RESOURCE SHARING ON CSMA/CD NETWORKS IN THE PRESENCE OF NOISE

BY

DUANE EDWARD DINSCHEL

B.S., University of Illinois, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

ABSTRACT

Resource sharing on CSMA/CD networks can be accomplished by using window-control algorithms for bus contention. The window-control algorithms are designed to grant permission to transmit to the station with the minimum contention parameter. Proper operation of the window-control algorithm requires that all stations sense the same state of the network in each contention slot.

Noise causes the state of the network to appear as a collision. False collisions can cause the window-control algorithm to terminate without isolating any stations. This thesis develops a two-phase window-control protocol and approximate recurrence equation with noise as a parameter to improve the performance of the window-control algorithms in the presence of noise. The results are compared through simulation, with the approximate recurrence equation yielding the best overall performance.

Noise is even a bigger problem when it is not detected by all stations. In such cases it is possible for the window boundaries of the contending stations to become out of phase. Consequently, it is possible to isolate a station other than the one with the minimum contention parameter. To guarantee proper isolation of the minimum, a broadcast phase must be added after the termination of the algorithm. This thesis discusses the protocol required to correct the window-control algorithm when noise is not detected by all stations.

ACKNOWLEDGEMENTS

The author would like to thank his advisor, Professor Benjamin W. Wah, for his help and direction in conducting this research.

The author also wishes to thank NASA (grant NAG 1-613) and NSF (grant DMC 85-19649) for their support of this work.

Finally, the author would like to thank his family and friends for their support and encouragement.

TABLE OF CONTENTS

PAGE

1. INTRODUCTION

1.1. Resource Sharing Issues

Physical limitations in the size and speed of current semiconductor technologies, combined with the availability of inexpensive processors and interconnection networks, have spurred great interest in distributed computing. One problem of distributed computing is to get the maximum throughput in the system. Chu, Holloway et al. show that a *saturation effect* occurs as more processors are added to a system [1]. In other words, the throughput of a system increases as more processors are added, but decreases after a certain point due to interprocessor communication overheads. Experience shows that a system's computing power normally increases by only a factor of 0.8 for each additional processor [2].

Although there are several essential issues involving multiprocessor systems [3], one critical consideration is a good resource sharing strategy. Such a resource sharing strategy should decompose work among the processors, in order that the maximum throughput of the system can be realized. There are two major issues to consider in regard to resource sharing: the job scheduling strategy and the type of interconnection network among the resources.

1.1.1. Scheduling Strategies

There are many different ways to categorize the various scheduling techniques. Probably the major distinctions are between static and dynamic techniques [4,5]. Static techniques make scheduling decisions based on fixed probabilities and queueing theory models [6]. Consequently, they are also known as *probabilistic* techniques. Static scheduling has the advantages of simplicity and low interprocessor communication overheads, but has a disadvantage in that it cannot account for changes in the workload distributions of the system. Ni and Hwang [6], and Chow and Kohler [4] have discussed various static scheduling strategies.

Dynamic techniques make scheduling decisions based on the current state of the system. Consequently, they are also often called *deterministic* techniques. Since dynamic scheduling techniques can account for changes in the workload distributions, they can potentially perform better than static techniques. However, they also introduce additional communication overheads to obtain the resource status. Chow and Kohler [4], Lo and Liu [7], Wah and Hicks [8], and Juang and Wah [9] have discussed various dynamic scheduling strategies.

Scheduling strategies can also be centralized or distributed. In centralized scheduling, one processor is responsible for scheduling all resources. It must receive status information from each processor, process that information according to a scheduling strategy, and send messages to processors to reschedule the tasks. Consequently, a lot of traffic flows in and out of the central scheduler, causing it to become a bottleneck of the system. Furthermore, the system is not fault-tolerant, because the scheduler is a critical component of the system. These drawbacks tend to favor distributing the scheduling intelligence among all the processors. The problems with distributed scheduling strategies are that they are more difficult to implement and require additional overheads for all processors to maintain the current status on the other processors. Nevertheless, the robustness of distributed scheduling strategies combined with their better performance, makes distributed scheduling the preferred choice in many multiprocessor applications [8, 9, 10, 11].

1.1.2. Interconnection Networks

The characteristics of the interconnection network among the resources are important factors in determining the load sharing strategy. There are basically three types of interconnection networks: shared bus, crossbar switch, and multistage dynamic blocking networks. The following examples assume homogeneous resources and processors.

The most basic and least expensive interconnection network involves a single shared bus that connects p processors to r resources (Figure 1.1). Each of the p processors must compete for using the bus before it can send a job to one of the r resources. There are two common methods of resolving bus contentions: token passing and carrier sense multiple access with collision detection, (CSMA/CD) [12]. This thesis focuses on the design of resource sharing strategies on CSMA/CD networks. Wah and Juang have studied resource scheduling issues on CSMA/CD networks for both single shared busses [13] and multiple shared busses [14].

The crossbar switch (Figure 1.2) is at the other end of the spectrum from the single shared bus. It is an expensive network in that connections must be possible from all p processors to all r resources. Such a network involves $p \cdot r$ switching elements for p processors and r resources. The crossbar switch is non-blocking, that is, connections from P_i to P_j do not inhibit connections from other processors to free resources. Therefore, crossbar networks do not impose any limitations on the resource scheduling algorithms. However, they are often used in small systems

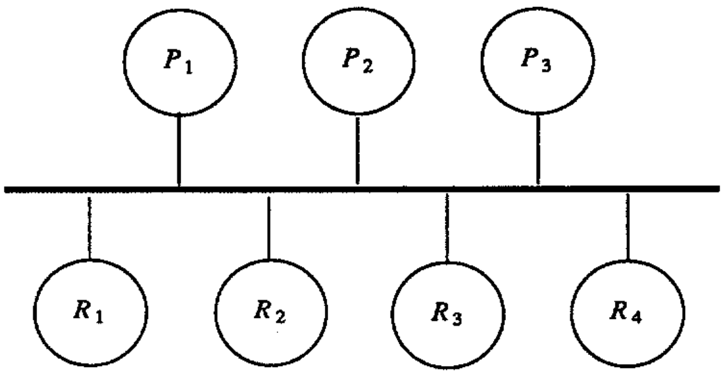

Figure 1.1. Single Shared Bus

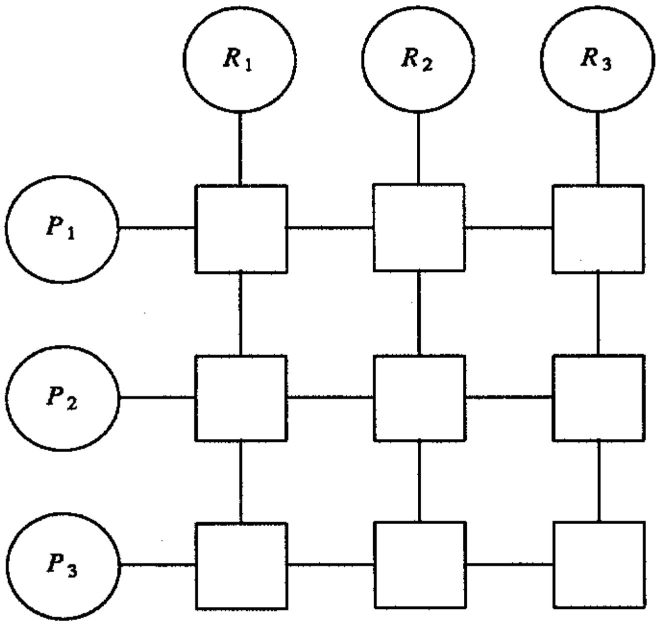

Figure 1.2. Crossbar Switch

due to their high cost of implementation.

Multistage dynamic networks offer a trade-off between cost and flexibility. They are constructed from a series of stages with switch boxes in each stage. The Omega network (Figure 1.3) is an example in this class. Each switch box can perform either a straight or exchange connection. Note that this network is less expensive than a crossbar switch in that there are only $O(n*\log_2 n)$ switch boxes to connect n processors to n resources. However, multistage networks are often blocking. Consider the scenario in Figure 1.3. The bold lines represent connections that have already been established. Suppose R_3 is busy and P_0 has a job ready. P_0 will be blocked from accessing the only free resource, R_1, due to the connection from P_2 to R_0.

Owing to the blocking effects in multistage networks, the design of the resource scheduling strategies is often more difficult. Wah has presented a distributed algorithm for multistage dynamic networks [8]. The algorithm consists of two phases: a resource phase and a request phase. In the resource phase, information concerning the number of free resources is passed from the resource side to the processor side. Then, in the request phase, the network propagates requests from the processors to the resources; each switch box makes an independent routing decision based on the results of the resource phase. If a request is blocked, it must be sent back to the exchange box in the previous stage for rerouting. Conflicts in a box due to two signals arriving simultaneously are resolved by a priority scheme. This algorithm is distributed in that exchange boxes operate independently rather than under a centralized control; it

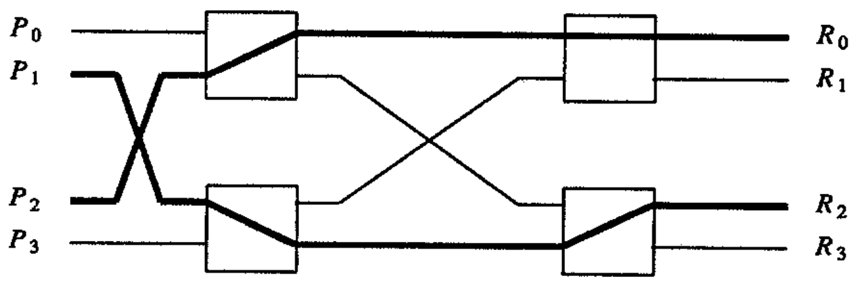

Figure 1.3. A 4-by-4 Omega Network

is deterministic in that requests are routed dynamically.

1.2. Objective of Thesis

This thesis focuses on the study of resource sharing strategies on CSMA/CD networks in the presence of noise. More specifically, it concentrates on window-control methods for bus contention on a particular CSMA/CD network, the Ethernet. The objective of this thesis is to develop a robust window-control algorithm for resource sharing on CSMA/CD networks in the presence of noise.

1.3. Organization

There are four chapters following the introduction. Chapter 2 provides background information on CSMA/CD networks and the various window-control algorithms that can be used to resolve bus contention for load balancing applications.

Chapter 3 deals with the problems of noise in relation to the window-control algorithms. It outlines the various types of noise and discusses how noise can affect the operation of a CSMA/CD network. The effects are divided into two classes, one in which the window boundaries of all stations remain synchronous, and the other in which the window boundaries of the stations may become asynchronized.

Chapter 4 contains the effects of noise on both the exponential backoff method and the window-control methods. It compares the effects of noise with regard to various numbers of stations, various probabilities of noise, and various window-control methods. This chapter deals with simulations for both synchronized and asynchronized window boundaries. Finally, this chapter outlines the implementation of a window-control algorithm in a real system in which noise is a factor.

Chapter 5 contains a summary of results and future research directions.

2. BACKGROUND

This chapter surveys previous work on CSMA/CD networks and window-control algorithms for resolving bus contention. The first section describes the operation of CSMA/CD networks and Ethernet's binary exponential backoff method for resolving bus contentions. The second section discusses the various window-control algorithms and the technique used to estimate the channel load.

2.1. CSMA/CD Networks

Carrier sense multiple access with *collision detection*, CSMA/CD was developed and patented by Xerox as part of its Ethernet local network [15]. The concept of CSMA/CD is fairly simple. Any station wishing to use the bus senses it to determine whether it is busy or idle. If the bus is idle, it may attempt to transmit; otherwise, it must wait until the bus is no longer busy. When two stations both detect that the bus is idle and simultaneously attempt to transmit, their transmitted signals will collide with each other. A collision detection mechanism is necessary to detect this event and abort the transmissions. In the worst case, assuming a signal propagation delay of τ from one end of the network to another, a station at one end of the network may attempt to transmit at time t and another station at the other end of the network at a time slightly less than $t+\tau$. The two transmissions will then collide, and the second station will detect the collision almost immediately. However, the first station will not detect the collision until τ units after the second station has transmitted. The total time from initial transmission to collision detection in the worst case is, therefore, 2τ. This amount of time is called a *contention slot*, and is the amount of time a station must listen to the bus after transmitting to guarantee that no collision will occur.

The fact that there are collisions leads to the issue of determining the time a station should wait before it retransmits. There are several solutions to this problem. The first is to transmit as soon as the station detects that the network is idle. This is known as *persistent* CSMA/CD [12]. However, under heavy traffic, this method works very poorly because stations constantly collide in their attempt to access the bus. Consequently, a *nonpersistent* protocol [12] may be used in which the stations always wait a random amount of time before retransmitting. This method does not work as well under light traffic. One compromise is to use a *p-persistent* protocol [12] in which the stations transmit immediately upon sensing the network idle with probability p and wait a random amount of time before retransmitting otherwise.

Ethernet uses the *truncated binary exponential backoff algorithm* to determine when a station can retransmit after detecting a collision [16]. According to the data link layer of the Ethernet specifications, a station has 16 attempts to transmit on the bus [16]. If it is unsuccessful, it waits an integral number of contention slots before retransmitting. The average delay before retransmission grows exponentially until n=10, but up to 15 retransmissions may be attempted. The number of slot times to delay before the n^{th} retransmission attempt is chosen as a uniformly distributed random integer, I, in the range $0 \le I \le 2^k$, where k=min(n,10). If all 16 attempts fail, the station quits transmitting and logs the event as an error.

The binary exponential backoff method has an advantage over the persistent methods for bus contention in that it can more effectively account for the system load [15]. However, its performance degrades as the number of stations increases. Furthermore, it is not possible to establish priority of one station over the others with regard to the use of the bus. Hence, it is not easy to implement a load balancing strategy in which the priority to transmit is based on the system workload. Both of these problems can be resolved by the window-control algorithm to be outlined in the next section.

2.2. Window-Control Algorithm

The purpose of the window-control algorithm is to identify the extremum from a set of physically dispersed random numbers called *contention parameters* [13]. The distribution of these numbers varies according to the load balancing scheme used. For example, the length of the job queue or a random number from a uniform distribution can be used as the contention parameter. Since procedures for isolating the minimum and maximum contention parameter are similar, this thesis discusses only procedures for isolating the minimum. Bus contention is resolved by granting permission to transmit to the station with the minimum contention parameter.

2.2.1. General Operation

B. W. Wah and J. Y. Juang have presented a window-control algorithm for determining priority on CSMA/CD networks [13]. It is a distributed algorithm in which each station maintains a window to decide whether it will retransmit in the future. All stations update their windows synchronously, each contention slot, according to the state of the network.

The algorithm assumes that all contention parameters lie in the interval (L,U]. The initial window is chosen with a lower bound, a, at L, an upper bound, b, at U, and a temporary upper bound, w, in (a,b]. The value of w is chosen according to the window-control protocol being used, and varies as a function of a and b. In each contention slot, all stations with contention parameters smaller than w transmit. If there is a collision, there must be at least two stations in the window, $(a,w]$; therefore, b is lowered to w in the next contention slot. If the state of the network is idle, the window must be empty; therefore, a is raised to w in the next contention slot. Otherwise, if there is a successful transmission, the algorithm terminates. This process is repeated until the minimum contention parameter has been isolated. The operation of the window-control algorithm for an individual station is outlined in Figure 2.1.

Operation of the window-control algorithm is best illustrated by an example (Figure 2.2) . Initially, each of the five stations generates a random contention parameter in the interval (L,U], and sets its window to $(L,w_1]$. During the first contention slot, stations 1, 2 and 4 transmit. Collision is detected, and the window is reduced to $(L,w_2]$. In the second contention slot no stations transmit. Therefore, the lower bound of the window is increased to w_2 and the upper bound of the window to w_3, resulting in a window $(w_2,w_3]$. Finally, in the third contention slot only station 2 remains-- the transmission is successful and the algorithm terminates.

Wah and Juang discuss the performance of several window-control algorithms in which the channel load and the distribution functions from which the contention parameters are generated are exactly known: binary-divide, dynamic programming, greedy window-control, and approximations to greedy window-control [13, 17]. These are outlined in the next sections and evaluated with respect to the presence of noise in the next chapter. In the following discussion, a is the current lower bound, b is the current upper bound, w is the upper bound of the next window, and n is the number of contending stations at the start of the algorithm.

2.2.2. Binary-Divide Window-Control

The simplest of the window-control methods is known as the binary-divide window-control. The new upper bound of the window, w, is set to the value $(a+b)/2$. If there is a collision, b is set to w; if the line is idle, a is set to w. The algorithm is reiterated until the minimum is isolated. The expected number of contention slots to resolve the collisions is $O(\log_2 n)$ [13].

```
procedure window_protocol_station_i;
/* procedure to find window boundaries for isolating one of the contending stations */
 /* window - function to calculate window size w,
  * random - function to generate local contention parameter,
  * transmit_signal - function to send signal to bus with other stations synchronously,
  * detect - function to determine state of the network,
  * x - local contention parameter,
  * lb - lower bound of interval containing minimum
  * ub - upper bound of interval containing minimum
  * contending - boolean to continue the contention process,
  * state- state of the network: collsion, idle, or success
  */
   [
   lb := L;
   ub := U;
   r := random(L,U);
   w := window(lb,ub);
   contending := true;
   while (contending) do [
          if (r ≥ lb and r ≤ w) then [
                     /* parameter is inside window, contend for bus */
                     transmit_signal();
                     /* test for unique station in the window */
                     state := detect;
                     if (state = collision) then
                                /* update upper bound of interval containing minimum */
                                ub := w;
                     else /* successful isolation of minimum */
                                return(lb,ub);
                     w := window(lb,ub); ]
          else[
                     state := detect();
                     if (state = idle) then
                                /* all parameters are outside window */
                                /* update lower bound of interval containing minimum */
                                lb := w
                     else
                                /* some other parameters are inside window, stop contending */
                                contending := false; ]
          ]
          return(failure);
   ]
```

Figure 2.1. Procedure illustrating the execution of the window-control algorithm by a single station [13]

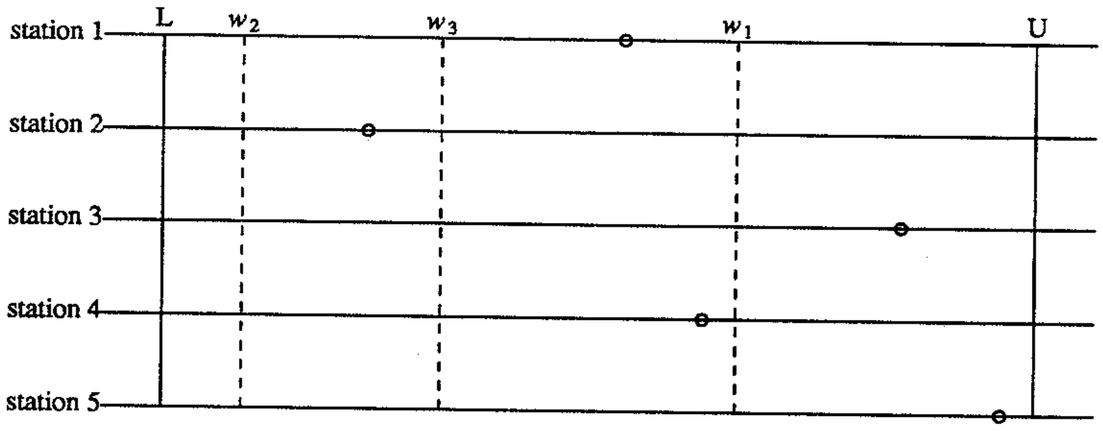

Fig. 2.2. Window-Control Algorithm [13]

2.2.3. Dynamic Programming Window-Control

A significant improvement to the number of contention slots to resolve collisions can be gained by using a dynamic programming window-control algorithm. The algorithm attempts to minimize the expected number of contention slots to resolve collisions, $M(a,b)$, given that there are n contention parameters in $(a,U]$, and that collision occurs in the current window. The following notations are first defined [13]:

$M(a,b)$:
The minimum expected number of iterations to resolve contention given that there are n contention parameters in $(a,U]$ and collision occurs in the current window $(a,b]$.

$g(w,n,a,b)$:
Probability of *success* in the next iteration if a window of $(a,w]$, $a < w < b$, is used.

$l(w,n,a,b)$:
Probability of *collision* in the next iteration if a window of $(a,w]$, $a < w < b$, is used.

$r(w,n,a,b)$:
Probability of *no transmission* in the next iteration if a window of $(a,w]$, $a < w < b$, is used.

It follows directly from the above definitions that

$$l(w,n,a,b) + g(w,n,a,b) + r(w,n,a,b) = 1. \tag{2.1}$$

Functions $l(w,n,a,b), r(w,n,a,b)$, and $g(w,n,a,b)$ are conditional probabilities based on the condition that there was a collision in the last iteration of the protocol. This implies that there must be at least two contention parameters within the boundary $(a,b]$, and that all contention parameters must be greater than a. This condition can be designated as event A.

