULifeTime2 0.2  /*Amount of time that a source
2 frame can be alive */
#define MACHdr        272.0      //MAC Header Length
#define PHYHdr        192.0      //PHY Header Length
#define ACKLgth       112.0      //Length of ACK frame

#define LinkSpeed    54000000.0   // 54 Mbps
#define NARS  100000            //Number of arrivals
#define STNS    30
#define ObsvSlots 5000    /*Number of slots to observe b/4
computing collision probability */

FACILITY s;   //Medium
FACILITY queues[STNS];

TABLE tbl;   //To hold frame response times for source 0
TABLE tbl1;   //To hold frame response times for source 1
TABLE tbl2;   //To hold frame response times for source 2
TABLE tbl3;   //To hold frame response times for all sources
TABLE rtxtbl;  //Table showing number of retransmissions
TABLE cwtbl;  //Table with contention window sizes
TABLE drptbl;  //Table with number of dropped frames
TABLE rtxtbl1;  //Table showing number of retransmissions
TABLE cwtbl1;  //Table with contention window sizes
TABLE drptbl1;  //Table with number of dropped frames
TABLE rtxtbl2;  //Table showing number of retransmissions
TABLE cwtbl2;  //Table with contention window sizes
TABLE drptbl2;  //Table with number of dropped frames
TABLE frmtbl;  //Table with frame sizes
TABLE prbtbl;  /*Table with collision probabilities for the
simulation */

QTABLE qtbl;
QTABLE qtbl1;
QTABLE qtbl2;

EVENT done;   //Event that the simulation is complete
EVENT ACK;    //Event that an ACK has been received.
EVENT idle;   //Event that the medium is idle after aAIFSTime
EVENT idle1;  //Event that the medium is idle after aAIFSTime1
EVENT idle2;  //Event that the medium is idle after aAIFSTime2
EVENT busy;   //Event that the medium is currently busy
EVENT finxmission;  /*Event that a station has finished a
transmission */
EVENT go;      //Event that a function can continue execution

int cnt;
int xmitting;  //Nbr of stations transmitting simultaneously
int j;
int collisions[STNS]; //holds number of collisions that occur
in a cycle */
double PktsSent[STNS]; //Holds amount of data sent in a cycle
double CollRate[STNS]; /*Holds instantaneous collision rate
per station */
double AvgCollRate[STNS]; //Holds average collision rate per
station
int lstCW[STNS];  /*Holds last contention window size for each
class */
int currCW[STNS]; /*Holds the current contention window size
for each class */
int PFactor[STNS]; /*Holds the current persistence factor for
each class */
double IATM;        //Frame inter-arrival times

void init();   //Initializes structures used in the simulation
void ResetCollArrays();  /*Resets the arrays used in computing
collision rates */
void ResetArrays();  /*Resets the arrays that keep track of
history */
float GenerateFrame();  //Generates frames
```

```
int ExpBackoff(int CW, int PF, int amaxCW); /*Exponential
Back-off algorithm */
void ResetCW(int k, int aminCW); /*Resets the contention
window using history */
void BackOffCtr(int w);  /*Counts down the number of slots in
the backoff interval */
void BackOffCtr1(int w); /*Counts down the number of slots in
the backoff interval */
void BackOffCtr2(int w); /*Counts down the number of slots in
the backoff interval */
void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int i); //Processes frames
void cust(int i);  //Generates and processes the frame
void source(int i);  //Simulates a frame source
void MediumSensing(); //Medium Sensing function
void MediumSensing1(); //Medium Sensing function
void MediumSensing2(); //Medium Sensing function
void ACKgenerator(); /*Generates ACK iff only one station is
transmitting. */
void CollRateEst(int i); /*Estimates the collision rate per
station */

void sim()                 //1st process - named sim
{
 int stdone;
 FILE *out;
 char filename[16];

