A WINDOW PROTOCOL FOR TRANSMISSION OF TIME CONSTRAINED MESSAGES

Wei Zhao
Department of Mathematics
Amherst College
Amherst, MA 01002

John A. Stankovic and Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 87-110
November 1, 1987

A Window Protocol for Transmission of Time Constrained Messages [1]

Wei Zhao
Department of Mathematics
Amherst College
Amherst, MA 01002

John A. Stankovic
Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

November 1, 1987

ABSTRACT

In this paper, we propose and study a new window protocol suitable for transmitting time constrained messages in a multi-access network. Our protocol differs from traditional window protocols in that it explicitly takes time constraints into account. In our protocol, the window is formed based on the latest time to send a message (LS). A major advantage of our window protocol is that a newly arriving message is immediately considered for transmission if its LS is less than that of all pending messages in the system. As a result, our new protocol closely approximates the optimal minimum-laxity-first policy. A performance evaluation through simulation shows that the new window protocol performs well in a wide range of environments, even under overloaded conditions.

[1] This work is part of the Spring Project at the University of Massachusetts funded in part by the Office of Naval Research under contract 048-716/3-22-85 and by the National Science Foundation under grant DCR-8500332.

Contents

1 **Introduction** 1

2 **Model** 4

3 **The Time Constrained Window Protocol** 5

3.1 Problems with Traditional Window Protocols 5

3.2 The Time Constrained Window Protocol without Laxity Ties 8

3.3 Property of the Time Constrained Protocol without Laxity Ties 12

4 **The Time Constrained Window Protocol with Message Laxity Ties** 13

4.1 Recognition and Resolution of Message Laxity Ties 14

4.1.1 Recognition and Resolution of Message Laxity Ties During Window Expansion 15

4.1.2 Recognition and Resolution of Message Laxity Ties During Window Contraction 17

4.2 Property of the Extended Protocol 18

5 **Performance Evaluation by Simulation** 21

5.1 Base-line Protocols 21

5.2 Simulation Model 23

5.3 Simulation Study Results 25

5.3.1 Environment Parameters' Effect on Performance 25

5.3.2 Sensitivity of Protocol Parameters 27

5.4 Summary of the Performance Observations 29

6 Final Remarks 30

List of Tables

1 Procedures and Functions for Stack Used in the Extended Protocol 35

List of Figures

1 The Time-Constrained Window Protocol 36

2 Procedure Contract_Window_and_Send 37

3 Procedure Pop_and_Send 38

4 Procedure Expand_Window_and_Send 39

5 Example 2 — Using the New Window Protocol 40

6 Example 3 — A Message Laxity Tie 41

7 Extended Procedure Expand_Window_and_Send 42

8 Example 3 — Using the Extended Protocol 43

9 Extended Procedure Contract_Window_and_Send 44

10 Effect of Environment Parameters I — $\alpha = 0.01$ 45

11 Effect of Environment Parameters II — $\alpha = 0.10$ 46

12 Sensitivity of δ for Window Protocol I — $\alpha = 0.01$ 47

13 Sensitivity of δ for Window Protocol II — $\alpha = 0.10$ 48

14 Sensitivity of η for Virtual Time Protocol I — $\alpha = 0.01$ 49

15 Sensitivity of η for Virtual Time Protocol II — $\alpha = 0.10$ 50

1 Introduction

In this paper, we propose and analyze a communication protocol for distributed hard real-time systems [28,33]. A *hard real-time system* is one in which the correctness of the system depends not only on the logical results, but also upon the time at which those results appear. Messages transmitted in such systems are referred to as *time constrained*, meaning that a message must be received by a deadline or it is lost. The protocol which we propose in this paper specifically takes messages' time constraints into account, and hence is suitable for the control of communication in distributed hard real-time systems.

The most common communication network used in distributed hard real-time systems is the multiple access network. In this type of network, stations transmit messages via a shared channel. Only one message can be successfully transmitted over the channel at any time. A *collision* occurs if, at any time, two or more messages are transmitted on the channel. No message can be received correctly in the event of a collision.

Based on how the collisions are handled, multiple access communication protocols can be broadly divided into three categories [20,21]:

1. *Inference avoiding* protocols: These protocols operate without taking past history of the channel into account. This category includes ALOHA [1] and various CSMA protocols [10].

2. *Inference seeking* protocols: These protocols make inference on the collision history, and usually solve collisions by partitioning some parameter space of messages. Various tree, window, stack, and urn protocols [2,4,5,7,8,11,13,30,31,32] belong to this category.

3. *Deterministic* or *Collision-free* protocols: These protocols work in such a way that collisions do not occur at all. The Time Division Multiple-Access Protocols (TDMA),

the Bit-Map Protocol [12], the Broadcast Recognition with Alternative Priorities Protocol [27,3,6], and the Multi-level Multi-access Protocol [26] are examples of protocols in this category.

The majority of communication protocols found in these three above categories do not directly address timing constraints. Each of the categories serves different application areas, and hence are valuable in different parts of the requirements space. Recently, protocols for hard real–time communication belonging to the first category have been developed and reported in [34,35]. In this paper we show that our new window protocol, which incidently belongs to the second category, is better than the hard real–time communication protocols of the first category.