Three mutually exclusive events can occur by choosing w, $a < w < b$, which can be identified as follows [13]:

$B = \{$exactly one of the x_i's is in $(a,w]$, given that all x_i's are greater than $a\}$

$C = \{$no x_i is in $(a,w]$, given that all x_i's are greater than $a\}$

$D = \{$more than one x_i is in $(a,w]$, given that all x_i's are greater than $a\}$

From these events, the conditional probabilities can be expressed as [13]:

$$g(w,n,a,b) = Pr(B \mid A) = \frac{Pr(A \cap B)}{Pr(A)} \tag{2.2}$$

$$r(w,n,a,b) = Pr(C \mid A) = \frac{Pr(A \cap C)}{Pr(A)} \tag{2.3}$$

The set $A \cap B$ represents the event that exactly one x_i is in $(a,w]$, one x_i is in $(w,b]$, and all other x_i's are greater then w. The set $A \cap C$ represents the event that at least two x_i's are greater than w.

Let $F_i(x)$ be the distribution function governing the generation of x_i. Wah and Juang presented the formulas for g,r and $Pr(A)$ for the general case of arbitrary independent distribution functions [13]. The functions g and r can be simplified by assuming uniform distribution functions for each station. $F_i(x)$ is then equal to x. These functions can be expressed as:

$$Pr(A) = \frac{(1-a)^n - n(b-a)(1-b)^{n-1} - (1-b)^n}{(1-a)^n} \tag{2.4}$$

$$g(w,n,a,b) = \frac{n(w-a)[(1-w)^{n-1} - (1-b)^{n-1}]}{Pr(A)\cdot(1-a)^n} \tag{2.5}$$

$$r(w,n,a,b) = \frac{(1-w)^n - n(b-w)(1-b)^{n-1} - (1-b)^n}{Pr(A)\cdot(1-a)^n} \tag{2.6}$$

The function, $M(a,b)$, can then be formulated recursively as follows:

$$M(a,b) = \min_{a<w<b} \left\{ 1 + g(w,n,a,b)\cdot 0 + l(w,n,a,b)\cdot M(a,w) + r(w,n,a,b)\cdot M(w,b) \right\} \tag{2.7}$$

A boundary condition δ must be set to terminate the algorithm after a reasonable number of iterations. By assuming that all contention parameters differ by at least δ, contention can always be resolved in one step when the window size is smaller than δ. Therefore, the following boundary condition is included:

$$M(a,b) = 1 \qquad \text{for } all \ (b-a) < \delta \ . \tag{2.8}$$

The dynamic programming algorithm requires an average of only 2.4 contention slots to resolve collisions. What is significant is that this value remains constant as the number of contending stations increases. Figure 2.3 illustrates the performance of the binary exponential backoff method as compared to the dynamic programming method.

There is an extensive computational overhead associated with this method. This overhead can be overcome in one of two ways. The first is to precompute $N(a,b)$ for a truncated dynamic programming algorithm and store the

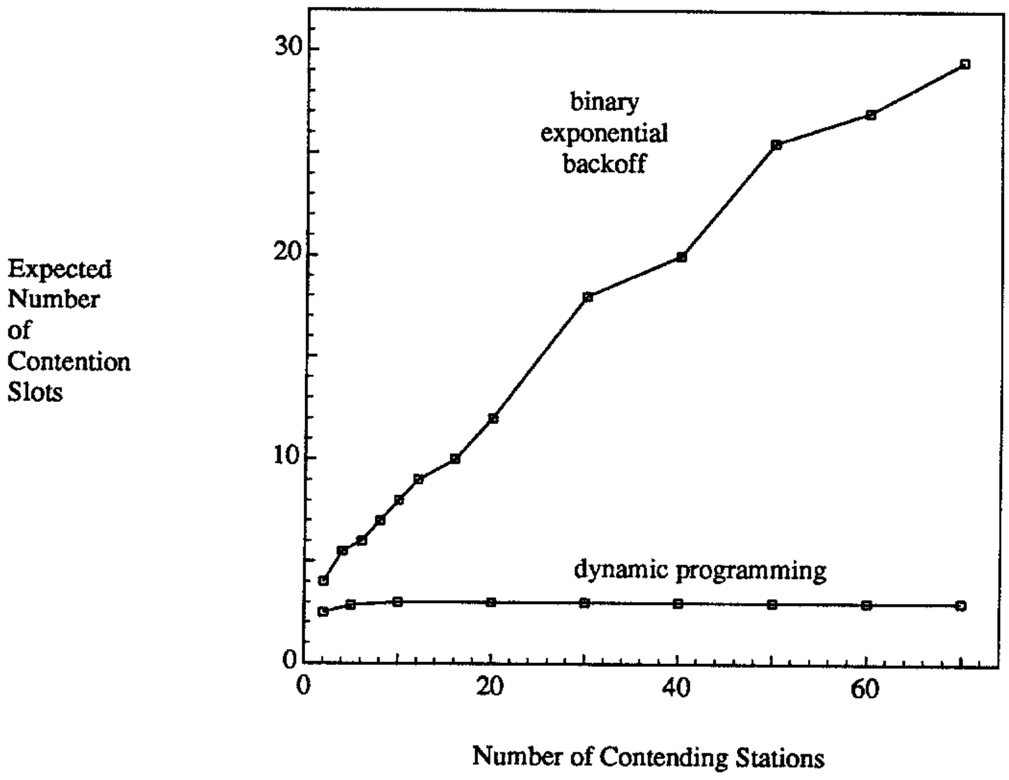

Figure 2.3. Binary Exponential Backoff vs. Dynamic Programming Window-Control

results of optimal w's for each (a,b) pair in a table. When $(b-a)<\delta$, a binary-divide algorithm can be used to complete the iterations. This method was used in the simulation results shown in Figure 2.3. The other method to reduce computation time involves using approximations to the dynamic programming algorithm as discussed in the next section.

2.2.4. Greedy Window-Control and Approximations

The computation time of the dynamic programming algorithm can be greatly reduced by finding a window to maximize the success probability in just the next iteration rather than in all possible future iterations. This technique requires an average of only 2.7 contention slots for bus contention [13]. Similar to the dynamic programming window-control algorithm, the overhead is also independent of the number of contending stations.

Computation time can be further reduced by using an approximation to the greedy window control scheme that calculates w directly from the following formula [13]:

$$w = \frac{-C - \sqrt{C^2 - 4D}}{2} \tag{2.9}$$

where

$$C = \frac{-(n-1)[F(a) + F(b)] + 2}{n} \tag{2.10}$$

$$D = \frac{F(a) + F(b) + (n-2)\cdot F(a)\cdot F(b)}{n} \tag{2.11}$$

This calculation of w has been shown to have a performance comparable to the optimal greedy window-control algorithm [13].

A final simplification is mentioned by Juang in his Ph.D. dissertation [17]. He showed that the window used in the first contention slot is critical to the overall performance. The ideal window for this first slot is $1/n$. After the first slot, a binary-divide scheme may be used to complete the algorithm. Again, the performance is comparable to that of the pure dynamic programming window-control algorithm.

2.2.5. Load Estimation

If the load of the channel is not exactly known, it must be estimated from the distributions of contention parameters and statistics of previous channel activity [13]. Wah and Juang have presented a method for estimating load, based on the windows used to isolate the successful stations in previous contention slots [13]. The method computes a moving average based on all previous windows, w(1), w(2), ..., w(t). This moving average, $w_{mv}(t)$, is computed according to the following formula [13]:

$$w_{mv}(t) = \frac{w_{mv}(t-1) + w(t)}{2}. \tag{2.12}$$

The estimation of the number of stations, $n_e(t)$, is approximated by the formula [13]:

$$n_e(t) = \frac{-1}{\log(1-w_{mv}(t))}, \qquad 0 \le w_{mv}(t) < 1 \tag{2.13}$$

This estimate of n can be used in the approximate greedy window-control algorithm, yielding a performance very close to that when the channel load is exactly known. However, the estimate also assumes that there is no noise on the network. The effects of noise on load estimation will be discussed in Chapter 3 and illustrated by simulation in Chapter 4.

3. EFFECTS OF NOISE ON CSMA/CD BUS CONTENTION

One major problem in any communication network is noise. Noise on a network may be defined as any unwanted signal present in the transmission medium. This chapter discusses the types of noise, how noise can affect the physical operation of an Ethernet network, and how noise affects the various window-control algorithms for bus contention.

The effects of noise on the window-control algorithm are divided into two cases in which noise is either detected by all stations or by just some of the stations. In the first case, the windows of different stations remain in phase, but in the second case the windows may become out of phase. All window-control algorithms assume that the channel load is exactly known and that noise is independent for each contention slot. The last section discusses the effects of noise when these assumptions are invalid.

3.1. Types of Noise

There are basically four types of noise: thermal noise, intermodulation noise, crosstalk, and impulse noise [15]. Thermal noise is caused by the thermal agitation of electrons in the transmission medium. It varies as a function of temperature, but is otherwise fairly predictable. Thermal noise is often referred to as white noise [15]. Most communication systems take thermal noise into account in their design and use signals with sufficient strengths such that thermal noise has little effect. Although thermal noise limits the effective bandwidth of any communication medium, it normally does not cause signals to be misinterpreted by the receiver. Consequently, thermal noise is not a factor in regard to the window-control algorithm.

Intermodulation noise is due to signals of different frequencies producing a new signal that is the sum or difference of a multiple of those frequencies. Intermodulation noise is important when two or more simultaneous transmissions of different frequencies are attempted on the line. Since all transmission attempts in the window-control algorithms are made at the same frequency, intermodulation noise is also not a factor.

Crosstalk can be a problem depending on the communication medium used. It is rare in coaxial cable and nonexistent in optical fiber, but might occur in twisted-pair lines. *Crosstalk* is electric coupling between lines such that the signal on one line affects the signal on the other. Nevertheless, the magnitude of the noise produced by crosstalk is relatively small and can usually be taken into account by the system designer [15].

Impulse noise, however, is a major factor in producing false signals. Impulse noise is an unpredictable burst of electrical energy that lasts a random amount of time. It can be caused by lightning, electromagnetic radiation, or any sudden pulse of energy. Such noise can often be found in harsh environments, such as those encountered in the military. Unlike the other types of noise, the randomness in the occurrence, duration, magnitude and nature of impulse noise makes it impossible to predictably filter out. Furthermore, impulse noise is usually of sufficient strength to cause logical 1's to be misinterpreted as logical 0's and logical 0's to be misinterpreted as logical 1's. It can also cause a whole block of transmitted signals to be altered dramatically. As a result, impulse noise has the greatest net effect on transmission on CSMA/CD networks and will be the main type of noise considered in this thesis.

3.2. Effects of Noise on Ethernet Network

To determine the effects of noise on bus contention on Ethernet networks, it is necessary to discuss the encoding of transmission signals on the interpretation of network status by stations, and the effects of noise on this interpretation. According to the physical layer specifications of Ethernets [16], an idle transmission line is set to a value of 0 volts. Data is encoded using *Manchester Code*, in which a logical 0 is represented by a transition from -0.225 V to -1.825 V, and a logical 1 is represented by a transition from -1.825 V to -0.225 V. Bits are transmitted at a frequency of 10 MHz, and the transitions are set to occur in the middle of each bit. Figure 3.1 illustrates the Ethernet specifications for its Manchester encoding scheme.

Any two different transmitting stations are likely to have their clocks skewed, and if simultaneous transmissions are attempted by both, the receiver will likely detect a collision either by a voltage level that is lower than it should be or by the lack of a detected transition in the data at the specified time. The receiver may also detect unexpected transitions less than 50 ns (1/2 bit time) apart. Figure 3.2 illustrates the results of a collision. Note that in the case of a collision, the overall high voltage is reduced from -.225 V to -.45 V and that the low voltage occurs at -3.65 V which is well below -1.825 V. Furthermore, there are two transitions less than 50 ns apart.

When collision is detected by the transmitting stations, the stations do not abort transmission immediately, but continue to transmit for a period of time to guarantee that the collision is heard by all stations [16]. The resultant

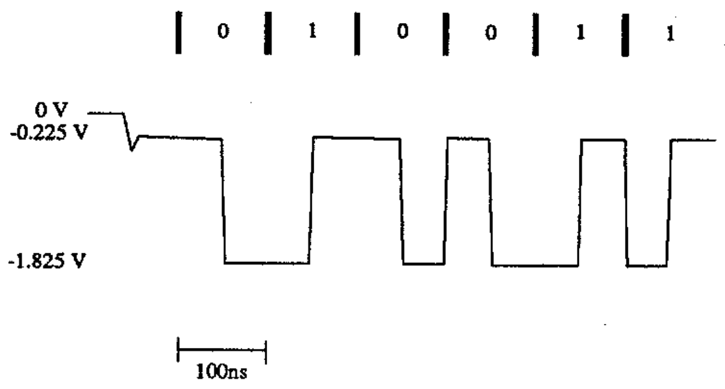

Figure 3.1. Manchester Encoding

transmission is known as a *jamming signal.*

What happens to a normal signal in the presence of noise? Most impulse noises introduced into the system will cause the voltage levels to be incorrect or cause the 10 MHz transition not to be detected. For example, a -1 V surge for 200 ns would cause the voltage levels to drop to -1.225 V and -2.825 V triggering a false collision, as illustrated in Figure 3.3. Figure 3.3 also shows how noise could interfere with the signal in such a way as to mask out the expected transition due to the 10 MHz clock. In this sense, it is relatively easy for noise to generate a false collision on the bus.

A false idle contention slot can occur only if there is a positive voltage surge that cancels out the signals present on the bus. As it is, a +2 volt surge will cause all voltages on the line to be greater than or equal to 0 when only one station is transmitting. Hence, it is necessary for the receiving station to recognize that voltages greater than 0 can be caused only by noise. The current Ethernet specifications [16] do not address this case, although minimal hardware can be added to the transceiver to detect positive voltages on the line. Under these conditions, the only way a false idle contention slot can occur is when the noise is the exact inverse of the signal, resulting in a net voltage of 0. This event is extremely unlikely for any signal, and hence, the probability that either a collided or

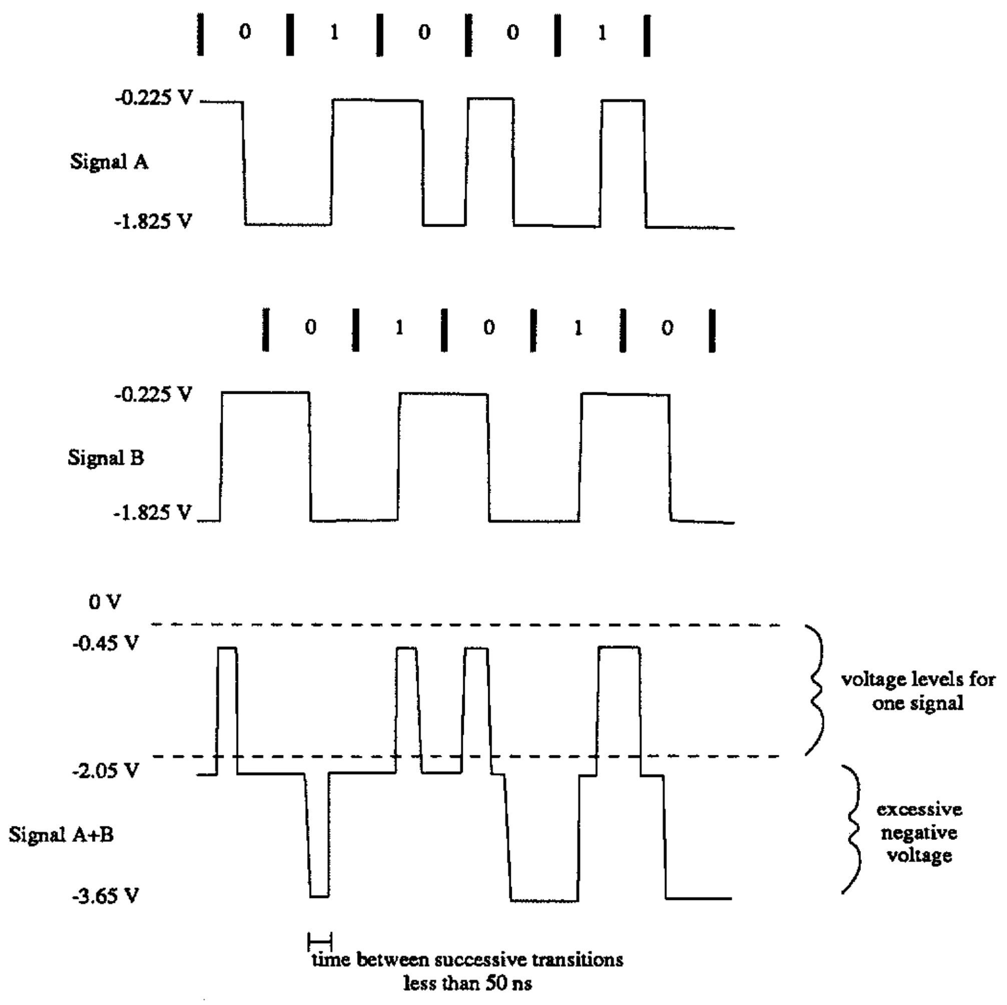
0
1
0
0
1
-0.225 V
Signal A
-1.825 V
0
1
0
1
0
-0.225 V
Signal B
-1.825 V
0 V
-0.45 V
-2.05 V
Signal A+B
-3.65 V
voltage levels for one signal
excessive negative voltage
time between successive transitions less than 50 ns

Figure 3.2. Collision of Two Signals

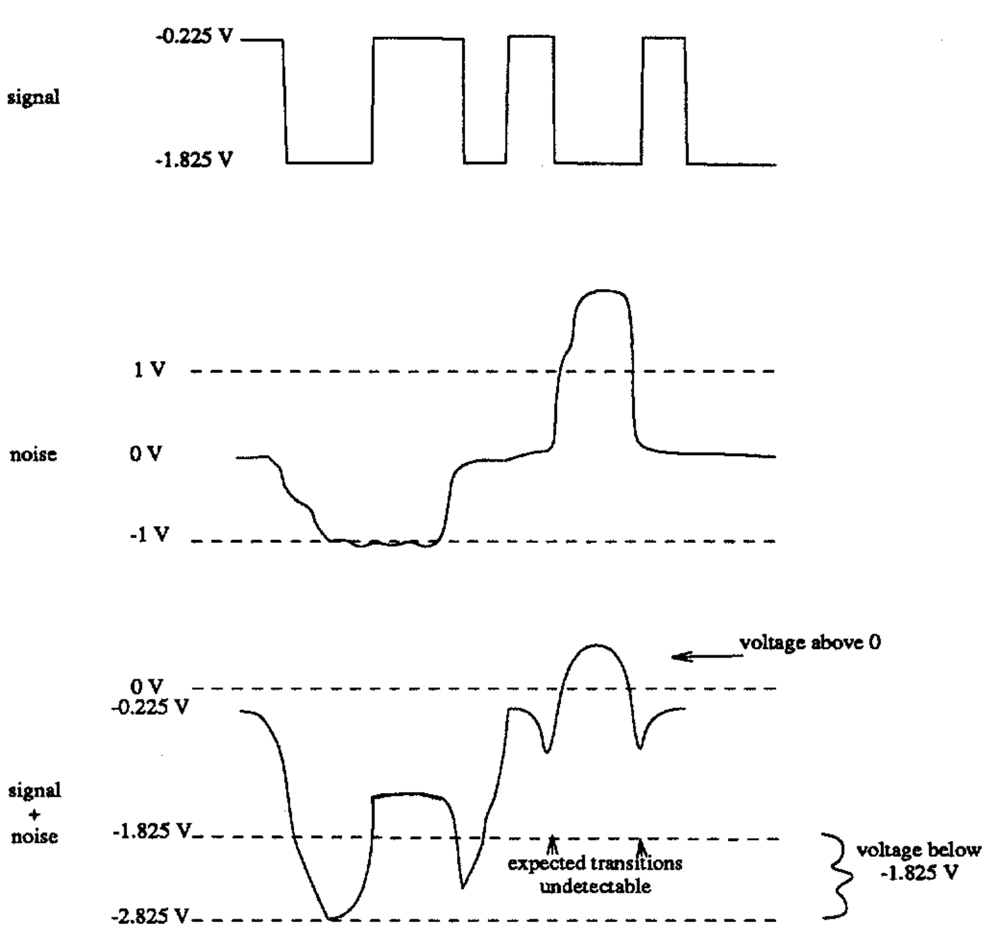

Figure 3.3. Signal with Added Impulse Noise

successful contention slot will falsely be detected as an idle slot is negligable.