 sprintf(filename, "outcpaedcf%d.txt", STNS);
 set_model_name("Simulation of AEDCF, w/ varying PF, CWmax");
 out = fopen(filename, "wt");
 for (IATM = 0.0500; IATM > 0.0095; IATM = IATM - 0.0020)
 {
  create("sim");             //required create statement

  max_processes(100000000);
  max_events(10000000);
  init();
  ResetCollArrays();
  ResetArrays();

  reset_prob(clock);
  cnt = NARS/8;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);
  clear(done);
        clear(ACK);
  clear(idle);
  clear(idle1);
  clear(idle2);
  clear(busy);
  clear(finxmission);
  clear(go);
  ResetCollArrays();
  xmitting = 0;
  reset();
  ResetArrays();
  cnt = 3*NARS;
  MediumSensing();
  MediumSensing1();
  MediumSensing2();
  ACKgenerator();
  for (j=0; j<STNS; j++)
   source(j);
  wait(done);                  //wait until all done
  fprintf(out ,"%i\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t
                %f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n"
              ,STNS,table_mean(frmtbl),IATM ,table_mean(tbl)
              ,table_var(tbl),table_mean(rtxtbl)
              ,table_mean(cwtbl),table_mean(drptbl)
              ,table_mean(tbl1),table_var(tbl1)
              ,table_mean(rtxtbl1),table_mean(cwtbl1)
              ,table_mean(drptbl1),table_mean(tbl2)
              ,table_var(tbl2),table_mean(rtxtbl2)
              ,table_mean(cwtbl2),table_mean(drptbl2)
              ,table_mean(tbl3),table_var(tbl3),util(s));
  fflush(out);
  rerun();
 }
 fclose(out);
   report();                             //print report
   mdlstat();
}

void ResetCollArrays()
{
 int i;
 for (i=0; i<STNS; i++)
 {
  collisions[i]=0;
  PktsSent[i]=0;
  CollRate[i]=0;
 }
```

```
}

void ResetArrays()
{
 int i;
 for (i=0; i<STNS; i++)
 {
  AvgCollRate[i]=0;
  lstCW[i]=0;
  currCW[i]=0;
  PFactor[i]=2;
 }
}

void init()
{
 s = facility("medium");               //initialize facility
 facility_set(queues,"src_queues",STNS);

 done = event("done");                 //initialize event
 ACK = event("ACK");
 idle = event("idle");
 idle1 = event("idle1");
 idle2 = event("idle2");
 busy = event("busy");
 finxmission = event("finxmission");
 go = event("go");

 tbl = table("Frame response tms");    //initialize table
 rtxtbl = table("Nbr of retransmissions");
 cwtbl = table("Size of contention window");
 drptbl= table("Nbr of dropped frames");
 tbl1 = table("Frame response tms - class 1");
 rtxtbl1 = table("Nbr of retransmissions - class 1");
 cwtbl1 = table("Size of contention window - class 1");
 drptbl1= table("Nbr of dropped frames - class 1");
 tbl2 = table("Frame response tms - class 2");
 rtxtbl2 = table("Nbr of retransmissions - class 2");
 cwtbl2 = table("Size of contention window - class 2");
 drptbl2= table("Nbr of dropped frames - class 2");
 tbl3 = table("System resp tms for all classes");
 frmtbl = table("Sizes of frames for all classes");
 prbtbl = table("Collision probabilities");

 qtbl = qhistogram("num from source 0", 10l);   /*initialize
qhistogram */
 qtbl1 = qhistogram("num from source 1", 10l);
 qtbl2 = qhistogram("num from source 2", 10l);
 xmitting = 0;
}
```

```
float GenerateFrame()    //Generates frames
{
 int tmpframe = random(1,4095); /*Returns a random # between 1
and 4095 */
 return((tmpframe*8.0) + MACHdr + PHYHdr);
}

int ExpBackoff(int CW, int PF, int amaxCW) //Exponential Back-
off algorithm
{
 int tempCW= (PF*(CW+1)) - 1;
 if (tempCW <= amaxCW)
  return(tempCW);
 else
  return (CW);
}

void ResetCW(int k, int aminCW)  /*Resets the contention
window using history */
{
 double MF;  //Multiplicative factor
 if (k%4 == 0) //Indicates it is of class 0
 {
  MF = AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 else if (k%2 == 1) //Indicates it is of class 1
 {
  MF = 3*AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 else    //indicates it is of class 2
 {
  MF = 5*AvgCollRate[k];
  if (MF < 0.8)
   MF = 0.8;
 }
 int tempCW = lstCW[k]*MF;
 if (tempCW < aminCW)
  currCW[k] = aminCW;
 else
  currCW[k] = tempCW;
 if (AvgCollRate[k]> 0.0625)
  PFactor[k] = 4;
 else
  PFactor[k] = 2;
}
void BackOffCtr(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle);
 }
}

void BackOffCtr1(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle1);
 }
}

void BackOffCtr2(int w)
{
 int st;
 int i;
 for (i=w; i>0; i--)
 {
  hold(aSlotTime);
  st = state(busy); //Sense medium
  if (st==1)   //If the medium is busy
   wait(idle2);
 }
}

void ProcessFrame(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;    /*keeps track of how many times a
retransmission occurs */
    int    w;         //used to keep track of number of slots
    int    CW;        /*used to keep track of contention
window size*/

 ResetCW(k, aCWmin);
 CW = currCW[k];
 t1 = clock;      //time of request
 reserve(q[k]);   /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime)
  {
   wait(idle);   //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr(w);
   xmitting++;    //Increment the nbr of stations transmitting
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;   //Decrement the nbr of stations transmitting
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime)
   {
    if (AvgCollRate[k]< 0.03125)
     CW = ExpBackoff(CW, PFactor[k], aCWmax);
    else
     CW = ExpBackoff(CW, PFactor[k], (2*(aCWmax+1) - 1));
    wait(idle);
    w = rand()%CW;
    BackOffCtr(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void ProcessFrame1(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     /*Processes
frames */
{
    double t1;
    int    st;
    int    retx=0;  //keeps track of nbr of  retransmissions.
    int    w;       //used to keep track of number of slots
    int    CW;  //used to keep track of contention window size

 ResetCW(k, aCWmin1);
 CW = currCW[k];
 t1 = clock;                     //time of request
 reserve(q[k]);     /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime1)
  {
   wait(idle1);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr1(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);  /*record response time for this
source */
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
```

```
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime1)
   {
    if (AvgCollRate[k]< 0.03125)
     CW = ExpBackoff(CW, PFactor[k], aCWmax1);
    else
     CW = ExpBackoff(CW, PFactor[k], (2*(aCWmax1+1) - 1));
    wait(idle1);
    w = rand()%CW;
    BackOffCtr1(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime1 && st!=1))
    record(1.0, drptbl); //Record that this frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
```

```
 release(q[k]);
}

void ProcessFrame2(FACILITY q[], double svc, TABLE tbl, TABLE
cwtbl, TABLE rtxtbl, TABLE drptbl, int k)     //Processes
frames
{
    double t1;
    int    st;
    int    retx=0;  //keeps track of nbr of retransmissions
    int    w;       //used to keep track of number of slots
    int    CW;      /*used to keep track of the size of the
contention window */