It is also unlikely that a successful contention slot is falsely detected. In the case of an idle contention slot, a false success could be triggered by an arbitrary 10 MHz signal that varies between -.225 V and -1.825 V. Likewise, a collision could falsely be detected as a success when noise cancels out the collision in such a way as to produce a 10 MHz signal that varies within the -.225 to -1.825 voltage range. Both of the above are extremely unlikely since first, random noise is likely to have a voltage outside the range between -.225 V and -1.825 V, and second, even if the voltage levels are valid, the noise is likely to have a frequency other than 10 MHz. The case of a collision being falsely detected as a success is even more unlikely, since either transmitting station will detect a signal different from the one it transmitted, and enforce the collision with a jamming signal. In the worst case, if an idle contention slot is falsely treated as a successful transmission, all stations will be expecting the nonexistent successful station to transmit, resulting in an idle channel. This case can be logged as an error, and the stations can restart the algorithm.

All in all, the probabilities of a false idle slot or a false successful slot are extremely small, while the probability of a false collision in the presence of impulse noise is very likely. In modeling the effects of noise, a fourth state, noise, could be added to give additional information regarding the state of the network. However, noise is only distinguishable from a true collision when there are voltages greater than 0 on the transmission line. In all other cases, noise is indistinguishable from collisions. Therefore, for the purposes of evaluations, all noise will be treated as resulting in false collisions.

There is still the issue of whether all stations on the network will detect noise when it occurs. The probability that this is true is high for noise on the global bus since Ethernet is designed to allow collisions to be detected by all stations through use of a jamming signal. It is very unlikely that the jamming signal will be misinterpreted as a false success or no transmission since the transmitting station is also listening to the line and can continue to issue the jamming signal if it detects that the signal is being cancelled out by noise. This leaves the problem, however, of noise occuring while no stations are transmitting. In this case, no station would issue a jamming signal to amplify the false collision. Whether a station detects a collision will then be dependent on the sensitivity of a stations's transmission-line sensing equipment.

Unfortunately, one cannot guarantee that all stations will always detect all noise whenever it occurs. Consider the Ethernet network in Figure 3.4. If noise is more prevalent only on the right side of the repeater, there will be

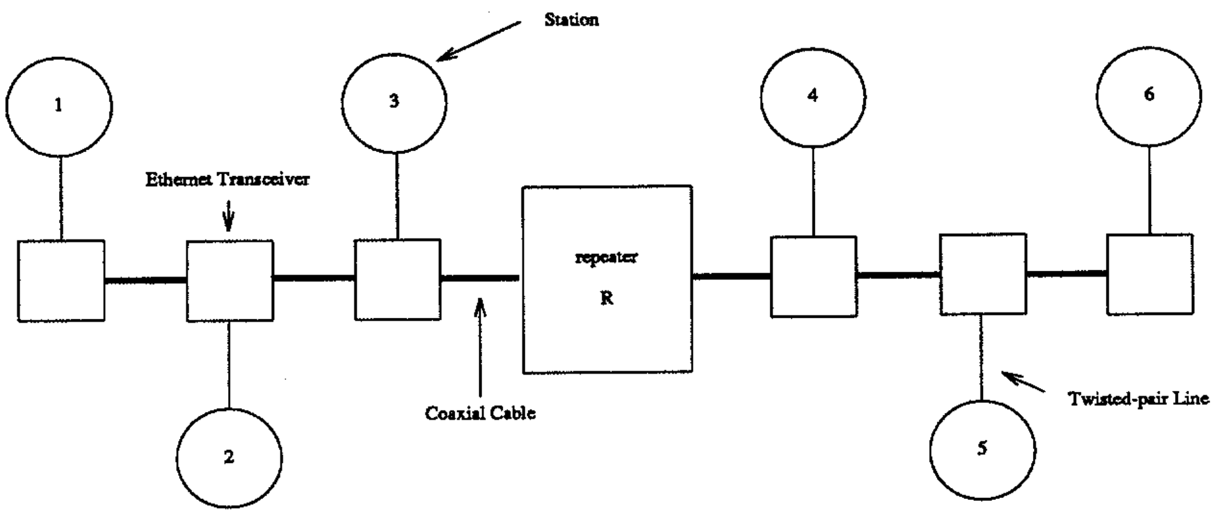

Figure 3.4. An Ethernet Network

occasions when stations 4,5 and 6 detect noise while stations 1,2 and 3 do not. If station 4 has a noisy connection to the network, it might detect noise, while the others do not. There might also be the case where the collision and noise detection mechanisms are more sensitive in some stations than others. Basically, any condition that causes noise to be localized to one or more areas of the network, or any discrepancy in transmission-line sensing equipment of some stations, may result in the stations detecting different states of the network during a given contention slot.

3.3. Effects of Noise on Window-Control Algorithms

The previous section discusses the fact that noise will almost always result in the detection of a collision by the stations on the network. If there is already a collision, noise has no effect. But if there is no collision, some or all of the stations may detect a false collision due to the noise, and inappropriately adjust their windows.

Analysis of the noise problem can be divided into two cases. The first case assumes that noise is always detected by all stations, and that all stations treat noise as a collision. Provided that the stations start with the same windows in the first contention slot, their windows will remain synchronized with respect to each other, since they always detect the same state of the network. The second case is more general and assumes that a subset of the stations detect noise while others do not. As a result, it is possible for stations to have different window boundaries in a given contention slot. These two cases are discussed separately in the following sections.

3.3.1. Noise Detected by All Stations

The assumption that the windows remain synchronized is necessary in developing a theoretical analysis for the performance of the window control algorithms under the presence of noise. The following sections discuss the effects of noise on the window-control algorithm along with the development of two new window-control algorithms that attempt to overcome the noise problem-- the two-phase algorithm and an approximate recurrence relation.

3.3.1.1. Original Window-Control Algorithm

When noise is apparent, windows in the original window-control algorithm may be updated incorrectly. Consider the scenario in Figure 3.5, where there is a window with no contention parameters in it and noise occurs in that slot. After the first real collision, the upper bound of the window is reduced to w_1, but all contention parameters are greater than w_1. Now suppose there is a burst of noise in the next contention slot that causes all stations to detect a false collision and reduce the upper bounds of their windows to w_2. In successive iterations, idle contention slots are detected and cause the upper bounds of the windows to increase to w_3,w_4,w_5, and so on, but all are less than w_1. In any event, the upper bound of the windows will never become greater than w_1, and the contention protocol will con-

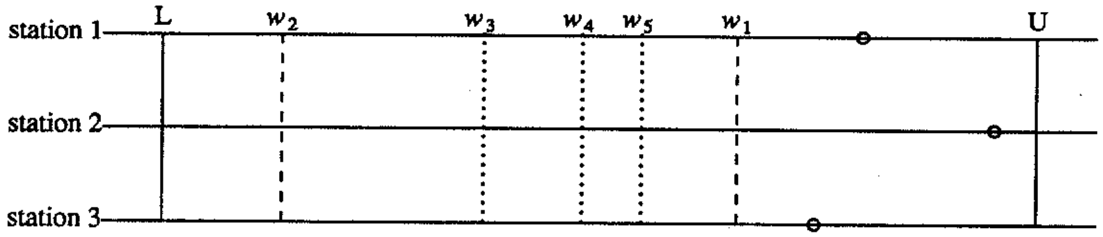

Figure 3.5. Window-Control with Noise

tinue to iterate indefinitely.

To correct this problem, it is necessary to terminate the protocol after the window has been reduced to a certain size. The termination point can be set to the value δ, the resolution of the contention parameters, as defined in Chapter 2. If the window is reduced to δ, the algorithm must be restarted. The number of contention slots to resolve the algorithm will then depend on the probability that collision is resolved before the window is reduced to a size δ and the associated probability for this event. If the probability of noise is high, then it is highly likely that the window will be reduced to a size less than δ without isolating the minimum. If δ is small, then extensive overheads are incurred before false collisions are detected. With respect to the window protocols outlined in Chapter 2, the number of contention slots to resolve collision has to be modified by the probability that collision is resolved before the window is reduced to a size δ. Let M be the number of contention slots to either isolate the minimum or reduce the window size to δ, and S the probability of success before this event occurs. Define T as the total number of contention slots to resolve contention. T is then formulated as follows:

$$T = S\cdot M + S(1-S)\cdot 2M + S(1-S)^2\cdot 3M + S(1-S)^3\cdot 4M + \cdots \tag{3.1}$$

$$T = SM\cdot\sum_{i=1}^{\infty} i\cdot(1-S)^{i-1} \tag{3.2}$$

$$T(1-S) = SM\cdot\sum_{i=1}^{\infty} i\cdot(1-S)^{i} \tag{3.3}$$

Subtracting (3.3) from (3.2) yields:

$$TS = SM\cdot\sum_{i=1}^{\infty}(1-S)^{i} \tag{3.4}$$

The summation in (3.4) is a standard geometric distribution that reduces to $1/S$ [18]. Therefore, Equation (3.4) reduces to:

$$T = M/S \tag{3.5}$$

3.3.1.2. Two-Phase Algorithm

The original window protocol suffers from the problem that once the upper bound of the window is reduced to a value smaller than the smallest contention parameter, it is not possible to recover until the window size is reduced to a size less than δ. To alleviate this problem, one can use a two-phase algorithm that adds a check phase each time no transmission is detected. If no transmission is detected in a given contention slot, then the other half of the window is tested to see whether it contains any contention parameters. If there is a collision, the protocol carries on as normal, but if no transmission is detected again, then the window's upper bound is increased to its previous value, while the lower bound is increased to the current upper bound. Referring to Figure 3.5, the two-phase algorithm would modify the situation as follows:

Slot 1) Collision:
window reduced to $(L, w_1]$.
Slot 2) Collision due to noise, although no transmission should be detected:
window reduced to $(L, w_2]$.
Slot 3) No transmission:
window raised to $(w_2, w_1]$.
Note: In the original window protocol, the window used is $(w_2, w_3]$.
Slot 4) Second no transmission:
window raised to $(w_1, w_0]$.
Slot 5) Protocol continues normally, unless further noise develops.

The state of no transmission is used to correct the windows, since whenever no transmission occurs, it is guaranteed, based on the assumptions discussed earlier, that the transmission line is definitely in that state. Therefore, whenever no transmissions occur twice in sequence, it is guaranteed that no contention parameters remain within the lower and upper bounds defined for the window. As a result, it is necessary to retract the lower and upper bounds to a different region of the contention parameters. This process is repeated until a region with a collision is found. Since a collision may be due to noise, the occurrence of collision in a contention slot cannot be used to permanently fix the upper bound of the window. If the probability of noise is low, then most of the collisions are due to simultaneous transmissions in a window. The algorithm is set to terminate when a successful transmission is detected, since it is assumed that there will be no false detections in this case.

Inherently, the two-phase algorithm implies need of additional memory to store previous upper window boundaries. However, these bounds are normally defined in the implementation of the window-control algorithms, as they are precomputed at design time and looked up in real time. In the case in which the window boundaries are computed in real time, such as the binary-divide algorithm, the maximum number of stored window boundaries is bounded by the difference between the number of collisions and the number of no transmissions.

This can be best illustrated by the binary-divide window control. Assume an initial window size of 1 and a resolution $\delta = 2^{-n}$. Each collision causes the window size to decrease by a factor of two and increases by one the number of upper bounds that must be stored in memory. However, each no transmission retrieves a previous value of the upper bound, and therefore decreases the number of upper bounds to be stored by one. In the worst case there will be n successive collisions, resulting in n upper bounds stored. The next collision will force the algorithm to restart, as the maximum number of collisions is bounded by n. Although each collision decreases the window size by a factor of two, each no transmission increases the window size by a factor of two only after the second no transmission is detected. It takes only one collision to decrease the size of the window, but it takes at least two no transmissions to increase the window size. Unless there is a sequence of n collisions, the window shrinks at a rate that is faster than the net growth of the memory stack and, therefore, the maximum amount of memory needed to store the upper bounds is n. A similar result holds true for the dynamic-programming protocol, except that during collisions the window is reduced by at least a factor of two. Hence, the required memory is likely to be less than n.

There is additional computational overhead to check the other half of the window in the two-phase protocol. Although the probability of success is increased in the two-phase protocol over the original protocol, the number of contention slots to reduce the window size to δ is also increased. Therefore, the net effect on the total number of contention slots to resolve contention, as defined by Equation 3.5, is uncertain. A comparison of the performance of the two-phase and the original protocols is included in the next chapter.

3.3.1.3. Heuristic Window-Control with Noise

The optimal window for each contention slot can be determined by including the probability of noise as a parameter in the dynamic programming window-control algorithm. First, one must consider the effects of noise on state detection and window boundaries. No information concerning the state of the network can be gained during the contention slot in which noise occurs. Windows can be updated with certainty after detection of a successful transmission or idle state on the network, since those states can occur only when there is no noise. Therefore, one can always be certain of the lower bound of the window. Windows can be updated only conditionally after detection of a collision, because the collision may be due to noise. Therefore, one cannot be certain of the upper bound of the window.

During a collision, if noise does not occur, it is guaranteed that there are at least two contention parameters that are less than the upper bound of the window. However, if noise occurs during a collision, one can guarantee only that there are at least two contention parameters less than the upper bound of some previous window in which noise did not occur. Let a *true collision* be defined as a collision in which no noise occurs. One can guarantee that there are at least two contention parameters smaller than the smallest previous upper bound of the window in which a true collision occurred. Unfortunately, it is not possible to differentiate between true and false collisions, so one must keep track of the upper bounds of the windows from all previous collisions. Since the choice of the window is history dependent, it cannot be optimized by dynamic programming methods because the Principle of Optimality is not satisfied.

Assume that the probability of noise is independent for each contention slot and has a fixed value, p, that is known by all stations. Also, assume that the number of contending stations, n, is known. Finally, assume that each contention parameter, x_i, has a distribution, $F(x)$, and lies in the interval $(L,U]$. The following definitions are used to formulate the problem of choosing the upper bound as a recurrence.

$M(a,q[k])$:
The minimum expected number of contention slots to resolve contention, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

$g(a,w,q[k])$:
The probability of successful transmission in the interval $(a,w]$, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

$r(a,w,q[k])$:
The probability of no transmission in the interval $(a,w]$, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

$l(a,w,q[k])$:
The probability of collision in the interval $(a,w]$, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

In the array, $q[k]$, $q_1=U$, and q_i is the upper bound of the window during the $(i-1)^{th}$ collision. It follows from the above definitions that:

$$g(a,w,q[k]) + r(a,w,q[k]) + l(a,w,q[k]) = 1. \tag{3.6}$$

Let the elements of $q[k]$ be ordered from smallest to largest; q_1 is now $min(q_1 \cdots q_k)$ and $q_k = U$. Let a *true upper bound* be defined as the upper bound of the window in which a true collision occurred. Since p is independent for each contention slot, each q_i, except q_k, has a probability p of being a true upper bound, and $(1-p)$ of being false. The probability that q_1 is the smallest true upper bound is $(1-p)$. The probability that q_i is the smallest true upper bound is equal to the probability that all smaller q_i's are false and q_i is true. The probability that all q_i's are false is equal to p^{k-1}. Let $Pr(Q_i)$ be the probability that q_i is the smallest true upper bound.

$$Pr(Q_i) = \begin{cases} p^{i-1}\cdot(1-p) & 1 \le i < k \\ p^{k-1} & i = k \end{cases} \tag{3.7}$$

All Q_i's are mutually exclusive, so $\sum_{i=1}^{k} Pr(Q_i) = 1$. If q_i is a true upper bound, then there are at least two contention parameters in the interval $(a,q_i]$ and all x_i's are in the interval $(a,U]$. This condition is designated as event $(A \mid Q_i)$, with the following probability:

$$Pr(A \mid Q_i) = \frac{[1-F(a)]^n - [F(q_i)-F(a)]\cdot[1-F(q_i)]^{n-1} - [1-F(q_i)]^n}{[1-F(a)]^n} \tag{3.8}$$

If q_i is the smallest true upper bound, three mutually exclusive events can occur by choosing $a < w < q_i$, which can be identified as follows:

$B = \{$ exactly one x_i is in $(a,w]$, given $A \mid Q_i$ $\}$
$C = \{$ no x_i is in $(a,w]$, given $A \mid Q_i$ $\}$
$D = \{$ more than one x_i is in $(a,w]$, given $A \mid Q_i$ $\}$

Using Bayes' rule [19], the probabilities of $g(a,w,q[k])$ and $r(a,w,q[k])$ can be expressed for all q_i's as:

$$g(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot Pr(B \mid A \mid Q_i) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot\frac{Pr((A \mid Q_i) \cap B)}{Pr(A \mid Q_i)} \tag{3.9}$$

$$r(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot Pr(C \mid A \mid Q_i) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot\frac{Pr((A \mid Q_i) \cap C)}{Pr(A \mid Q_i)} \tag{3.10}$$

The set $((A \mid Q_i) \cap B)$ represents the event that exactly one x_i is in $(a,w]$, that at least one x_i is in $(w,q_i]$, and that all other x_i's are in $(w,U]$. Appropriate substitution in Equation (3.9) yields:

$$g(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i) \cdot \frac{[F(w)-F(a)]\cdot[[1-F(w)]^{n-1} - [1-F(q_i)]^{n-1}]}{Pr(A \mid Q_i)\cdot[1-F(a)]^n} \tag{3.11}$$

The set $((A \mid Q_i) \cap C)$ represents the event that at least two x_i's are in $(w,q_i]$, and all x_i's are in $(w,U]$. Appropriate substitution in Equation (3.10) yields:

$$r(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i) \cdot \frac{[1-F(w)]^n - [F(q_i) - F(w)]\cdot[1-F(q_i)]^{n-1} - [1-F(q_i)]^{n-1}}{Pr(A \mid Q_i)\cdot[1-F(a)]^n} \tag{3.12}$$

The heuristic recurrence relation can then be expressed as:

$$M(a,q[k]) = \min_{a<w<U} \left\{ 1 + g(a,w,q[k])\cdot 0 + l(a,w,q[k])\cdot M(a,q[k+1]) + r(a,w,q[k])\cdot M(w,q[k]) \right\} \tag{3.13}$$

The recurrence terminates upon detection of successful transmission. This is represented by the term $g(a,w,q[k])\cdot 0$ in Equation (3.13). However, since the number of collisions may be infinite, the number of states is unbounded, and Equation (3.13) cannot be solved. Consequently, only an approximation of the exact recurrence equation may be realized.

The following approximation attempts to reduce the number of states to a tractable finite value, by making certain assumptions. It assumes that the number of contending stations, n, and the probability of noise, p, are known at the start of contention. It also assumes p is independent for each contention slot. Furthermore, the approximation assumes that the upper bound of the next window chosen, w, is always less than the upper bound of the current window, and that after a collision, the upper bound of the window is reduced to w. Finally, although the conditional probabilities of success, idle and collision are dependent on all previous upper bounds of the windows, the only upper bounds used are the current upper bound and the absolute upper bound of the contention parameters. The probabilities in Equations (3.6-3.12) are then solved by using $k=2$, $q_1=b$ and $q_2=U$.

If one truncates the recurrence relation at $(b-a) < \delta$, it is possible that contention may not be resolved. Therefore, $M(a,q[k])$ must be redefined as the estimated number of contention slots to resolve contention either successfully or unsuccessfully, given that the current window is $(a,b]$ and that all contention parameters are in $(a,U]$. Since the only variable upper bound of the window is b, the approximate recurrence equation can be renamed in terms of a, w and b. Using the assumptions previously stated, the approximate recurrence equation is:

$$M(a,b) = \min_{a<w<b} \left\{ 1 + g(a,w,q[2])\cdot 0 + l(a,w,q[2])\cdot M(a,w) + r(a,w,q[2])\cdot M(w,b) \right\} \tag{3.14}$$

Since the recurrence relation always terminates when $(b-a) < \delta$, the boundary condition for $M(a,b)$ is:

$$M(a,b) = 1 \quad when\ \ b-a < \delta \tag{3.15}$$

As stated in equation (3.5), the total number of contention slots to resolve contention successfully is M/S. However, S is also dependent on all previous windows, and is as complex to model as M. It can also be approximated as a recurrence equation using the same assumptions that were used to generate $M(a,b)$ in Equation (3.14). Let $S(a,b)$ be the probability of successfully isolating the minimum, given the current window, $(a,b]$, upper bound of next window, w, and all contention parameters in the interval (a,U]. $S(a,b)$ is the probability of success in the current window plus the probability of collision times the probability of success in window $(a,w]$ plus the probability of no transmission times the probability of success in window $(w,b]$. $S(a,b)$ can then be formulated as:

$$S(a,b) = \left\{ g(a,w,q[2]) + l(a,w,q[2])\cdot S(a,w) + r(a,w,q[2])\cdot S(w,b) \right\} \tag{3.16}$$

The boundary condition for $S(a,b)$ when $b-a < \delta$ is the probability that there is exactly one contention parameter in the window $(a,b]$. This is equal to $g(a,b,q[2])$.

The minimum number of contention slots, $T(a,b)$, is then defined as:

$$T(a,b) = \min_{a<w<b} \frac{M(a,b)}{S(a,b)} \tag{3.17}$$

$$T(a,b) = \frac{1}{g(a,b,q[2])} \qquad \text{for } b-a < \delta \tag{3.18}$$

It must be guaranteed that the w chosen for each $M(a,b)$ is consistent with the w chosen for each $S(a,b)$. Since $M(a,b)$ and $S(a,b)$ are computed from smaller subintervals, the w's for these subintervals can be computed and stored in a lookup table to be shared by both $M(a,b)$ and $S(a,b)$. The algorithm for deriving $T(a,b)$ is simulated in the next chapter, and the associated program is given in Appendix B.

The approximate recurrence relation also requires additional memory overhead for implementation. While the original dynamic programming algorithm needs a separate table entry for each n, the recurrence relation requires a separate table entry for each combination of n and p. The number of entries for each (n,p) pair can be reduced by using a truncated binary decision tree as outlined by Juang and Wah [13]. The decision tree chooses one of two paths depending on whether a station detects collision or an idle line. No path is needed for the detection of a successful transmission, since such detection terminates the algorithm.

Using a four-level decision tree with 16-bit entries, and 16-bit random numbers for the contention parameters, each (n,p) pair would then require 0.03 Kbytes of memory. The standard Ethernet controller chip, MCS 8396, has 8 Kbytes of ROM available, so 266 (n,p) pairs may be stored. The next chapter discusses a possible implementation of the approximate recurrence relation in a real system.

3.3.2. Windows Becoming Out-of-Phase

This section discusses the case of the windows of the contending stations becoming asynchronized due to noise. As was discussed earlier, it is possible that not all stations may sense the same state of the network during a given contention slot. Figure 3.6 illustrates how this can affect the window-control protocol. During the first contention slot, all stations except station 1 transmit, and all stations reduce the upper bounds of their windows to w_2. Suppose in the second contention slot, station 2 detects noise while the others do not. It will then lower its upper bound to $w_3(2)$, while the other stations detect no transmission and raise the upper bounds of their windows to w_3.

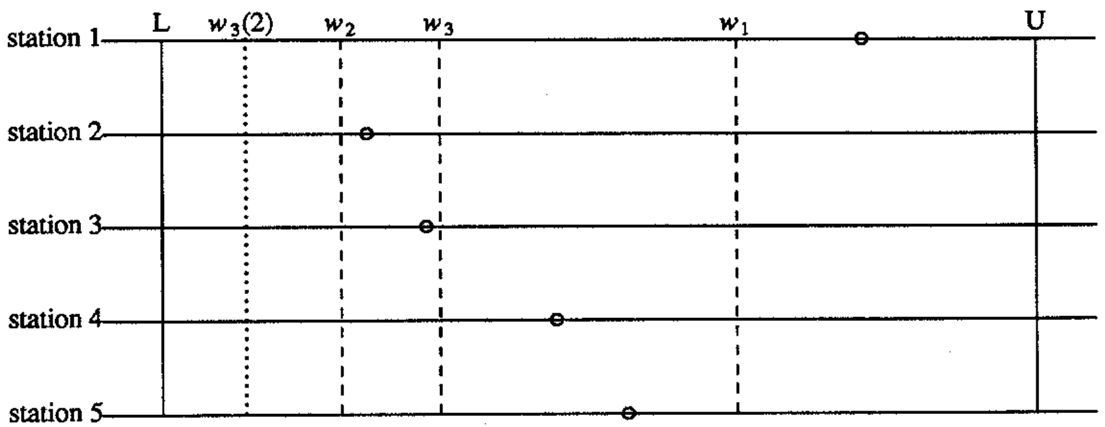

Figure 3.6. Wrong Minimum Identified by Window-Control Algorithm when Windows are Out-of-Phase

Now, during the third contention slot, only station 3 transmits and is successful. Station 3 is, therefore, falsely identified as the minimum.

To correct this problem, one logical solution is to add a broadcast phase after the minimum has been identified. After a station transmits successfully, it broadcasts its parameter, x, to all other stations in the next message. Following the first broadcast, all stations with contention parameters less than x attempt to broadcast their parameters in a second broadcast. If there is a collision, all stations restart the window-control algorithm with the upper bound of the contention parameters set to x. If some station broadcasts its parameter successfully during the second broadcast phase, it captures the bus and begins transmission of its message. Otherwise, if the line remains idle during the second broadcast phase, the original successful station captures the bus and transmits its message.

There is an additional problem in that all stations may not terminate the algorithm at the same time. This problem can be alleviated by the successful station continuing to transmit in the next σ contention slots before broadcasting its parameter, where σ must be long enough to guarantee that all stations have an opportunity to either detect that successful transmission is occurring or else timeout and terminate the algorithm. During the contention slot that a station succesfully transmits, it will be the only station with a contention parameter smaller than the upper bound of its window. Since the other stations' contention parameters are higher than the upper bounds of their windows, they will continue to sense the line but will not transmit. In the following contention slots, one of two events will occur to terminate iterations of new windows. The unsuccessful stations can either detect success and wait for the broadcast, or they can detect a series of false collisions and reduce the upper bounds of their windows until their window sizes become smaller than δ. The number of contention slots to reduce a window from its original size to δ, given a series of collisions, is $O\,((log_2(1/\delta))$, and is the lower bound on σ. The event of a station reducing its window to a size smaller than δ can be designated as the start of a timeout period. If a station senses a successful transmission within this timeout period, it listens for the broadcasted parameter. If a station senses an idle line within the timeout period, it restarts the algorithm. Otherwise, if a station continues to sense false collisions during the entire timeout period, it aborts its contention attempt.

If a station aborts contention due to a timeout, or detects success in the middle of the broadcast phase, it will not know the value of the broadcasted parameter. Also the value of the broadcast parameter might become corrupted, and therefore, unreadable due to noise. These problems can be remedied by all stations involved in such events forcing a collision in the second broadcast phase, and restarting their algorithms with an upper bound of U. In this case, the upper bounds of the windows may not be the same for all stations in the restarted algorithm, but this situation will be corrected during the next broadcast phase. Operation of the window-control algorithm with broadcast phase is illustrated in Figure 3.7.

The broadcast phase adds significant overhead to the time required to resolve contention. For applications where it is not necessary to identify the minimum contention parameter, the broadcast phase can be deleted. The successful station automatically captures the bus, and all other stations abort contention upon either detecting success or reaching a timeout. Even when all stations detect noise independently, the performance of the window-control algorithm is still better than Ethernet's exponential backoff algorithm. These results are shown in Chapter 4.

3.4. Dependent Noise

In reality, noise is often longer than one contention slot in length. Therefore, the probability that noise occurs during a given contention slot is dependent on whether noise occurred in previous contention slots. To model dependent noise, it is necessary to retain state information for the entire history of contention. Because of the complexity of modelling dependent noise, this section discusses only its general effects and does not attempt a detailed analysis.

In regards to the window-control algorithms, noise in successive contention slots will result in false collisions that continue to reduce the window. If the noise terminates before the window is reduced to a size less than δ, then the algorithms continue normally. If the noise continues after the window is reduced to a size less than δ, the stations enter a timeout period, as discussed in the last section. If the noise abates before the end of the timeout period, then the stations will restart the window-control algorithm; otherwise, the stations will abort the algorithm. The general effect of longer bursts of noise is that the probability of reducing the window to a size less than δ without being successful is greater. As a result, the algorithm is more likely to need to be restarted, and the total number of contention slots to resolve contention is increased. The behavior of the system under long bursts of noise should be similar to the behavior of the independent noise models for very high probabilities of noise.

On the other hand, short bursts of noise less than one contention slot in length will only affect the current contention slot. Therefore, such noise can be modelled as independent noise. If the burst of noise is too short, it might

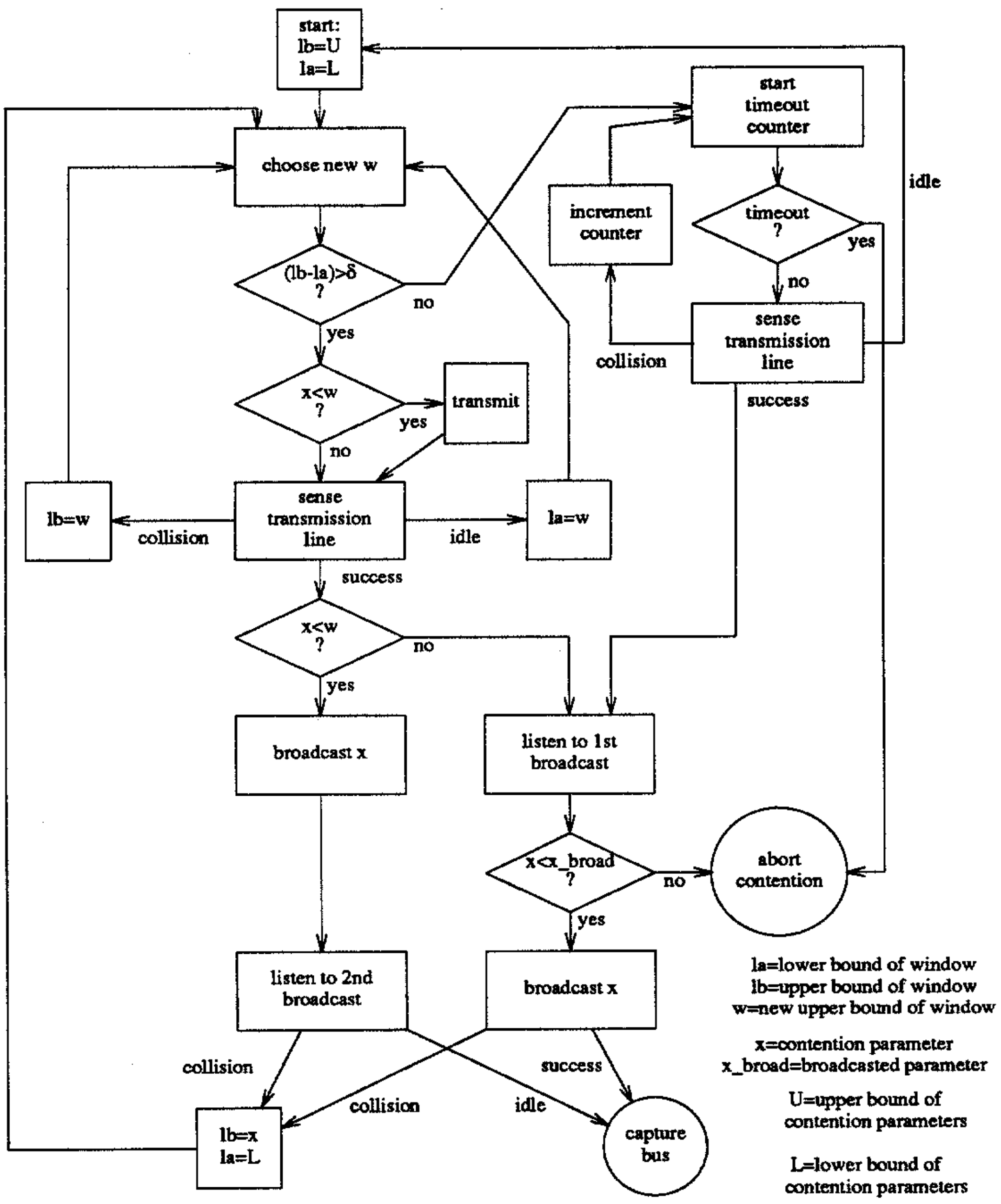

Figure 3.7. Window-Control Algorithm with Broadcast Phase

not be detected by all stations, but this problem was already discussed. Consequently, although the probability of noise is actually history dependent, the general behavior of the system can still be fairly accurately determined assuming a fixed, independent probability of noise for each contention slot.

3.5. Load Estimation

Noise also has an effect on load estimation. As stated in Chapter 2, the channel load is estimated according to a moving average of the windows used to successfully isolate the minimum contention parameter. However, noise causes false collisions such that the window used to isolate the minimum contention parameter may be lower than the window that is actually needed. Consequently, the added collisions due to noise may make the network appear as if the channel load was higher. The next chapter simulates the effects of noise on load estimation for the approximate greedy window-control algorithm.

3.6. Summary

This chapter has shown that noise can affect the detection of the states, collision, idle and successful transmission on the network. Noise may result in the false detection of a collision, but it will almost never result in the false detection of a successful transmission or idle line. Noise can also cause the window boundaries of the contending stations to become out of phase when its detection is localized to a particular station or set of stations.

This chapter outlined the effects of noise on the original window-control algorithm for the case where noise is detected by all stations. It also presented two protocols for countering these effects: the two-phase algorithm and the approximate recurrence relation given by Equation (3.18). In all protocols, it is possible for the window to be reduced to a size less than δ without isolating a station. In such cases, the algorithms must be restarted.

In the case where the windows may become out of phase, it is possible for the window-control algorithm to isolate a station that does not have the minimum contention parameter. Therefore, a broadcast phase must be added to correct for this phenomenon. The next chapter compares the performance of the various window-control algorithms through simulation, and discusses the implementation of a window-control algorithm in a real system in which noise is a factor.

4. SIMULATION AND ANALYSIS OF WINDOW-CONTROL PROTOCOLS

This section outlines simulation models for the binary exponential backoff algorithm and window-control algorithms and uses the simulations to gauge the performance of the models under various conditions. It then uses the results to present an implementation of a window-control algorithm in a real system in which noise is a factor. The first simulations assume that the channel load is exactly known, that noise is independent for each contention slot and that windows remain synchronized. They are used primarily to show the general effect of noise on the various algorithms. The next set of simulations shows what happens when the windows in the window-control algorithms become asynchronized. The last simulations show the results of load estimation on the approximate greedy window-control algorithm. All simulations were run on SUN 3/50 and SUN 3/260 workstations. The simulations involving the original dynamic programming lookup table and the binary exponential backoff algorithm were written in Fortran77, and all other simulations were written in C. The associated programs are included in the appendices.

All the simulations were performed with a 95% confidence interval C, which is ±5% of the mean, computed as follows [19]:

$$C = \pm\, 1.96 \times \sqrt{\frac{\sigma^2}{Z}}$$

In the equation above, Z is the total number of trials performed, and σ^2 is the variance of the sample. In the simulations, C was calculated arbitrarily after every 800 trials. If C was in the interval $\pm[\, 0.05 \times \textit{mean}\,]$, then the simulation was terminated, otherwise it was run again with another 800 trials. Consequently, Z was some multiple of 800.

All simulations considered noise probabilities ranging from 0 to 0.9. The asymptotic value for the probability of noise is 1.0. When the probability of noise is 1.0, the algorithms can never terminate, and the number of contention slots to resolve contention is infinite.

4.1. Independent Noise, Synchronized Windows

All the following models assume that for each contention slot, all stations sense the same state of the network, and noise is independently generated in each contention slot with a fixed probability. In the simulations, noise was modelled by generating a random number during each contention slot. If that random number was less than the threshold of noise defined for a given simulation, then the network was treated as if it were in a state of collision, regardless of whether or not there was an actual collision during that contention slot. The contention parameters were chosen as uniformly distributed random numbers in the interval [0,1). Upon restart of the algorithm, the contention parameters were regenerated to avoid infinite loops when the two smallest contention parameters differ by less than δ. In all the protocols, the number of stations contending in a given slot was counted and placed in h. If there was noise or $h > 1$, then there was a collision. If $h = 0$, then the line was idle. Otherwise, if $h = 1$, then the algorithm terminated successfully.

4.1.1. Binary Exponential Backoff Algorithm

Figure 4.1 illustrates the effects of noise on the binary exponential backoff algorithm for 10, 20, 30, 40 and 50 contending stations. Note how the performance degrades fairly evenly for probabilities of noise < 0.8, as the number of stations is increased. This corresponds to the linear degradation in performance of the binary exponential backoff

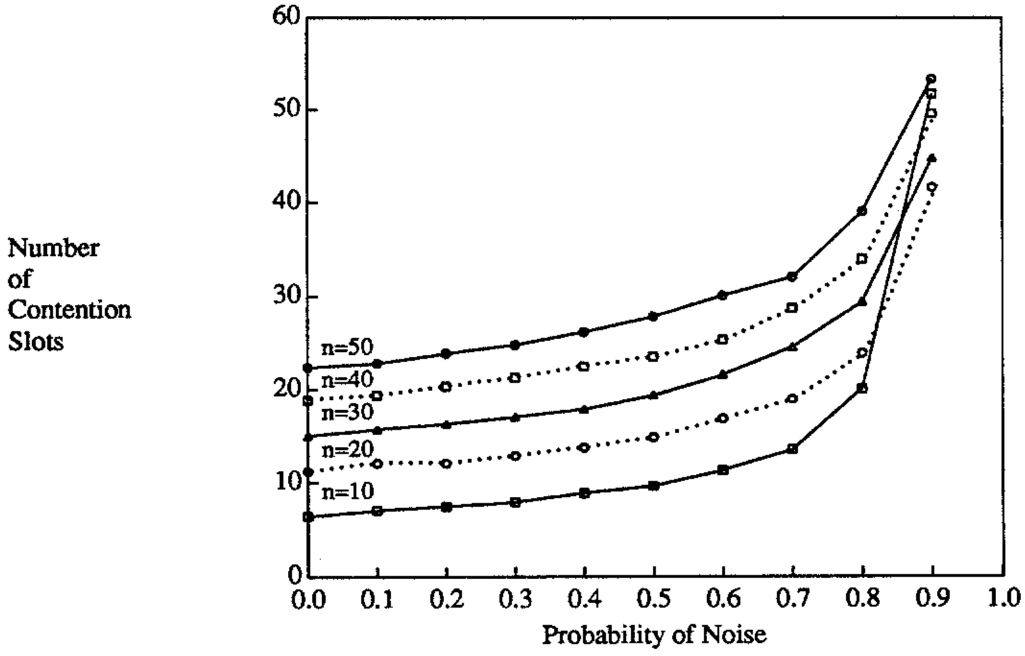

Figure 4.1. Binary Exponential Backoff Algorithm (n=10,20,30,40,50)

algorithm when noise is not present (Figure 2.2). Under the presence of most levels of noise, the binary exponential backoff algorithm still degrades linearly as the number of stations, n, increases.

There seems to be an anomaly for the number of contention slots at 90% noise and 10 contending stations, but this can be explained from the fact that the exponential backoff algorithm causes the mean waiting time between successive transmissions to become greater for each retransmission. As the number of contention slots becomes greater, there are more empty slots. Early in the algorithm, the probabilities of collision are much higher when n is large than when n is small. However, at some point in the algorithm, the probability of no transmission is greater when n is small than when n is large. Higher probabilities of noise tend to cause the number of contention slots to increase to the point where this is the case. Consequently, at very high probabilities of noise, the performance of the algorithm when n is small may be worse than when n is large. Another consequence of this result is that noise has a greater effect when n is small than when n is large. This can be seen from Figure 4.1 where the performance curve is much steeper for $n=10$ than for $n=50$.

4.1.2. Binary-Divide Window-Control Algorithm

The first window-control simulation shows the effects of noise on the binary-divide window-control algorithm. The simulation was written in C and simulated on a SUN 3/260. The algorithm was restarted if contention was not resolved when $\delta \leq 1/(10n)$.

Figure 4.2 shows the various performance curves for 10, 20, 30, 40 and 50 contending stations. Note that the performance degrades logarithmically as the probability of noise increases. The performance also degrades as the number of stations increases, but nonlinearly, unlike the binary exponential backoff algorithm.