 ResetCW(k, aCWmin2);
 CW = currCW[k];
 t1 = clock;                     //time of request
 reserve(q[k]);      /*Enter this frame in the queue for the
source being processed */
  if ((clock - t1) < dot11MSDULifeTime2)
  {
   wait(idle2);  //Wait until the medium is idle for aAIFSTime
   w = rand()%CW;
   BackOffCtr2(w);
   xmitting++;
   set(busy);
   use(s,svc);    //reserve medium
   clear(busy);
   record(clock-t1, tbl);
   record(clock-t1, tbl3);     //record system response time
   set(finxmission);  //Indicate that the transmission is done
   wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
   xmitting--;
   clear(go);    /*Once go has been received continue
execution */
   clear(finxmission);
   PktsSent[k]++;   //Indicate one frame has been sent
   lstCW[k] = CW;
   st = state(ACK);
   //As long as no ACK has been received.
   while (st!=1 && retx<dot11RetryLimit && (clock - t1) <
dot11MSDULifeTime2)
   {
    if (AvgCollRate[k]< 0.03125)
     CW = ExpBackoff(CW, PFactor[k], aCWmax2);
    else
     CW = ExpBackoff(CW, PFactor[k], (2*(aCWmax2+1) - 1));
    wait(idle2);
    w = rand()%CW;
    BackOffCtr2(w);
    retx++;
    xmitting++;
    set(busy);
    use(s,svc);    //reserve medium
    clear(busy);
    record(clock-t1, tbl);
    record(clock-t1, tbl3); //record system response time
    set(finxmission);  /*Indicate that the transmission is
done */
    wait(go);    /*Wait for the ACKgenerator function to check
xmitting */
    xmitting--;
    clear(go);    /*Once go has been received continue
execution */
    clear(finxmission);
    collisions[k]++;
    PktsSent[k]++;   //Indicate one frame has been sent
    lstCW[k] = CW;
    st = state(ACK);
   }
   record(retx, rtxtbl); //record number of retransmissions
   record(w, cwtbl);  //Record size of contention window
   if (retx == dot11RetryLimit || ((clock - t1) >=
dot11MSDULifeTime2 && st!=1))
    record(1.0, drptbl); //Record that frame was dropped
   else
   {
    clear(ACK);
    record(0.0, drptbl); //Record that frame wasn't dropped
   }
  }
  else
   record(1.0, drptbl); //Record that frame was dropped
 release(q[k]);
}

void cust(int i)     //process customer
{
 char procname[32];
 sprintf(procname, "cust %d", i);
 create(procname);    //required create statement
 float frameSize = GenerateFrame();
 record(frameSize, frmtbl);
 double svc = frameSize/LinkSpeed;
 if (i%4 == 0)
 {
  note_entry(qtbl);   //note arrival
  ProcessFrame(queues, svc, tbl, cwtbl, rtxtbl, drptbl, i);
 //Processes frames
  note_exit(qtbl);
 }
 else if (i%2 == 1)
 {
  note_entry(qtbl1);
  ProcessFrame1(queues, svc, tbl1, cwtbl1, rtxtbl1, drptbl1,
i);    //Processes frames
  note_exit(qtbl1);
 }
 else
 {
  note_entry(qtbl2);
  ProcessFrame2(queues, svc, tbl2, cwtbl2, rtxtbl2, drptbl2,
i); //Processes frames
        note_exit(qtbl2);   //note departure
 }
    cnt--;
    if(cnt == 0)
        set(done);     //if last arrival, signal
}

void source(int i)
{
 char procname[32];
 sprintf(procname, "source %d", i);
 create(procname);
 int stdone;
 CollRateEst(i);    /*Start the collision rate estimator for
this source*/
 do
 {
  hold(expntl(IATM));  /*Generate packets according to an
exponential distribution */
        cust(i);    //initiate process cust
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing()    //Check state of medium
{
  create("MediumSensing");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime);  //Wait for an AIFS interval
   st = state(busy);
   if (st==2)
```

```
    set(idle);   /*If medium is free for an AIFS interval set
idle event */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing1()    //Check state of medium
{
  create("MediumSensing1");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime1);  //Wait for an AIFS1 interval
   st = state(busy);
   if (st==2)
    set(idle1);   /*If medium is free for an AIFS1 interval
set idle event */
   else
    hold(aSlotTime);
  }
  else
   hold(aSlotTime);
  stdone = state(done);
 }
 while (stdone == 2);
}

void MediumSensing2()    //Check state of medium
{
  create("MediumSensing2");
 int st;       //State of medium
 int stdone;
 do
 {
  st = state(busy);
  if (st == 2)
  {
   hold(aAIFSTime2);   st = state(busy);
   if (st==2)
    set(idle2);
   else
```