4.1.3. Original Dynamic Programming Window-Control Algorithm

The dynamic window-control simulations were written in Fortran and simulated on a SUN 3/50, using the dynamic programming method with lookup table as outlined in Chapter 2. The dynamic programming algorithm was truncated when the window size, δ_1, became less then $1/(10n)$ as in Wah and Juang's model [13]. If the window size became smaller than δ_1, a binary-divide algorithm was used to complete the iterations. The binary-divide

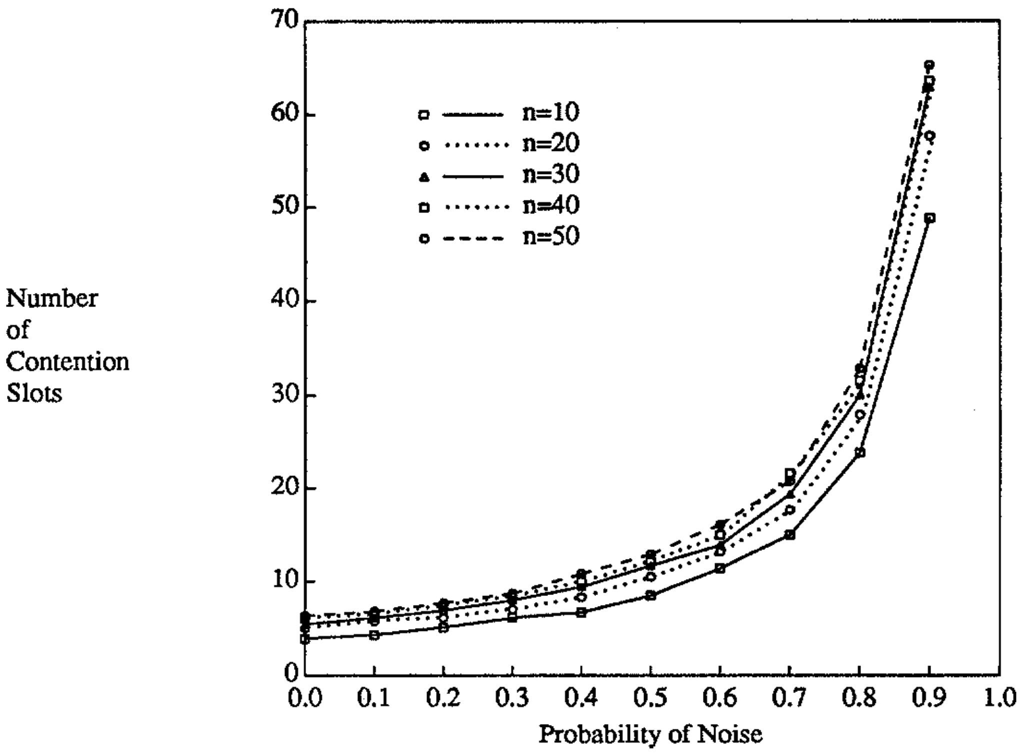

Figure 4.2. Binary-Divide Window-Control
(δ_2=1, n=10,20,30,40,50)

algorithm was terminated at δ_2, at which point the algorithm was restarted if a minimum had not been identified. The total resolution was therefore: $\delta_1 \cdot \delta_2 = \delta_2 / (10n)$.

Figure 4.3 shows the results of varying δ_2 for n=10 stations. (If δ_2=1, then the binary-divide phase is skipped altogether, and the algorithm is restarted when δ_1 is reached.) As δ_2 decreases, more iterations are wasted before the algorithm can be restarted after an error due to noise. For higher probabilities of noise, the probability of failure increases and the number of restarts increases. The net effect is that decreasing δ_2 causes an increase in the number of contention slots that becomes more pronounced as the probability of noise increases.

Figure 4.4 shows the various performance curves of n= 10, 20 and 30 when δ_2=1. Note that the number of contentions is relatively independent of n, unlike the binary exponential backoff and binary-divide window-control algorithms.

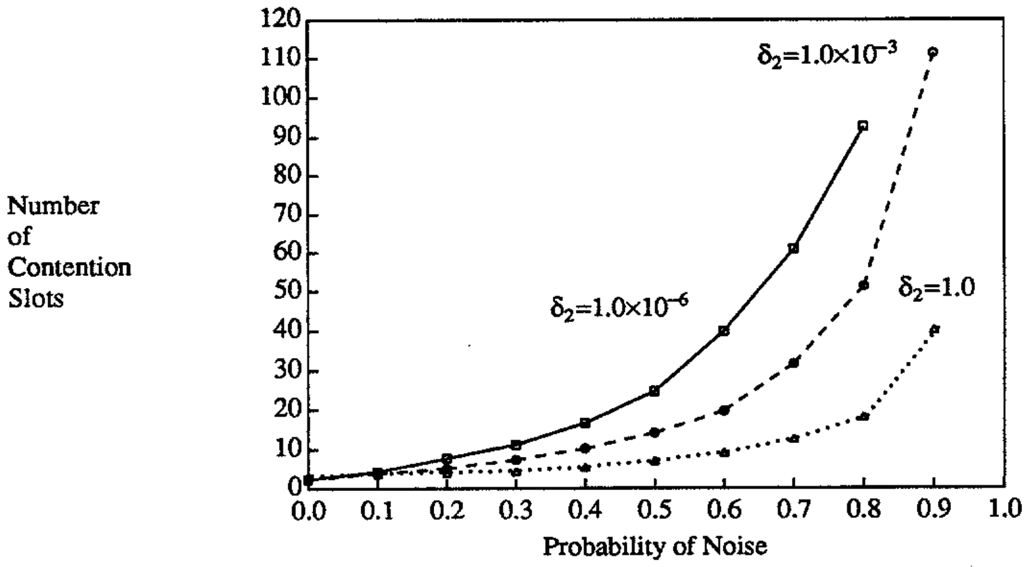

Figure 4.3. Original Dynamic Window Protocol
(n=10)

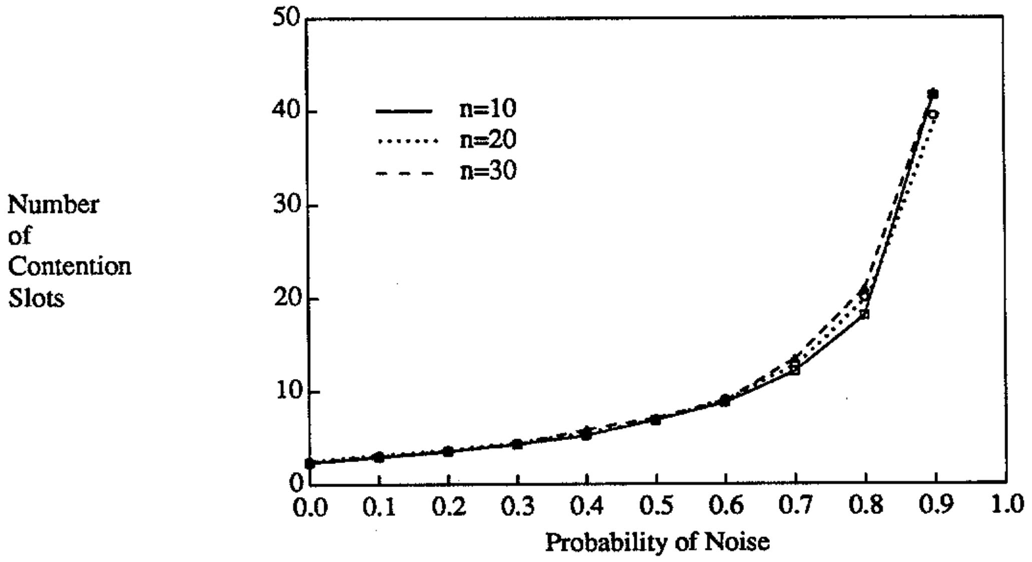

Figure 4.4. Original Dynamic Programming Algorithm
$\delta_1=(10n)^{-1}$, $\delta_2=1$

4.1.4. Two-Phase Window-Control Algorithm

The two-phase model is the same as the one outlined in Chapter 3. It used the same lookup table as the dynamic programming algorithm, was written in Fortran, and was simulated on a SUN 3/50.

Figure 4.5 illustrates the effects of varying δ_2 for the two-phase model when n=10. Note that for low probabilities of noise ($p \leq 0.3$), the performance is about the same as the original algorithm, but for high probabilities of noise the performance degrades much more rapidly than the original algorithm as δ_2 is increased. This is due to the fact that for low probabilities of noise, the two-phase algorithm has a higher probability of correcting itself when the contention parameters get outside the window. However, at higher probabilities of noise, the probability to correct this problem is much smaller, and it is more efficient to just restart the algorithm. The effects of noise on the two-phase protocol are much more extreme when δ_2 is small, as can be seen by the steeper performance curves.

Nevertheless, the performance of the two-phase algorithm is relatively independent of n, as illustrated in Figure 4.6. The algorithm is simulated for δ_2=1 and n = 10, 20 and 30, and the performance is very similar to that of the original algorithm.

4.1.5. Approximate Recurrence Relation

Chapter 3 developed an approximate recurrence relation (Equation 3.17) to minimize the number of contention slots to resolve contention with the probability of noise added as a parameter. This algorithm was written in C and simulated on a SUN 3/260 with floating point accelerator. No binary phase was used, implying δ_2=1 as in the other window-control algorithms.

The approximate recurrence relation estimates that the probability that b is a true upper bound is equal to the probability of noise in a contention slot, $1-p$. Let p_1 be the probability that b is a true upper bound. In Equation (3.7), $\Pr(Q_1) = p_1$ and $\Pr(Q_2) = (1-p_1)$. It turns out that $p_1 = 1 - p$ is not a good approximation, since b may be a false upper bound due to previous false collisions. A much better heuristic approximation can be made by assuming a smaller probability that b is true, such as $p_1 = (1-p)^2$.

Table 4.1 contains a comparison of the theoretical and simulated performance of the approximate recurrence equation for both values of p_1. The theoretical performance is the expected number of contention slots to resolve contention as calculated by the approximate recurrence equation. During calculation, the optimal window for each

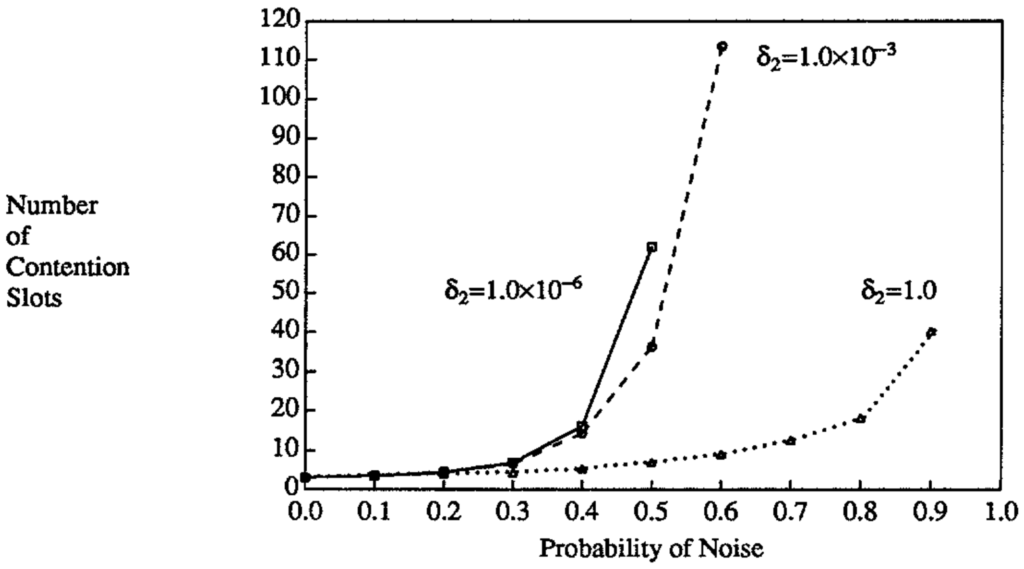

Figure 4.5. Two-Phase Algorithm
(n=10)

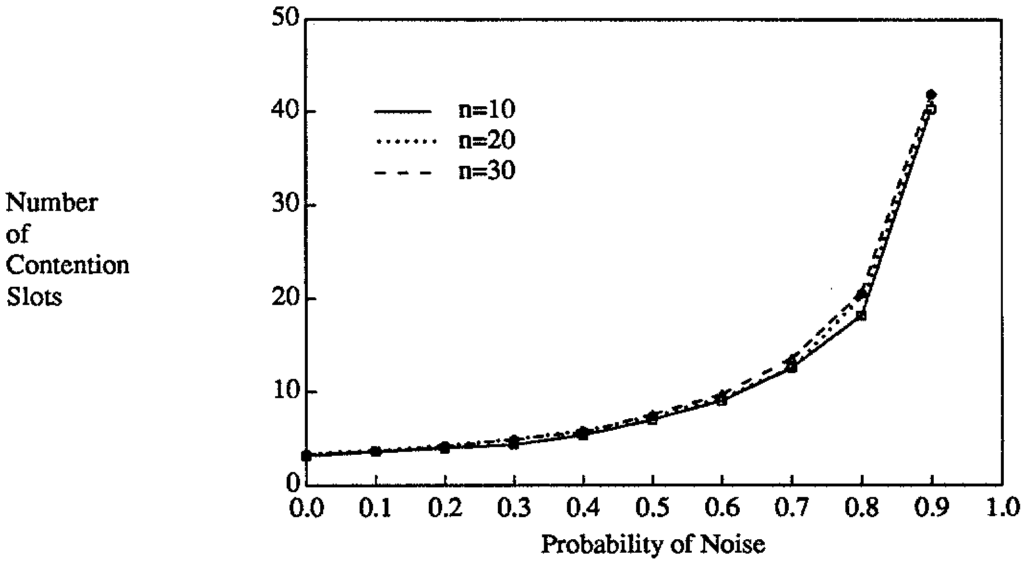

Figure 4.6. Two-Phase Algorithm
$\delta_1=(10n)^{-1}$, $\delta_2=1$

pair of lower and upper window boundaries was computed and stored in a table. The simulated performance is the actual performance of the network when using these windows. The correlation between the simulated and theoretical performance is dependent on the accuracy of the approximation. In Table 4.1, the simulated performance is much closer to the theoretical performance when p_1 is approximated by $(1-p)^2$. The overall performance is also improved by using $p_1 = (1-p)^2$.

It is possible to obtain even further improvements by developing an even more accurate approximation of p_1. Such an approximation should yield a better overall performance and simulated values that are even closer to the

Table 4.1. Comparison of the Performance of the Approximate Recurrence Equation for Various p_1's.

Dynamic Programming Algorithm (n=10)				
p = probability of noise during contention slot, p_1 = estimated probability that b is a true upper bound				
Probability of Noise	$p_1 = 1-p$		$p_1 = (1-p)^2$	
	Theory	Simulation	Theory	Simulation
0	2.5	2.4	2.5	2.4
0.1	2.6	2.9	2.7	2.8
0.2	2.8	3.4	2.9	3.2
0.3	3.0	4.0	3.3	4.0
0.4	3.3	4.9	3.9	4.8
0.5	3.7	6.0	4.8	6.0
0.6	4.5	8.0	6.2	7.3
0.7	5.8	11.4	8.6	9.6
0.8	8.4	16.7	13.5	15.1
0.9	16.5	36.5	28.5	28.5

Table 4.2. Comparison of the Performance of the Approximate Recurrence Equation for Various n's

Dynamic Programming Algorithm				
Probability of Noise	n=20		n=30	
	Theory	Simulation	Theory	Simulation
0	2.6	2.5	2.6	2.5
0.1	2.8	3.0	2.8	2.9
0.2	3.0	3.3	3.0	3.4
0.3	3.4	3.9	3.4	3.9
0.4	4.0	4.8	4.0	4.9
0.5	4.9	6.2	4.9	6.3
0.6	6.3	7.6	6.4	7.7
0.7	8.9	9.5	8.9	9.9
0.8	13.8	15.1	14.0	15.0
0.9	29.2	29.6	29.4	30.2

theoretical values. Such developments are left to future research. At any rate, both approximations perform better than the original dynamic programming algorithm, which does not account for noise.

The performance of the approximate recurrence relation is also relatively independent of the number of stations, similar to the other window-control algorithms. Table 4.2 compares the simulated expected number of contention slots of n=20,30 and $p_1 = (1-p)^2$, under various probabilities of noise; (n=10 is shown in Table 4.1).

There is one drawback to the approximate recurrence relation-- the time required to create the lookup table. Table 4.3 compares the time to create the lookup tables for the simulations. The long computation time prohibits simulation of higher values of n.

4.1.6. Comparison of Algorithms

Figure 4.7 contains a comparison of all the algorithms for n=20 and δ_2=1. For low levels of noise, all window-control algorithms perform better than the binary exponential backoff algorithm. For higher levels of noise, most window-control methods, except binary-divide, still perform better than the binary exponential backoff algorithm. All algorithms suffer from a logarithmic degradation in performance as the probability of noise increases.

Several interesting observations can be made by comparing the performance of the window-control algorithms. As expected, the pure binary-divide algorithm has the worst performance. However, the performances of the original dynamic programming algorithm and two-phase algorithm are almost identical when $\delta_2 = 1$. The approximate recurrence equation has the best performance for all levels of noise. The improvements gained are most pronounced at higher levels of noise. Therefore, it is worthwhile to include noise as a parameter in the formation of the recurrence equation.

Table 4.3. Time to Create Lookup Table for Simulation of Approximate Recurrence Equation

Time to Compute Lookup Table for Approximate Recurrence Equation (in minutes)			
n=10	n=20	n=30	n=40
9.9	81.0	280.4	456.8

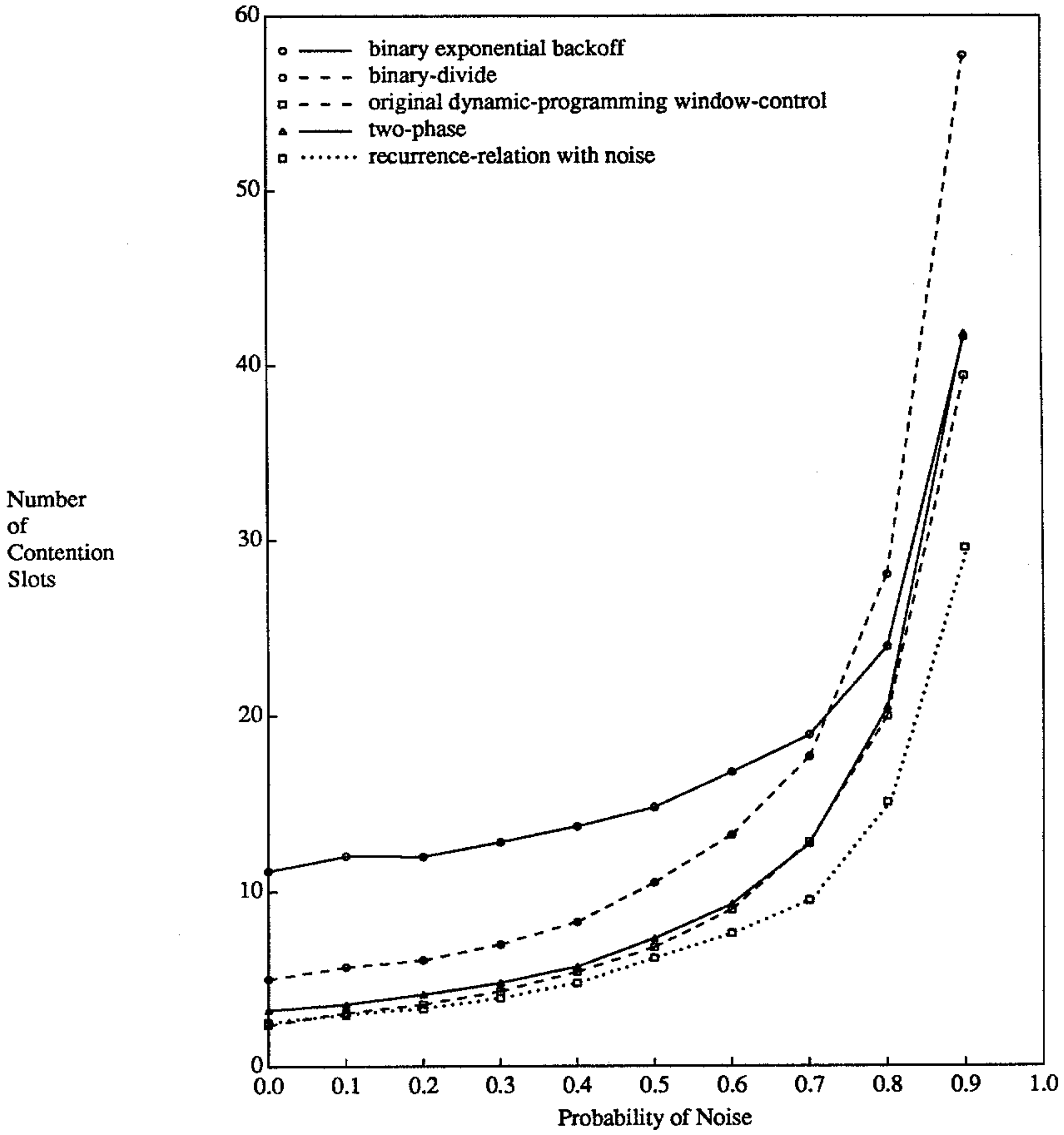

Figure 4.7. Comparison of Bus Contention Algorithms
(δ=1, n=20)

4.2. Asynchronized Windows

This section looks at the effects of noise when it causes individual stations to detect different states of the network. In the worst case, each station may have a noisy link to the network with a given probability of noise that is independent from that of other links. The simulation models the two-phase algorithm with $\delta_1 = 1/(10n)$ and $\delta_2=0.1$. It compares the case where all stations detect all noise for n=10, to the case where all stations have independent but equal probabilities of detecting noise for n=10, 20 and 30.

In the simulation, restarts were allowed only upon the detection of an idle line, as outlined in Section 3.3.2. To keep track of the true minimum, contention parameters were not regenerated during a restart. To guarantee that each contention parameter differed by at least $\delta = \delta_2 \cdot \delta_1 = 1/(100n)$, the contention parameters were modified by including each station's unique identification number. The identification number was an integer, I, in the interval [0,n). Contention parameters were formed by selecting a random integer in the interval [0,100), and by adding to it I/n. The contention parameter was scaled to the range [0,1) by dividing the result by 100. Figure 4.8 illustrates the effects of noise on isolating a station (not necessarily the minimum), when the windows remain synchronized and when the windows become asynchronized.