```
    hold(aSlotTime);
   }
   else
    hold(aSlotTime);
   stdone = state(done);
  }
  while (stdone == 2);
 }

 void ACKgenerator()     /*Determines whether or not an ACK can
 be sent */
 {
  create("ACKgenerator");
  set_priority(2);
  int stdone;
  double ACKsize = ACKLgth+ PHYHdr;
  double svc = ACKsize/LinkSpeed;
  record(ACKsize,frmtbl);
  do
  {
   wait(finxmission);
   if (xmitting==1 || xmitting==0)  /*Generate an ACK iff one
 station is transmitting */
   {
    hold(aSIFSTime);
    set(busy);
    use(s, svc);   //Transmit the ACK frame
    clear(busy);
    set(ACK);
    set(go);
   }
   else
   {
    clear(ACK);
    hold(aSIFSTime);
    set(go);
   }
   stdone = state(done);
  }
  while (stdone == 2);
 }

 void CollRateEst(int k)  /*Estimates the collision rate per
 station */
 {
  char procname[32];
  sprintf(procname, "CollRateEst %d", k);
  create(procname);
  int stdone;
  int i;
  do
  {
   for (i=0; i<ObsvSlots; i++)
    hold(aSlotTime);
   if (PktsSent[k]!= 0)
    CollRate[k] = (collisions[k]*1.0)/PktsSent[k];
   else
    CollRate[k] = 0;
   /*The equation below is based on page 3 of the AEDCF paper.
 alpha = 0.8 */
   AvgCollRate[k] = (0.2* CollRate[k]) + (0.8* AvgCollRate[k]);
   record(AvgCollRate[k],prbtbl);
   ResetCollArrays();
   stdone = state(done);
  }
  while (stdone == 2);
 }
```

REFERENCES

[1] *IEEE 802.11 Std. 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE, 1999.

[2] *IEEE 802.11 Std. 802.11g, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 4: Further Higher Data Rate Extension in the 2.4 GHz Band*, IEEE, 2003.

[3] G. Berger-Sabbatel et al., "Fairness and its Impact on Delay in 802.11 Networks," *Proc. Of Global Telecommunications Conference*, 2004 (GLOBECOM 2004), 2004, pp. 2697-2793.

[4] G. Bianchi, "Performance analysis of the IEEE 802.11 Distributed Coordination Function," *IEEE Journal on Selected Areas in Communications*, March 2000, pp. 535-547.

[5] G. Bianchi and I. Tinnirello, "Kalman filer estimation of the number of competing terminals in an IEEE 802.11 network," *Proc. Of Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM 2003), 2003, pp. 844-852.

[6] P. Ferré et al., "Throughput Analysis of IEEE 802.11 and IEEE 802.11e MAC," *Proc. Of Wireless Communications and Networking Conference*, 2004, (WCNC 2004), 2004, pp. 783-788.

[7] P. Garg et al., "Using 802.11e MAC for QoS over Wireless," *Proc. 0f 2003 IEEE International Conference on Communication* (ICC 2003), 2003, pp. 537-542.

[8] A. Grilo et al., "A Scheduling Algorithm for QoS Support in IEEE802.11e Networks," *IEEE Wireless Communications*, June 2003, pp. 36-43.

[9] A. Lindgren et al., "Evaluation of Quality of Service Schemes for IEEE 802.11 Wireless LANs," *Proc. Of Local Computer Networks* (LCN 2001), 2001, pp. 348-351.

[10] S. Mangold et al., "Analysis of IEEE 802.11e for QoS Support in Wireless LANs," *IEEE Wireless Communications*, Dec. 2003, pp. 40-50.

[11] K. Medepalli and F.A. Tobagi, "Throughput Analysis of IEEE 802.11 Wireless LANs using an Average Cycle Time Approach," to be published in *Proc. Of Global Telecommunications Conference*, 2005 (GLOBECOM 2005),

[12] W. Pattara-Atikom et al., "Distributed Mechanisms for Quality of Service in Wireless LANs", *IEEE Wireless Communications Magazine*, June 2003, pp. 26-34.