It is interesting that the performance for the asynchronized windows is better than the performance for synchronized windows. This becomes more evident at higher levels of noise, but should not be surprising due to the operation of the two-phase protocol. If the proper minimum is not isolated due to noise, some other contention parameter may lie within its designated window and just happen to be the only one transmitting. Some stations may reduce their windows faster than others and be able to restart the algorithm sooner or some stations may lag behind and still happen to be within the window bounds. Either of these cases opens up greater possibilities of isolating a station more quickly when noise is significant.

The window-control algorithm can be used just as a means for resolving contention without finding the minimum. In such cases, it has the same function as the binary exponential backoff algorithm. Since the performance of the window-control algorithm for isolating stations is improved when the windows become out of phase, the same arguments hold concerning the superiority of the window-control algorithms to the binary exponential backoff algorithm.

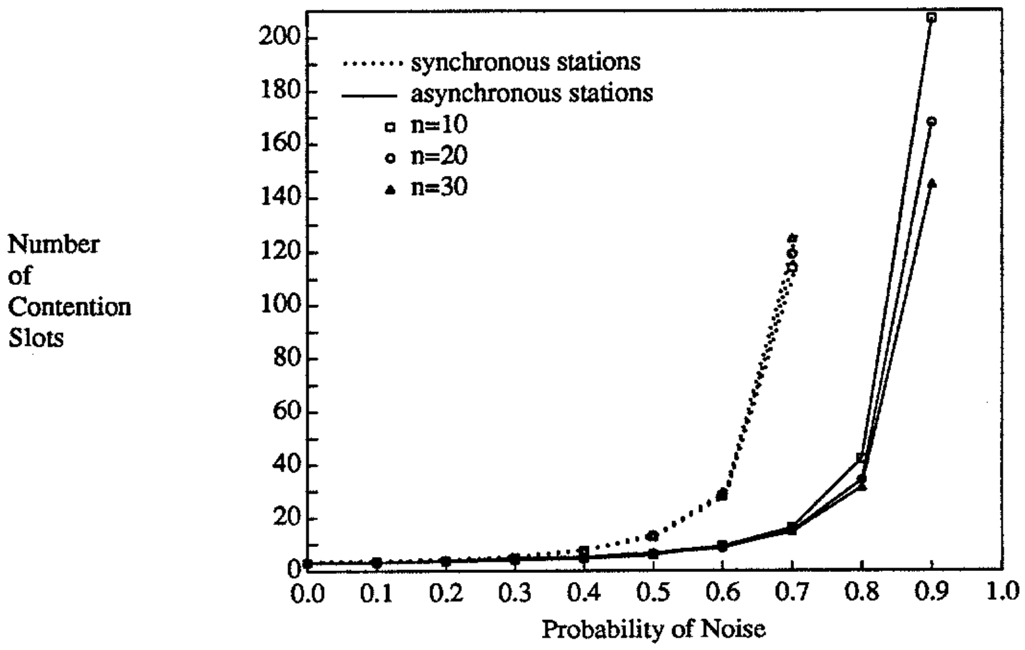

Figure 4.8: Two-Phase Algorithm with Independent Noise Detection $\delta_1=(10n)^{-1}$, $\delta_2=0.1$

Table 4.4 shows the percent of true minimums identified at the termination of the algorithm. Note that the percent of true minimums decreases as n is increased. This decrease is more pronounced at higher probabilities of noise. To find the true minimum, the algorithm would have to be restarted after the broadcast phase with modified initial window boundaries, as outlined in Section 3.3.2. As a result, the number of contention slots to resolve contention would actually be greater than the number shown in Figure 4.8. One would also have to add in the time to complete the broadcast phase for each restart. Therefore, the time required to isolate the minimum may increase dramatically if the windows become out of phase.

One should note that in reality, it is very unlikely that all stations on the network will sense the network differently. Instead, it is more likely that one or two stations or groups of stations may detect noise differently than the others. As a result, the percent of true minimums in Table 4.4 and the performance curves of the independent stations in Figure 4.8 are lower bounds.

Table 4.4. Percent of True Minimums

Independent Stations			
Probability	% true minimums		
of Noise	n=10	n=20	n=30
0	100	100	100
0.1	94.1	93.9	93.6
0.2	85.2	86.4	86.1
0.3	80.1	81.9	77.7
0.4	72.6	70.9	70.3
0.5	68.5	64.9	62.9
0.6	61.0	57.7	54.6
0.7	55.6	49.7	45.6
0.8	51.5	43.3	39.4
0.9	50.1	41.1	35.6

4.3. Load Estimation

This section simulates the effects of noise on load estimation. The number of contending stations was estimated by a moving average of previous windows using Equations (2.12) and (2.13). These load estimating equations were used in the approximate greedy window-control outlined in Chapter 2. The approximate greedy window-control algorithm was chosen because it was the simplest of the load dependent window-control algorithms to simulate. The simulation assumed that there were 100 stations in the system. Since contention only occurred when there were two or more contending stations, the estimated value of n was constrained to the interval, [2,100]. The simulation was written in C and performed on a SUN 3/260. As in previous simulations, n was the number of contending stations, and $\delta_1 = 1/(10n)$.

Figure 4.9 illustrates the effects of noise on the approximate greedy window-control algorithm with load estimation. In all cases, the performance when the channel load is estimated is worse than when the load is exactly known. The performance is much worse for smaller values of n, especially when the probability of noise is high. This problem is due to the fact that false collisions may cause the minimum contention parameter to be isolated by a window with an upper bound smaller than that which is actually needed. The net result is that the network appears to be more heavily loaded than it actually is. Consequently, noise has a greater effect on load estimation when n is small.

Table 4.4. Percent of True Minimums

Independent Stations			
Probability of Noise	% true minimums		
	n=10	n=20	n=30
0	100	100	100
0.1	94.1	93.9	93.6
0.2	85.2	86.4	86.1
0.3	80.1	81.9	77.7
0.4	72.6	70.9	70.3
0.5	68.5	64.9	62.9
0.6	61.0	57.7	54.6
0.7	55.6	49.7	45.6
0.8	51.5	43.3	39.4
0.9	50.1	41.1	35.6

4.3. Load Estimation

This section simulates the effects of noise on load estimation. The number of contending stations was estimated by a moving average of previous windows using Equations (2.12) and (2.13). These load estimating equations were used in the approximate greedy window-control outlined in Chapter 2. The approximate greedy window-control algorithm was chosen because it was the simplest of the load dependent window-control algorithms to simulate. The simulation assumed that there were 100 stations in the system. Since contention only occurred when there were two or more contending stations, the estimated value of n was constrained to the interval, [2,100]. The simulation was written in C and performed on a SUN 3/260. As in previous simulations, n was the number of contending stations, and $\delta_1 = 1/(10n)$.

Figure 4.9 illustrates the effects of noise on the approximate greedy window-control algorithm with load estimation. In all cases, the performance when the channel load is estimated is worse than when the load is exactly known. The performance is much worse for smaller values of n, especially when the probability of noise is high. This problem is due to the fact that false collisions may cause the minimum contention parameter to be isolated by a window with an upper bound smaller than that which is actually needed. The net result is that the network appears to be more heavily loaded than it actually is. Consequently, noise has a greater effect on load estimation when n is small.

Figure 4.11 illustrates the performance of various bus contention algorithms with load estimation for $n=20$. Modifications were made for the dynamic programming algorithm and approximate recurrence relation because of the large amount of time required to create the lookup tables. Consequently, tables were only created for n equal to multiples of five between five and fifty inclusive. The scaling factor in the load estimation equation was not used for the approximate recurrence relation, since better performance was obtained by omitting it. The scaling factor is not as critical to the performance of the approximate recurrence relation, since choice of the windows is dependent on noise, unlike the other window-control algorithms. As a result, the window used to isolate the minimum can be used to determine the load of the network more accurately in the approximate recurrence relation than in the other algorithms.

Note that the load dependent window-control algorithms have a worse performance than the performance obtained when the load is exactly known, which can be seen by comparing Figure 4.7 to Figure 4.11. Nevertheless, the performance of the window-control algorithms is still better than the exponential backoff algorithm for most levels of noise. Also, the performance of the approximate recurrence relation is better than the performance of the other bus contention algorithms in the presence of noise, even when the load is estimated.

4.4. Implementation of a Window-Control Algorithm on Ethernet in the Presence of Noise

Implementation of a window-control algorithm on a 10-Mbit/s Ethernet network is restricted by the stringent real-time requirement that each contention slot has a duration of less than 60 μs [16]. The only algorithm that can be computed quickly enough in real time to satisfy this time requirement is the binary-divide algorithm. The approximate greedy window-control algorithm may be used if a lookup table is employed to compute the square root [13]. To use any of the other window-control algorithms, the sequence of windows would have to be precomputed and stored in a lookup table.

Because of its superior performance for all probabilities of noise, the approximate recurrence relation ought to be used in implementations where noise is a significant factor. Chapter 3 discussed how the sequence of windows could be stored on an MCS 8396 using a four-level binary decision tree with each (n,p) pair requiring 0.03 Kbytes of memory. One practical arrangement is to let n range from 5 to 100 by 5's, and let p range from 0 to 0.9 by 0.1. The required memory would then be 6 Kbytes, which can fit on the MCS 8396. Interpolation could be used for

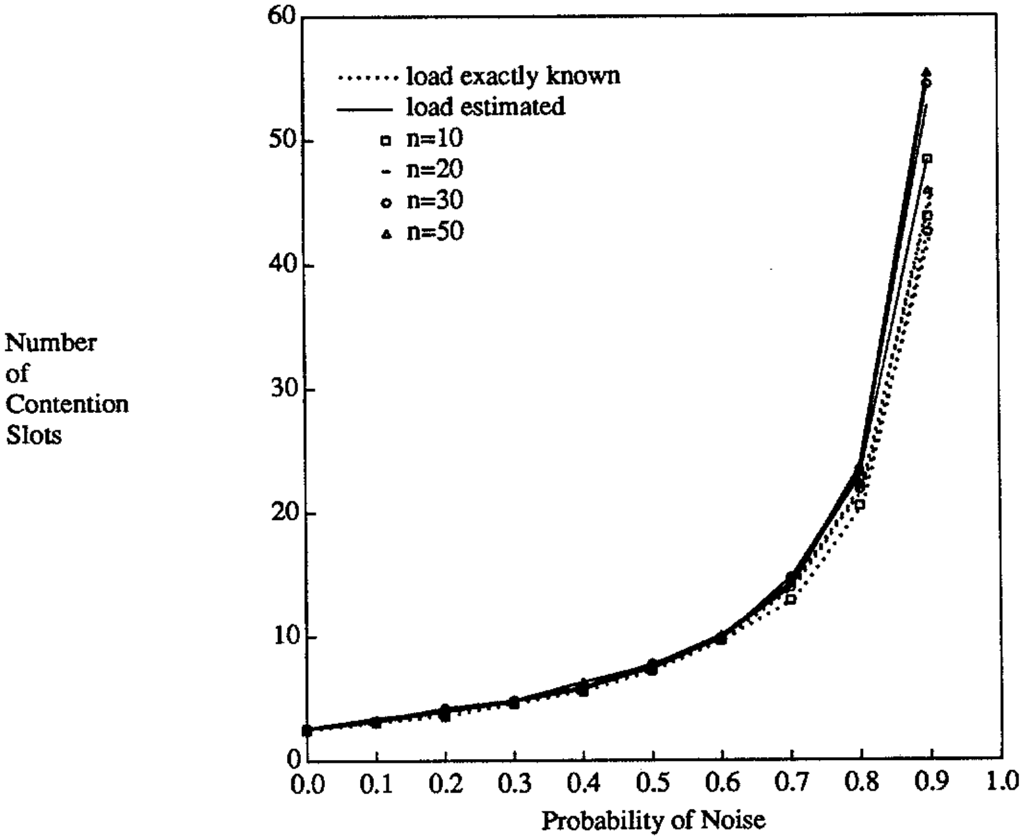

Figure 4.10. Approximate Greedy Window-Control with Improved Load Estimation (n=10,30,50, $\delta_1=(10n)^{-1}$, $\delta_2=1$)

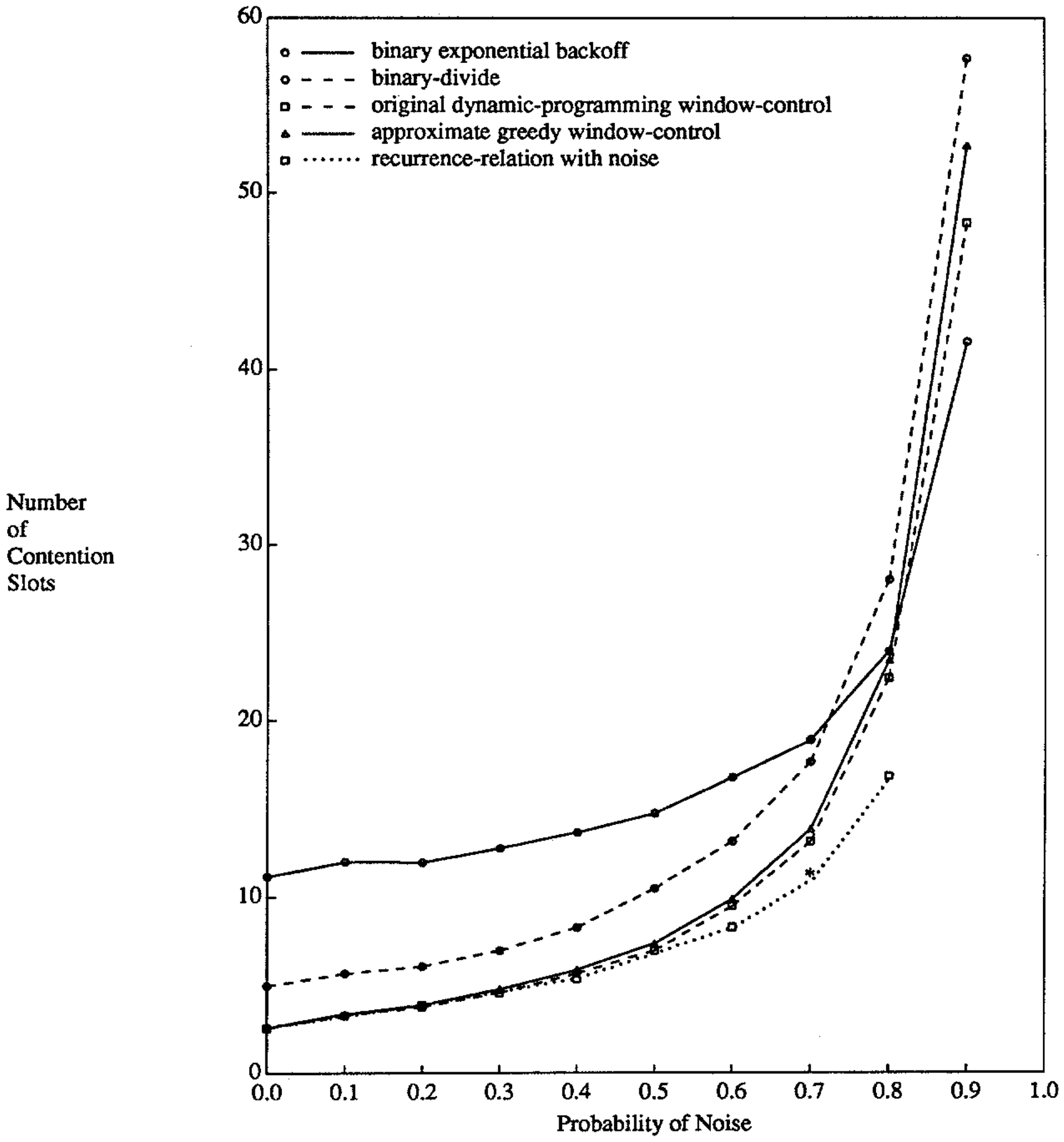

Figure 4.11. Comparison of Bus Contention Algorithms with Load Estimation (δ=1, n=20)

intermediate values of n and p. Nevertheless, the one drawback to this method is the enormous amount of time required to create the lookup tables.

In situations where the probability of noise is low, ($p \leq 0.3$), the original dynamic programming algorithm has a performance that is almost as good as the approximate recurrence relation. The advantage of the dynamic programming algorithm is that only one table is required for each n. Consequently, the intermediate values of n may also fit into the ROM of the MCS 8396. Furthermore, the computation time to create the tables in the original dynamic programming algorithm is about 1/4 the time required to create each table in the approximate recurrence relation with the same n. As a result, one might want to consider using the original dynamic programming algorithm if the probability of noise in the system is low most of the time.

In order to get the maximum benefit from the window-control algorithms, the channel load has to be estimated as accurately as possible, and the probability of noise has to be estimated for the approximate recurrence relation. The channel load can be estimated using the moving window technique with appropriate scaling to account for noise. The probability of noise can be estimated by the contending stations from the percentage of errors in transmitted messages. Coding theory techniques could be used to determine this value.

To guarantee that the window-control algorithm operates properly if the windows get out of phase, the modifications of Section 3.3.2 should be included. The major modifications are that a broadcast phase must be added to verify that the isolated station is the true minimum, and the algorithm should never be restarted unless the contending station detects that the line is idle. The broadcast phase can be omitted for use in bus contention where it is not necessary to isolate the minimum. Finally, the contending station should abort contention if it continues to sense collision on the network after it has reduced its window size to δ and the timeout period has expired.

5. CONCLUSIONS

The main result of this study is that by making a few adjustments to the window-control protocols, they can still be used to resolve bus contention when noise is present. One major adjustment is to include a termination condition, δ, for when the window size becomes too small, and to restart the algorithm with the original window when this occurs. Another adjustment is needed to account for the case where all stations do not detect the noise. In this case, the window boundaries may become out of phase, and the wrong station may be isolated. A broadcast phase must be added to verify that the proper station has been identified.

Another conclusion of this study is that the performance of most window-control algorithms in the presence of noise is superior to the performance of the binary exponential backoff algorithm. The gain in performance is greater at low probabilities of noise and high channel loads. Since normal operation of the network is likely to have low probabilities of noise, the window-control algorithms are preferable to the binary exponential backoff algorithm for resolving bus contention. Improvements to the dynamic programming algorithm outlined in Chapter 2 can be made by including noise as a parameter in a recurrence equation. This study developed an approximate recurrence relation that had superior performance to previously developed window-control algorithms in the presence of noise.

One drawback to the performance of the window-control algorithms is due to the effect of noise on load estimation. These effects are greater when the number of contending stations is small and the probability of noise is high, but can be corrected by adjusting the load estimation equation to account for noise.

Future research should investigate several of the issues discovered in this study. A good implementation of the broadcast phase should be realized with minimum added cost to resolve bus contention. Also, one should develop a better method for estimating the load of the system when noise is present. Additionally, one should investigate methods for estimating the probability of noise on the system for use in the approximate recurrence equation.

Previous discussion assumed that the stations were all synchronized initially and that they all began iterations of the window-control algorithms at the same time. In actuality, some stations may begin iterations before the others. Furthermore, their clocks might become skewed so that their contention slots overlap. Future work could be to investigate the problems of initializing the algorithms and synchronizing the clocks. In light of the study of asynchronized windows, it is very likely that the performance of the window-control algorithms will continue to be better than that of the exponential backoff method.