[13] D. Pong and T. Moors, "Call Admission Control for IEEE 802.11 Contention Access Mechanism," *Proc. Of Global Telecommunications Conference*, 2003 (GLOBECOM 2003), 2003, pp. 174-178.

[14] J.d.P. Pravón and S. Shankar N, "Impact of Frame Size, Number of Stations and Mobility on the Throughput Performance of IEEE 802.11e," *Proc. Of Wireless Communications and Networking Conference*, 2004, (WCNC 2004), 2004, pp. 789-795.

[15] J.W. Robinson and T.S. Randhawa, "Saturation Throughput Analysis of IEEE 802.11e Enhanced Distributed Coordination Function," *IEEE Journal on Selected Areas in Communications*, June 2004, pp. 917-928.

[16] L. Romdhani et al., "Adaptive EDCF: enhanced service differentiation for IEEE 802.11 wireless ad-hoc networks," *Wireless Communications and Networking Conference*, 2003, (WCNC 2003), pp. 1373-1378.

[17] M. Wentink et al., "HCF Ad Hoc Group Recommendation – Normative Text to EDCF Access Category," Mar. 2002; http://grouper.ieee.org/groups/802/11/Documents/DocumentHolder/2-241.zip

[18] Y. Xiao and J. Rosdahl, "Throughput and Delay Limits of IEEE 802.11," *IEEE Communication Letters*, Aug. 2002, pp. 355-357.

[19] Y. Xiao, "Saturation Performance Metrics of the IEEE 802.11 MAC," *Proc. Of 58th IEEE Vehicular Technology Conference* (VTC 2003), 2003, pp. 1453-1457.

[20] Y. Xiao, "Performance Analysis of IEEE 802.11e EDCF under Saturation Condition," *Proc. Of 2004 IEEE International Conference on Communication* (ICC 2004), 2004, pp. 170-174.

[21] Y. Xiao, "IEEE 802.11e: QoS Provisioning at the MAC Layer," *IEEE Wireless Communications Magazine*, June 2004, pp. 72-79.

[22] Y. Xiao et al., "Game theory models for IEEE 802.11 DCF in wireless ad hoc networks," *IEEE Radio Communications Magazine*, Mar. 2005, pp. S22-S26.

[23] K. Xu et al., "Performance Analysis of Differentiated QoS supported by IEEE 802.11e Enhanced Distributed Coordination Function (EDCF) in WLAN," *Proc. Of Global Telecommunications Conference, 2003* (GLOBECOM 2003), 2003, pp. 1048-1053.

[24]S. Xu, "Advances in WLAN QoS for 802.11: an Overview," *14th IEEE 2003 International Symposium on Personal, Indoor and Mobile Radio Communication Proceedings*, 2003, pp. 2297-2301.

[25] H. Zhu et al., "A survey of quality of service in IEEE 802.11 networks," *IEEE Wireless Communications Magazine*, August 2004, pp. 6-14.

VITA

Daniel Tangyi Fokum was born on August 4, 1979 in Kulundji, Kwango, Zaire (present day Democratic Republic of Congo). He was educated in local parochial schools in Cameroon, graduating from Sacred Heart College, Mankon in 1997. He studied Mathematics at the University of Buea, Cameroon, for a year before coming to the United States in 1998 to study Computer Science at Park College in Parkville, Missouri. He graduated from Park University magna cum laude in December 2000. His degree was a Bachelor of Arts in Computer Science with a minor in Mathematics.

After working for a year at Truman Medical Centers' in Kansas City, Missouri, Mr. Fokum began a master's program in network architecture at the University of Missouri-Kansas City. Upon completion of his degree requirements, Mr. Fokum plans to continue his education in computer science.