APPENDIX A. SIMULATION PROGRAMS IN FORTRAN

```
c ************************ Fortran Programs ****************************
c * binary exponential backoff, dynamic programming window-control,
c * two-phase, and two-phase with independent noise detection algorithms
c *********************************************************************
c
c Each program consists of a declaration of variables, Lookup Table Generation
c (except binary exponential backoff), Simulation, and Data Analysis.
c The Lookup Table Generation and Data Analysis are exactly the same for
c all programs, and are therefore, listed only once in detail.
c
c
c Note:
c Operation of all Fortran programs requires that the output from a
c random number generator be piped into the standard input. The random
c generator function is written in C and the executable file is called
c 'randgen'.
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c          Dynamic Programming Window-Control
c
c 1. Compute optimal window size of the window protocol with continuous i.i.d.
c distributions :
c   - using dynamic programming formulation
c   - evaluation of dynamic programming is truncated when interval
c     is less than 1.0 / (5 * no. of contending stations)
c   - binary-divide window search is used after truncation point
c     in dynamic programming algorithm
c   - algorithm is restarted if minimum has not been identified
c     in binary-divide window search and window size is less than delta2
c
c 2. Evaluate the dynamic programming window search scheme by simulations
c   - assume that the no. of contending stations is known
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c variables :
c  n,nr : no. of contending stations
c  dw : size of automic interval
c  cn(400,400) : matrix to store intermediate window size and expected
c           no. of contentions and the search tree
c  tx(i) : contending parameter of station i
c  tb(i) : contention parameter for binary phase of station i
c  wb(30) : working space for histogram
c  rr: random number read in from standard input file
c  vflag: specifies whether or not simulation is due to a new set of
c       parameters or is just a continuation due to the 95% confidence
c       interval not being satisfied
c  ccc: confidence interval calculated from the variance and number of trials
c  pfc: random number used to determine the probability of noise during a
c      contention slot
c  icount: total number of trials for a given simulation
c  fctst: threshold of noise
c  delta2: resolution of window size for binary phase
c  la,lb: lower,upper bounds of window in dynamic programming phase
c  a,b: lower,upper bounds of window in binary-divide phase
```

```
c   w,lw: next upper bound of window
c
c
      double precision rr
      integer n,la,lb,lw
      real dw,tx(40),cn(400,400),tb(40)
      real a,b,w
      real nr,jj,ccc,delta2
      integer wb(30)
      common ns,rr,dw
      logical vflag
c
c file (cc.data) : store the evaluation results
c
      open(unit=1,file='cc.data',status='new')
      write(6,1200)
1200  format('+++++   window control using dynamic programming   +++++',
     .   /,'+++++   continuous iid (discrete approximation)   +++++')
      write(6,1210)
1210  format('+++++  (dw = 1.0 / r * no. of contending stations) +++++')
c
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c Lookup Table Generation:
c iterate on no. of contending stations assuming there is no noise
c (This portion of the program was developed by J. Y. Juang [17] )
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
2000  do 7700 ns=10,30,10
c
c initialize parameters
c
      nr=float(ns)
      do 7700 iter=10,10
      n=ns*iter
      dw=1.0/float(n)
c
c
      do 3010 i=1,n
      do 3010 j=i,n
3010  cn(i,j)=0.0
c
c i : lower bound of last window
c i+k : upper bound of last interval
c j : candidate of optimal window
c
c iterate on lower and upper bound of last window
c
      do 4400 k=1,n-1
      do 4400 i=1,n-k
      wmin=9999.0
c exhaustive enumeration on all possible candidates of optimal window
      do 4300 j=i,i+k-1
      call prob(i,j,i+k,pg,pl,pr)
      tw=pl*cn(i,j)+pr*cn(j+1,i+k)
      if(tw .ge. wmin) goto 4300
      kwin=j
      wmin=tw
4300  continue
      cn(i,i+k)=1.0+wmin
      cn(i+k,i)=float(kwin)
c     write(1,4350) i,j,kwin,cn(i,i+k)
c4350  format('lower bound : ',i3,' upper bound : ',i3,' window : ',
c     . i3,' exp. contentions : ',f7.5)
4400  continue
c
c save evaluation results
c
c     write(6,5100) ns,iter,cn(1,n)
c5100  format(//,'+++++ no. of contending stations : ',i4,/,
c     .'+++++ resolution factor r (n = r * no. of station) :',i4,/,
c     .          '>>>>> ave. no. of contentions (analysis) : ',f9.3)
c     write(6,5150)
c5150  format(//,'<<<<< matrix of optimal windows >>>>>')
c     do 5200 i=1,n
c     write(6,5300) i
c     do 5200 j=1,n/10
c     write(6,5400) (cn(i,k), k=10*j-9,10*j)
c5200  continue
c5300  format(/,'===== row ',i3,' =====')
c5400  format(10(f9.5,1x))
c
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Simulation
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c format standard input file
133   format(d15.10)
c
c set window resolution for binary phase
      delta2=1
c
7900  do 7710 j=0,90,10
      jj=float(j)
      fctst=jj*0.01
      npkt=800
      vflag=.false.
8001  rewind 1
c
c initialize number of contention slots
c
8000  kstep=0
c
c generate contention parameters
c
8110  do 8100 i=1,ns
      read(5,133) rr
      tx(i)=rr
8100  continue
      la=1
      lb=n
8500  kstep=kstep+1
c
c generate noise
c
```

```
      read(5,133) rr
      pfc=rr
c
c contention resolutions
c
      if(lb .eq. la) goto 8700
c     if lb=la then go to the start of the binary phase, otherwise
c     retrieve the  discrete window value and convert it to integer
8590  w=cn(lb,la)
      lw=ifix(w)
c     convert to real window value
      w=dw*w
c
c collision detections
c
8601  ntrx=0
      do 8600 i=1,ns
      if(tx(i) .gt. w) goto 8600
      ntrx=ntrx+1
8600  continue
c
c collision?
      if (pfc .le. fctst .or. ntrx .gt. 1) then
        lb=lw
        goto 8500
c success?
      else
        if ( ntrx .eq. 1) goto 9000
      endif
c no transmission?
      la=lw+1
      goto 8500
c
c
c
c Binary Phase
c
8700  if (delta2 .eq. 1.0) then
        kstep=kstep-1
        goto 8110
              endif
c regenerate contention parameters for binary phase
      w=dw*float(lb)
      do 8720 i=1,ns
      read(5,133) rr
      if(tx(i) .gt. w) tb(i)=1.1
      if(tx(i) .le. w) tb(i)=rr
8720  continue
c
c contention resolution using binary-divide algorithm
8730  a=0.0
      b=1.0
8800  w=0.5*(a+b)
      z=w-a
c
c  restart evaluation if window has becomes too small.
c
      if (z .le. delta2) goto 8110
c
c collision detection
```

```
8621  ntrx=0
    do 8820 i=1,ns
    if(tb(i) .gt. w) goto 8820
    ntrx=ntrx+1
8820  continue
c
c collision ?
    if( pfc .le. fctst .or. ntrx .gt. 1) then
      b=w
      goto 8800
    endif
c
c success ?
    if (ntrx .eq. 1) goto 9000
c
c no transmission ?
    a=w
    goto 8800
c
c
c contention resolved. collect data
c
9000  write(1,9100) kstep
9100  format(i5)
    npkt=npkt-1
    if(npkt .gt. 0) goto 8000
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c Data Analysis
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c initialize sum, variance and histogram to 0 for new simulations
c (e. g. not continuations due to failure to satisfy requirements
c  for the 95% confidence interval)
c
    if (.not. vflag) then
      s=0.
      v=0.
      do 99 i=1,30
99      wb(i)=0
      icount=0
      endif
    rewind 1
c calculate mean and store number of contention slots in appropriate wb(i)
100  read(1,110,end=199)ncp
110  format(i5)
    s=s+float(ncp)
    icount=icount+1
      incp=ncp
c histogram has bins for number of contention slots greater than 24
      if (ncp .gt. 25 .and. ncp .le. 50) incp=25
      if (ncp .gt. 50 .and. ncp .le. 100) incp=26
      if (ncp .gt. 100 .and. ncp .le. 200) incp=27
      if (ncp .gt. 200 .and. ncp .le. 1000) incp=28
      if (ncp .gt. 1000 .and. ncp .le. 9999) incp=29
      if (ncp .gt. 9999) incp=30
    wb(incp)=wb(incp)+1
    goto 100
```

```
199  xmean=s/icount
c calculate variance
200  rewind 1
     vload=0
210  read(1,110,end=300)ncp
     v=v+(float(ncp)-xmean)**2
     goto 210
300  varian=v/icount
c calculate 95% confidence interval and run another 800 trials if necessary
     conf=1.96*sqrt(varian/icount)
     rewind 1
c
c distribution of no. of contentions
c
     write(6,500) icount
500  format('***** no. of packets transmitted : ',i5)
     ccc=0.05*xmean
     if (conf .gt. ccc) then
       npkt=800
       vflag=.true.
       rewind 1
       goto 8001
       endif
c print results
     write(6,400) xmean,varian,conf
400  format(/,'<<<<< ave. no. of contentions (simulation) : '
    . ,f9.4,/
    .       '<<<<< variance of contention period : ',f9.2,
    .'confidence=',f9.4)
     write(6,502)
502  format(//,'-- ncp --- frequency --- probability --- cum.--')
     cum=0.
     do 550 i=1,30
     k=i
     if (k .eq. 25) k=50
     if (k .eq. 26) k=100
     if (k .eq. 27) k=200
     if (k .eq. 28) k=1000
     if (k .eq. 29) k=4999
     if (k .eq. 30) k=5000
     pr=float(wb(i))/icount
     cum=cum+pr
     if (k .lt. 25) then
      write(6,560)k,wb(i),pr,cum
     else
      write(6,561)k,wb(i),pr,cum
     endif
550  continue
560  format(2x,i4,7x,i5,7x,f6.4,7x,f6.4)
561  format(x,'<',i4,7x,i5,7x,f6.4,7x,f6.4)
c
     write(6,570) j1
     write(6,580) j
570  format (//,'noise level= ',i5,'%')
580  format ('continued noise= 'i5,'%')
     tm=etime(at,bt)
     write(6,601) tm,at,bt
601  format ('elapsed time=',3(f10.2,3x))
7710 continue
7700 continue
```

```
      stop
      end
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c subroutine for computing probability of collision, success, and idle
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
      subroutine prob(n1,nw,n2,pg,pl,pr)
c
      real nr
      common ns,nr,dw
c
      pg=1.0
      pl=0.0
      pr=0.0
      if(n1 .ge. n2-1) return
c
      a=dw*float(n1-1)
      b=dw*float(n2)
      w=dw*float(nw)
c
      x=nr*(w-a)*((1.0-w)**(ns-1) - (1.0-b)**(ns-1))
      y=(1.0-a)**ns - (1.0-b)**ns - nr*(b-a)*(1.0-b)**(ns-1)
      z=(1.0-w)**ns - (1.0-b)**ns - nr*(b-w)*(1.0-b)**(ns-1)
      if (y .le. 0.0) return
      pg=x/y
      pr=z/y
      pl=1.0-pg-pr
      return
      end
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  end of Dynamic Programming Algorithm
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
c
c
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                 Two-Phase Algorithm
c
c The two-phase algorithm is exactly the same as the original dynamic
c programming algorithm except for the method by which windows are
c updated after an idle contention slot is detected.
c After an idle contention slot, the lower bound of the window, a, is
c raised to the value, w; the upper bound of the window, b, is raised
c to the value of the upper bound from the previous contention slot.
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c  New variables (variables from dynamic programming algorithm are retained):
c  kdeep= level of iterations within dynamic programming part of algorithm
c  bdeep= level of iterations within binary divide algorithm
c  kw(),bw()= arrays that store previous upper bounds for backtracking
c          within the dynamic prog. alg. and the binary alg.
c
      real kw(10),bw(20)
      integer kdeep,bdeep
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include  Lookup Table Generation here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Simulation
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c generate contending parameters
c
      delta2=1.0
133   format(d15.10)
7900  do 7710 j=70,90,20
      vflag=.false.
      jj=float(j)
      fctst=jj*0.01
      npkt=800
8001  rewind 1
c
c contention resolutions
c
8000  kstep=0
8555  la=1
      lb=n
      kdeep=1
      kw(kdeep)=n
      w=cn(lb,la)
c
c generate contention parameters
c
8500  do 8100 i=1,ns
      read(5,133) rr
      tx(i)=rr
8100  continue
c
c generate noise
c
      read(5,133) rr
      pfc=rr
c
      kstep=kstep+1
      if(lb .eq. la) goto 8700
c     if lb=la, then go to the start of the binary phase
8601  ntrx=0
      lw=ifix(w)
c   convert to real window value
      w=dw*w
c
c collision detections
c
      do 8600 i=1,ns
      if(tx(i) .gt. w) goto 8600
      ntrx=ntrx+1
8600  continue
c
c collision?
      if (pfc .le. fctst .or. ntrx .gt. 1) then
        if (kdeep .eq. 30) goto 8500
      kdeep=kdeep+1
      kw(kdeep)=lw
      lb=lw
      w=cn(lb,la)
      goto 8500
c success?
    else
      if ( ntrx .eq. 1) then
      goto 9000
      endif
    endif
c no transmission?
c
c
    la=lw+1
    lb=kw(kdeep)
    w=lb
    if (kdeep .gt. 1) kdeep=kdeep-1
    if (lb .lt. la) lb=la
    goto 8500
c
c  Start of Binary Phase
c
8700  if (delta2 .eq. 1) then
           kstep=kstep-1
           goto 8555
           endif
c
c regenerating contending parameters
c
    w=dw*float(lb)
    bdeep=1
    bw(bdeep)=1.0
    do 8720 i=1,ns
    read(5,133) rr
    if(tx(i) .gt. w) tb(i)=1.1
    if(tx(i) .le. w) tb(i)=rr
 8720  continue
c
c contention resolution using binary-divide algoirthm
 8730  a=0.0
     b=1.0
     w=0.5*(a+b)
 8800  z=w-a
c
c  restart evaluation if window has become too small.
c
     if (z .le. delta2) then
      lb=n
      la=1
      kdeep=1
      kw(kdeep)=n
      w=cn(la,lb)
      w=dw*w
      goto 8555
      endif
c
c collision detection
 8621  ntrx=0
     do 8820 i=1,ns
    if(tb(i) .gt. w) goto 8820
    ntrx=ntrx+1
8820  continue
c
c collision ?
    if( pfc .le. fctst .or. ntrx .gt. 1) then
      b=w
      bdeep=bdeep+1
      bw(bdeep)=b
      w=0.5*(a+b)
      goto 8800
    endif
c
c success ?
    if (ntrx .eq. 1) then
      goto 9000
      endif
c
c no transmission ?
c
    if (bdeep .eq. 0) then
      la=la+1
      if (kdeep .gt. 1) kdeep=kdeep-1
      lb=kw(kdeep)
      if (lb .lt. la) lb=la
      goto 8500
    else
      a=w
      b=bw(bdeep)
      w=b
      bdeep=bdeep-1
    endif
    goto 8800
c
c contention resolved. collect data
c
9000  write(1,9100) kstep
9100  format(i5)
    npkt=npkt-1
    if(npkt .gt. 0) goto 8000
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include Data Analysis here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include subroutine for computing
c     probability of collision, success, and idle here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c end of Two-Phase Algorithm
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
c
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c          Two-Phase Algorithm with Independent Noise Detection
```

```
c
c  This program uses the same protocol as the two-phase algorithm
c  except that the noise is detected independently for each station.
c  Consequently, arrays were set up to keep track of all the necessary
c  information (such as window boundaries) for each station.
c  The resolution of the dynamic programming lookup table is 1/10n,
c  and the resolution of the binary window size in the binary-divide
c  phase is 1/10.  This leads to a net resolution of 1/100n.
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c  All variables are the same as those used in the two-phase algorithm,
c  except the following variables were converted to arrays:
c     la,w,lb: la(40),w(40),lb(40)
c     a,b: la2(40),lb2(40)
c     kdeep,bdeep: kkdeep(40),bbdeep(40)
c     kw(10),bw(20): kw(40,10),bw(40,20)
c     pfc: pfc(40)
c
c  The following new variables were added:
c     mintx: station with minimum contention parameter
c     newtx: station isolated by two-phase algorithm
c     truecnt: number of true minimums at end of simulation
c     wait(40): used to keep track of which stations are in wait state
c             Wait state is entered when the window size becomes less
c             than 1/100n.
c     binary(40): used to keep track of which stations are in binary-divide
c             phase
c     done: flag to denote that some station has transmitted successfully
c
      integer mintx,newtx,truecnt
      logical binary(40),wait(40),done
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include Lookup Table Generation here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Simulation
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c generate contending parameters
c
133   format(d15.10)
7900  do 7710 j=0,90,10
      vflag=.false.
      jj=float(j)
      fctst=jj*0.01
      truecnt=0
      npkt=800
8001  rewind 1
c generate contention parameters for first set of stations
8000  mintx=1
      do 8100 i=1,ns
      read(5,133) rr
      n1=int(100*rr)
      n2=(i-1.0)/(nr+0.0)
      tx(i)=(n1+n2)/100
c set mintx to minimum of contending parameters
      if (tx(i) .lt. tx(mintx)) mintx=i
```

```
8100  continue
c initialize contention for first set of stations
     kstep=0
6000  do 6003 i=1,ns
     kw(i,1)=n
     la(i)=1
     lb(i)=cn(n,1)
     kkdeep(i)=2
     kw(i,kkdeep(i))=lb(i)
     binary(i)=.false.
     wait(i)=.false.
     done=.false.
     la2(i)=0
     lb2(i)=1
6003  continue
c
c
6010  ntrx=0
c  generate noise + transmissions
c    include line 6011 here instead of where it is now for synchronous
c              noise detection
      do 6013 i=1,ns
6011  read(5,133) rr
      w(i)=dw*lb(i)
      if (.not. wait(i)) then
        if (binary(i)) then
          if ( tb(i) .le. lb2(i)) ntrx=ntrx+1
        else
          if ( tx(i) .le. w(i) ) ntrx=ntrx+1
        endif
      endif
      pfc(i)=rr
6013  continue
c
c
c analyze state of system for each station
c and adjust window accordingly
c
      do 6023 i=1,ns
      if (wait(i)) goto 6200
c                                goto start of wait state
      if ( lb(i) .le. la(i) ) goto 6100
c                 goto start of binary phase
c collision?
      if (pfc(i) .le. fctst .or. ntrx .gt. 1) then
        llb=lb(i)
        lla=la(i)
        w(i)=cn(llb,lla)
        lw=ifix(w(i))
        w(i)=dw*w(i)
        kkdeep(i)=kkdeep(i)+1
        kw(i,kkdeep(i))=lw
        lb(i)=lw
6083    goto 6023
        endif
c success?
      if (ntrx .eq. 1) then
        if ( tx(i) .le. w(i) ) then
            done=.true.
            newtx=i
          else
            tx(i)=1.1
          endif
          goto 6023
        endif
  c no transmission?
          la(i)=lb(i)+1
          if (kkdeep(i) .gt. 1) kkdeep(i)=kkdeep(i)-1
          lb(i)=kw(i,kkdeep(i))
        goto 6023
  c
  c start of binary phase
  c
  6100  if (.not. binary (i)) then
  c           initialize if necessary
         binary(i)=.true.
         bbdeep(i)=1
         lb2(i)=w(i)
         la2(i)=w(i)-dw
         bw(i,bbdeep(i))=lb2(i)
         if ( tx(i) .gt. w(i)) then
           tb(i)=1.1
         else
           tb(i)=tx(i)
         endif
       endif
  c collision?
       if (pfc(i) .le. fctst .or. ntrx .gt. 1) then
         lb2(i)=0.5*(la2(i)+lb2(i))
         bbdeep(i)=bbdeep(i)+1
         bw(i,bbdeep(i))=lb2(i)
  6183    goto 6500
         endif
  c success?
       if (ntrx .eq. 1) then
         if ( tb(i) .le. lb2(i) ) then
             done=.true.
             newtx=i
         else
             tx(i)=1.1
         endif
         goto 6500
       endif
  c no transmission?
         if (bbdeep(i) .eq. 1) then
           binary(i)=.false.
           la(i)=la(i)+1
           if (kkdeep(i) .gt. 1) kkdeep(i)=kkdeep(i)-1
           lb(i)=kw(i,kkdeep(i))
         else
           bbdeep(i)=bbdeep(i)-1
           la2(i)=lb2(i)
           lb2(i)=bw(i,bbdeep(i))
         endif
  6500  if (( lb2(i)-la2(i)) .lt. 1.0/(200*ns)) then
  c enter wait state if window is too small
         lb(i)=cn(n,1)
         la(i)=1
         binary(i)=.false.
              kkdeep(i)=2
```

```
      kw(i,kkdeep(i))=lb(i)
      wait(i)=.true.
      endif
    goto 6023
c
c start of wait state
c
6200  if (pfc(i) .gt. fctst .and. ntrx .eq. 1) then
            tx(i)=1.1
    else
            if (pfc(i) .gt. fctst .and. ntrx .eq. 0) wait(i)=.false.
    endif
c
6023  continue
    kstep=kstep+1
    if ( .not. done .and. kstep .le. 10000) goto 6010
    if (mintx .eq. newtx) truecnt=truecnt+1
9000  write(1,9100) kstep
9100  format(i5)
    npkt=npkt-1
    if (npkt .gt. 0) goto 8000
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include Data Analysis here (plus the following lines)
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
    write(6,580) truecnt
580   format ('number of true minimums= ',i5)
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include subroutine for computing
c    probability of collision, success, and idle here
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  end of Two-Phase Algorithm with Independent Noise Detection
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
c
c
c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c               Binary Exponential Backoff Algorithm
c
c  This program simulates the exponential backoff algorithm of an ethernet
c  network in the presence of noise.
c  ns=#of stations
c  slot(i)=next slot that station i will attempt to transmit
c  doub(i)=number of transmission attempts station i has made
c        (new wait time= (rand#)* 2**doub(i) if i<=10
c                    or 1024 (2**10)    if i>10 )
c        (station i drops out of contention if i=16)
c  kstep=number of contention slots to resolve conflict
c  kcount=number of stations transmitting in a given contention slot
c  npkt=number of runs per simulation
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
     double precision rr
```

```
      integer ns,npkt
      integer slot(60),doub(60)
      integer wb(30),wj,wj1
      logical vflag
c
c
c initialization simulation
c
133   format(d15.10)
      write(6,5110)
5110  format('Ethernet Exp Backoff Model')
      do 7720 ns=10,10
      write(6,5100) ns
5100  format(///,'number of contending stations:',i5)
      do 7700 j=0,90,10
      vflag=.false.
      fctst=float(j)/100.0
      npkt=800
8001  rewind 1
2000  kstep=0
c initialize noise for the first slot and the next slot that
c a station may transmit if it detects a collision
      read(5,133) rr
      do 1001 i=1,ns
        doub(i)=1
        read(5,133) rr
              slot(i)=2.0*rr+1
1001  continue
c
c count the number of transmissions in the current slot
c
1000  kcount=0
      kstep=kstep+1
      do 2001 i=1,ns
              if (slot(i) .eq. kstep) then
                kcount=kcount+1
          doub(i)=doub(i)+1
          if (doub(i) .gt. 16) then
             slot(i)=0
             goto 2500
             endif
c
c determine next slot that station will transmit if collision is detected
c
          read(5,133) rr
          if (doub(i) .le. 10) then
            slot(i)= 2**doub(i)*rr +1 +slot(i)
           else
            slot(i)= 1024*rr +1 +slot(i)
          endif
c          write (6,999) kstep,i,slot(i)
c999         format(i5,4x,i5,4x,i5)
        endif
2001  continue
2500  if (kstep .gt. 5000) goto 9000
c
c add noise
c
      read(5,133) rr
      pfc=rr
```

```
c If there is a collision (kcount>1 or noise) or no tranmission (kcount=0)
c then repeat above procedure for the next contention slot.
c If there is a successful transmission, terminate contention.
    if (pfc .le. fctst) kcount=2
    if (kcount .ne. 1) goto 1000
c
c    write(6,998) kstep
c998  format(//,'kstep=',i5)
c
9000  write(1,9100) kstep
9100  format(i5)
5000  npkt=npkt-1
    if (npkt .gt. 0) goto 2000
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include Data Analysis here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  end of Binary Exponential Backoff Algorithm
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

```
/*  This program generates a continuous stream of random numbers
 *  and is written in C. The corresponding executable file is
 *  called 'randgen'.  The output from randgen must be piped
 *  into the standard input of all Fortran programs in this Appendix.
 *  (The random number had to be written in C, because the standard
 *   random number generator in the Fortran library cycled after a
 *   certain point.) */

#include <stdio.h>

main()
{
  int i;
  double z;
  srand(1);
  for (;;)
    {
     z= rand()/2147483648.0;
     printf("%5.10f0,z);
    }
}  /* end of random number generator */
```

APPENDIX B. SIMULATION PROGRAMS IN C

```
/*************************************************************************
 *              C programs
 *
 *          includes the programs for the approximate recurrence relation,
 *          binary-divide algorithm, greedy approximate window-control,
 *          and load estimation
 *************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>

#define NMAX 799 /*number of trials in a given run */

/*************************************************************************
 * Approximate Recurrence Relation:
 *   -create a lookup table from the recurrence relation outlined in
 *    section 3.3.1.3
 *   -simulate contention using the lookup table
 *
 * The lookup table has a resolution of 1/(10n), and the algorithm
 * is restarted with new contention parameters if contention is not
 * resolved when the window size is less than 1/(10n).
 *************************************************************************/

/*************************************************************************
 * global variables:
 *
 *************************************************************************/
float Nx[401][401]; /* lookup table for computing N(a,b)
                For current window [a,b), Nx[a][b] contains the
                expected minimum number of contention slots to
                resolve contention. Nx[b][a] contains the
                optimal window for interval [a,b).
                Nx[b][a] is shared by both subroutines:
                N() and S(). */
float Sx[401][401]; /* lookup table for computing S(a,b)
                For current window [a,b), Sx[a][b] is the estimated
                probability of success and Sx[b][a]=Nx[b][a]. */
double p,       /* probability of noise */
     dw;      /* window resolution-- 1/(10n) */
int n;        /* number of initial contending stations */


/*************************************************************************
 * subroutines
 *************************************************************************/

/* prob(), return random # Uniform[0,1)
*/
   double
   prob()
   {
      static double divis = 2147483648.0;
      double y;

      y=rand();
      return (y/divis);
    }


double
g2(a,w,b) /* compute g in dynamic programming formulation */
double w,a,b;
{
  double pow(),x,y,z1,z2,z,p1;
 /* compute g2 assuming false collision-- b invalid */
  p1=(1.0-p)*(1.0-p);
  x=n*(w-a)*pow((1.0-w),(n-1.0))*(1.0-p);
  if (a < 1.0)
    {
   y=pow((1.0-a),(n-0.0));
   z1=x/y;
    }
  else
   z1=0.0;
 /* compute g2 assuming true collision -- b valid */
  x=n*(w-a)*(pow((1.0-w),(n-1.0))-pow((1.0-b),(n-1.0)));
  if (a < 1.0)
    {
   y=pow((1.0-a),(n-0.0))-n*(b-a)*pow((1.0-b),(n-1.0))-pow((1.0-b),(n-0.0));
   z2=x/y;
    }
  else
   z2=0.0;
 /* compute z */
  z=(1.0-p1)*z1 + p1*z2;
  return(z);
}


double
r2(a,w,b) /* compute r in dynamic programming formulation */
double w,a,b;
{
  double pow(),x,y,z,z1,z2,p1;
 /* compute r2 assuming false collision-- b invalid */
  p1=(1.0-p)*(1.0-p);
  if (a < 1.0)
    {
   x=(1.0-w)/(1.0-a);
   y=n+0.0;
   z1=pow(x,y)*(1.0-p);
    }
  else
   z1=0.0;
 /* compute r2 assuming true collision-- b valid */
  if (a < 1.0)
    {
   x=pow((1.0-w),(n-0.0))-n*(b-w)*pow((1.0-b),(n-1.0))-pow((1.0-b),(n-0.0));
   y=pow((1.0-a),(n-0.0))-n*(b-a)*pow((1.0-b),(n-1.0))-pow((1.0-b),(n-0.0));
   z2=x/y;
    }
  else
```

```
   z2=0.0;
 /* compute z */
  z=(1.0-p1)*z1 + p1*z2;
  return(z);
}


double
N(a,b) /* Compute the estimated minimum number of contention slots to resolve
        contention given the current window [a,b). N() is dependent
        on minimizing the term N()/S() for smaller subwindows.
        The terms N() and S() are computed recursively until (b-a)<1,
        at which point the following boundary conditions are used:
          N(a,b)=1, S(a,b)=g(a,b).
       */
int a,b; /* a = lower bound;  b = upper bound */
{
  double N(),S(),g2(),r2();
  double aa,ww,bb,wtest,ntest,stest,nmin,zmin,smax,g,r,l;
  int w,wmin;

  wmin=a;
  if (Nx[a][b] > 0.0) /* do not recompute Nx[a][b] if already in table*/
           {
           nmin=Nx[a][b];
           }
  else if (b <= (a+1) )
           {
           if (b == (a+1) )
             {
             Nx[a][b]=1.0;
             Nx[b][a]=a;
             }
           nmin=1.0;
           }
  else
           {
           aa=dw*(a-1.0);
      bb=dw*(b-0.0);
      zmin=9999.0;
           for (w=a; w<b; w++)
             {
             ww=dw*w;
             g=g2(aa,ww,bb);
             r=r2(aa,ww,bb);
             l=1.0-g-r;
       /* recursion equation for N(a,b) */
             ntest=(1.0+l*N(a,w)+r*N((w+1),b));
        Nx[b][a]=w;
       /* calculate S(a,b) using same w as N(a,b) */
        stest=S(a,b);
        if (stest == 0) /* to prevent possible divide by zero in wtest */
          stest=0.0000001;
       /* calculate N()/S(); store minimums in appropriate tables */
        wtest=ntest/stest;
        if (wtest < zmin)
          {
              zmin=wtest;
              nmin=ntest;
         smax=stest;
         wmin=w;
         }
            }
           Nx[a][b]=nmin;
           Nx[b][a]=wmin;
      Sx[a][b]=smax;
            }
  return(nmin);
 }


double
S(a,b) /* Compute the estimated probability of success given the current
        window [a,b). This value is used in N(a,b) to calculate the
        minimum N(a,b)/S(a,b)  */
int a,b; /* a = lower bound; b= upper bound */
{
  double S(),g2(),r2();
  double s,aa,ww,bb,zz;
  double g,r;
  int w,wmin;

  aa=dw*(a-1.0);
  bb=dw*b;
  if(Sx[a][b] > -1.0 ) /* do not recompute Sx[a][b] */
    {
    zz=Sx[a][b];
    }
  else if (b <= (a+1))
    {
    zz=g2(aa,bb,bb);
    Sx[a][b]=zz;
    if (b !=a)
      Sx[b][a]=a;
    }
  else
    {
    w=Nx[b][a]; /* choose w equal to same value chosen by N(a,b) */
    ww=dw*w;
    g=g2(aa,ww,bb);
    r=r2(aa,ww,bb);
      /* recursion relation for S(a,b) */
    zz=(g+(1.0-g-r)*S(a,w)+r*S((w+1),b));
    }
  return(zz);
}




/****************************************************************
 * main program for Approximate Recurrence Relation
 ****************************************************************/
main()
{
 /*** The following variables have the same definitions as in    ***
  *** the original dynamic programming algorithm              ***/
   int i,j,k,la,lb,lw,ns,pp,wb[41];
   int npkt,kstep,ntrx,icount,ncp,done,vflag;
   double w,sum,v,varian,xmean,conf;
   float pfc,ccc,tx[41];
 /****************************/
  int cc[1000], /* array storing number of contention slots to resolve
             contention for each iteration of the algorithm */
      cflag; /* flag to determine whether confidence interval is
             satisfied */
  float n1,s1,z1; /* temporary values for calculation of N(1,ns)/S(1,ns)=
              theoretical number of contention slots to resolve
              contention */
  double z,
      N(),S(); /* procedures as defined above */
  struct tms buffer; /* used to determine time required to simulate */
  long int tm,tm1;  /*   algorithm and to create lookup tables    */

  srand(1);
 for(pp=20; pp<=20; pp+=10)
  {
/* formulate dynamic programming table */
  /* initialize matrices */
  for (i=1; i<=300; i++)
  for (j=1; j<=300; j++)
   {
   Nx[i][j]=0.0;
   Sx[i][j]= -1.0;
   }

/* iterate on number of stations */
  n=40;
  ns=10*n;
  dw=0.1/(n+0.0);
  p=(pp+0.0)/100.0;
  la=1;
  lb=ns;
  /* N() creates table */
  z=N(la,lb);
  n1=Nx[la][lb];
  s1=Sx[la][lb];
  printf("number of stations= %d0,n);
  printf("number in theory(N),(S)=");
  printf("%4.5f %4.5f0,n1,s1);
  printf("total number of slots= %4.5f0,n1/s1);
  printf("user time, system time0);
  buffer.tms_utime=times(&buffer);
  buffer.tms_stime=times(&buffer);
  tm1=buffer.tms_stime;
  tm=buffer.tms_utime;
  printf("%ld---%ld0,tm,tm1);
  fflush(stdout);

/****************************************************************
 * simulate results
 *
 *
 ****************************************************************/
  cflag=1;
  vflag=1;
  while (cflag == 1) /* repeat NMAX trials until 95% confidence is
                established */
  {
```

```
 for (npkt=0; npkt<=NMAX; npkt++)
 {
  kstep=0;
  done=1;
  while (done == 1) /* repeat contention until minimum has been isolated */
  {
  la=1;
  lb=ns;
  for (k=1; k<=n; k++)
    tx[k]=prob();
    while (done == 1 & lb > la)
    {
    lw=Nx[lb][la];
    w=dw*lw;

    /* collision detections */
    for (k=1,ntrx=0; k<=n; k++)
      if (tx[k] <= w)
        ntrx++;

    /* generate noise */
    pfc=prob();

    /* determine state of network */
    if (pfc <= p || ntrx > 1)  /*collision*/
           lb=lw;
    else if(ntrx == 1)        /*success*/
           done=0;
    else                    /* no transmission */
           la=lw+1;
    kstep++;
    }
  }
  /* contention resolved, store result */
  cc[npkt]=kstep;
 } /* end npkt loop */
/*******************************************************************
* data analysis
*
*
*******************************************************************/
 if (vflag == 1)
   { /* initialize variables if new est of iterations, otherwise
        add values in cc[i] to previous set(s) */
   sum=0.0;
   v=0.0;
   vflag=0;
   for (k=1; k<=30; k++)
     wb[k]=0;
   icount=0;
   }
 for (npkt=0; npkt<=NMAX; npkt++)
   { /* calculte mean and place ncp into appropriate bin for histogram */
   ncp=cc[npkt];
   icount++;
   sum+=(ncp+0.0);
   if (ncp > 4999)
     ncp=30;
   else if (ncp > 1000)
     ncp=29;
   else if (ncp > 200)
     ncp=28;
   else if (ncp > 100)
     ncp=27;
   else if (ncp > 50)
     ncp=26;
   else if (ncp > 25)
     ncp=25;
   wb[ncp]++;
   xmean=sum/icount;
   }
 for (npkt=0; npkt<=NMAX; npkt++)
   { /* calculate variance */
   ncp=cc[npkt];
   v+=(ncp-xmean)*(ncp-xmean);
   }
 varian=v/icount;
 conf=1.96*sqrt((double)(varian/icount));
 printf("number of packets transmitted= %d0,icount);
 printf("avg. no. of contentions (sim)= %9.3f0,xmean);
 printf("variance,confidence= %10.3f %9.3f0,varian,conf);
 fflush(stdout);
 /* if confidence is not great enough, do more iterations */
 ccc=0.05*xmean;
 if (conf < ccc )
   cflag=0;
} /* end while(!cflag) loop */

 /* print distribution */
 printf("noise(percent)= %d0,pp);
 printf("---ncp--- ---frequency---0);
 for (k=1; k<=30; k++)
 {
 printf("%d",k);
 printf("        ");
 printf("%d0,wb[k]);
 }
 printf("user time, system time0);
 buffer.tms_utime=times(&buffer);
 buffer.tms_stime=times(&buffer);
 tm1=buffer.tms_stime/3600;
 tm=buffer.tms_utime/3600;
 printf("%ld---%ld0,tm,tm1);
 fflush(stdout);
 }
}
/*************************************************************************
 * end of main program for Approximate Recurrence Relation
 *************************************************************************/
 .sp 5
/***********************************************************************
 * This program models both the greedy approximate window-control
 * and binary-divide algorithms with or without load estimation.
 * The appropriate sections are outlined be /* ... */ and can
 * be substituted in, depending on which simulation is desired.
 *
 * As in all the other programs, delta = 1/(10n), and the algorithm
 * is restarted if the window size becomes smaller than delta.
 ***********************************************************************/
#include <stdio.h>
```

```
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>

#define NMAX 799 /*number of trials in a given run */
/* prob(), return random # Uniform[0,1)
 */
    double
    prob()
    {
       static double divis = 2147483648.0;
       double y;

       y=rand();
       return (y/divis);
    }

/****************************************************************/
main()
{
  int i,j,k,pp,cc[1000],wb[41];
  int N;
  int npkt,kstep,ntrx,icount,ncp,done,vflag,cflag;
  double p,dw,n,sum,v,varian,xmean,conf;
  float pfc,ccc,tx[51];
  struct tms buffer;
  long int tm,tm1;
  double a,b,w,
        wmv,  /* wmv = moving average of previous windows used to
                 estimate n  */
        C,D;  /* temporary values used in the calculation of the
              approximate greedy window-control algorithm */

  srand(1);
 for(pp=0; pp<=90; pp+=10)
 {

/* iterate on number of stations */
  N=30;
  n=N+0.0;
  wmv=1.0/n;  /* initialize moving window average to 1/n */
  dw=0.1/(n+0.0);
  p=(pp+0.0)/100.0;
  printf("0umber of stations= %d0,N);

/****************************************************************
 * simulate results
 *
 *
 ****************************************************************/
  cflag=1;
  vflag=1;
  while (cflag == 1)
 {
  for (npkt=0; npkt<=NMAX; npkt++)
  {
   kstep=0;
   done=1;
   while (done == 1)
   {
```

```
  a=0.0;
  b=1.0;
  for (k=1; k<=N; k++)
   {
   tx[k]=prob();
   }
   while (done == 1 & (b-a)>dw)
   {
   /* compute new window */
     /*****************************************************
      * use the following calculations for the approximate
      * greedy window-control algorithm:
      *    C=((n-1)*(a+b)+2) / n;
      *    D=(a+b + (n-2)*a*b) / n;
      *    w=(C-sqrt(C*C-4*D))/2;
      *****************************************************
      * or use this calculation for binary-divide:
      *    w=(a+b)/2.0;
      *****************************************************/

    /* collision detections */
    for (k=1,ntrx=0; k<=N; k++)
     if (tx[k] <= w)
       ntrx++;

    /* generate noise */
    pfc=prob();

    /* determine state of network */
    if (pfc <= p || ntrx > 1)  /*collision*/
     {
          b=w;
          }
    else if(ntrx == 1) /*success*/
          done=0;
    else /* no transmission */
          {
     a=w;
          }
    kstep++;
    }
   if (kstep > 5000)
    break;
   }
   /* contention resolved: store result,                 */
   /* compute moving window average and update estimation of n */
   cc[npkt]=kstep;
   /*********************************************************
    * add these calculations if using load estimation:
    *     wmv=(wmv+w)/2;
    *     n=1/(log(1/(1-wmv)));
    *********************************************************/
  } /* end npkt loop */
/*************************************************************
 *
 * include data analysis here (exactly the same as the last program)
 *
 *************************************************************/
/* end of program
 *************************************************************/
```

REFERENCES

[1] W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Computer*, vol. 13, pp. 57-69, Nov. 1980.

[2] G. Fielland and D. Rogers, "32-bit computer system shares load equally among up to 12 processors," *Electronic Design*, vol. 32, pp. 153-168, Sept. 6, 1984.

[3] D. D. Gajski and J. K. Peir, "Essential issues in multiprocessor systems," *IEEE Computer*, vol. 18, pp. 9-27, June 1985.

[4] Y. C. Chow and W. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Transactions on Computers*, vol. C-28, pp. 334-361, May 1979.

[5] Y. T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers*, vol. C-34, pp. 204-217, March 1985.

[6] L. M. Ni and K. Hwang, "Optimal load balancing strategies for a multiple processor system," *Proceedings: 1981 International Conference on Parallel Processing*, pp. 352-357, 1981.

[7] V. Lo and J. W. S. Liu, "Task assignment in distributed multiprocessor systems," *Proceedings: 1981 International Conference on Parallel Processing*, pp. 358-360, 1981.

[8] B. W. Wah and A. Hicks, "Distributed scheduling of resources on interconnection networks," *Proceedings: 1982 National Computer Conference, AFIPS Press*, pp. 697-709, 1982.

[9] J. Y. Juang and B. W. Wah, "Optimal scheduling algorithms for multistage resource sharing interconnection networks," *1984 IEEE Computer Software and Applications Conference*, pp. 217-224, 1984.

[10] F. A. Briggs, K. S. Fu, K. Hwang, and B. W. Wah, "PUMPS architecture for pattern analysis and image database management," *IEEE Transactions on Computers*, vol. C-31, pp. 969-983, Oct. 1982.

[11] J. Y. Juang and B. W. Wah, "Unified window protocols for contention resolution in local multi-access networks," *3rd Annual Conference of the IEEE Computer and Communications Societies*, 1984.

[12] A. S. Tanenbaum, *Computer Networks*. New Jersey: Prentice Hall Inc., 1981.

[13] B. W. Wah and J. Y. Juang, "Resource scheduling for local computer systems with a multiaccess network," *IEEE Transactions on Computers*, vol. C-34, pp. 1144-1156, Dec. 1985.

[14] J. Y. Juang and B. W. Wah, "Channel allocation in multiple contention bus networks," *IEEE INFOCOM Conference*, 1986.

[15] W. Stallings, *Data and Computer Communications*. New York, New York: Macmillan Publishing Co., 1985.

[16] Digital Equipment Corp., Intel Corp., and Xerox Corp., *The Ethernet: Local Area Network Data-Link Layer and Physical Specifications*. Version 1.0, Sept. 30, 1980.

[17] J. Y. Juang, *Resource Allocation in Computer Networks*. Ph. D. Dissertation, Purdue University, Aug. 1985.

[18] G. B. Thomas and R. L. Finney, *Calculus and Analytic Geometry*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1980.

[19] W. Mendenhall and R. L. Scheaffer, *Mathematical Statistics with Applications*. North Scituate, Massachusetts: Duxbury Press, 1973.