Let us now briefly consider the state of the art in time constrained message communication. We begin with an analogy. The design objectives of multiple access protocols and scheduling algorithms are quite similar: Both are for allocation of a serially-used resource to a set of processes [14]. It is known from the theory of hard real–time scheduling that, in the static case, i.e., when all the task characteristics are known a priori, minimum-deadline-first and the minimum-laxity-first scheduling policies are optimal in the sense that they can schedule a set of tasks if there is some policy which can do so [17]. In the dynamic case, these policies also offer better performance than others [9]. Due to this fact, we will adopt in our window protocol a network-wide transmission policy in which the message with the minimum laxity is transmitted first.

Kurose *et al.* suggest a window protocol for real–time communication [13,14], implementing the minimum-laxity-first policy. However, they assume that the laxities of all the messages are constant. This is quite a restrictive assumption. Under this assumption, the minimum-laxity-first policy is identical to the first-come-first-served policy. Panwar *et al.*

[22,23] have also studied the problem of optimal transmission policy for real–time communication. In their work, it is assumed that the length of the i-th message being transmitted on the channel is *independent of what the i-th message is.* In other words, the lengths of messages vary with the order in which they are transmitted. They prove that if the channel is not allowed to remain idle when there is a message waiting to be sent, the minimum-laxity-first policy is the best. Their assumptions are also quite restrictive. In our protocol, all of above assumptions are removed: We allow messages to have arbitrary laxities, and allow message lengths to be determined at the instant they arrive, and hence to be invariant with the order in which they are transmitted.

The literature sometimes associates message priority with real–time communication [38]. That is, deadline information is mapped onto a priority. Using priorities is limited because of the fixed number of priority levels available, and because it can be quite difficult to dynamically assign priorities to accurately reflect deadlines of currently active messages.

In our new window protocol, the window is formed based on message's latest time to send (LS). The management of the window is performed in such a way that a newly arriving message is immediately considered for transmission if its LS is less than that of *all* pending messages in the system. As a result, our protocol closely implements the minimum-laxity-first policy and hence is suitable for time constrained communication. It is also not limited by arbitrary limits on the number of priority levels, nor is it necessary to perform any dynamic analysis of the relative deadlines of active messages. Our performance evaluation through simulation shows that the time constrained window protocol performs well in a wide range of environments.

The remainder of this paper is organized as follows: Section 2 defines the system model. Section 3 describes the new time constrained window protocol when no two messages have

the same laxity. It is shown that under these circumstances, the protocol accurately implements the minimum-laxity-first transmission policy. Section 4 extends the protocol to the case where two or more messages may have the same laxity. Section 5 presents the results of simulation studies comparing the performance of our protocol with that of two baseline protocols — one ideal protocol, and another known as the virtual time VTCSMA protocol [34,35]. The results show that the new window protocol performs very close to the ideal protocol and it performs better than the virtual time CSMA protocol most of the time. Section 6 summarizes the conclusions of the paper.

2 Model

In a multiple access network, a set of nodes are connected to one communication channel. At any given time, only one message can be successfully transmitted over the channel. The maximum end-to-end delay for a bit is τ. We assume that the time axis is *slotted.* The length of a slot is defined to be one time unit. Given that the maximum end-to-end delay is τ, we let the length of a slot be equal to τ. A node can start transmitting a message only at the beginning of a slot. The length of a message is a multiple of the length of a slot. *The normalized end-to-end delay*, α, is defined as

$$\alpha = \tau/\text{Mean of Message Length}.$$

Each message, M, can be characterized as follows:

- Identification number of the message, I_M, is a positive integer. At a given instant of

time, each message waiting on a node should have unique identification number.

- Length, L_M, which is the total number of time units needed to transmit message M;
- Deadline, D_M, is the time by which message M must be received by its destination;
- Latest time to Send the message M, LS_M, is equal to $D_M - L_M$;
- Laxity at time t for message M, $LA_M(\text{t})$, is the maximum amount of time the transmission of message M can be delayed given current time t. Therefore,

$$LA_M(t) = D_M - L_M - \text{t} = LS_M - \text{t}.$$

When it is clear from the context, we may omit argument t as well as subscript M in the above expressions.

From these definitions, it is clear that if we transmit tasks according to their latest time to send, it is equivalent to the minimum-laxity-first transmission policy.

3 The Time Constrained Window Protocol

3.1 Problems with Traditional Window Protocols

In traditional window protocols [2,4,5,7,13,30], which were not designed for time constrained communication, each node maintains a data structure called a *window*. The window simply is a pair of numbers, defining an interval on the axis of some message parameter. Different window protocols use different message parameters, such as message arrival time, node id, etc. Each node continually monitors the channel state, maintaining a current window. If a node senses that the channel is idle and it has a message in the current window, it transmits the message. There are three possible outcomes for the transmission. One, this

transmission is successful if only one node transmits. In this case, the node transmitting the message continues the transmission until completion. Two, a collision results if more than one node transmits its message. In this case, all nodes sending the messages abort transmission. Then, the window is *partitioned* or *split* into two or more smaller windows and the protocol deals with each of these smaller windows separately and recursively. Three, it is also possible that no message is in the current window, hence, no message is transmitted. At this point, every node notices that the channel is idle. In this case, the traditional window protocols either recursively use another smaller window split at the time of a previous collision resolution, if any, or start a new window. We also note that if there is a collision, most of the traditional window protocols have a policy that a newly arriving message is not allowed to join the competition. That is, a newly arriving message must wait until all the old messages in the (split) windows have been transmitted. With this policy, the number of collisions and the variation in response time may be reduced, and hence it is reasonable when the communication is not time constrained. However, as we will show, this policy is not adequate for time constrained communication because in this kind of communication, the most important performance metric is the the ratio of message loss, different from that for non time constrained communication.

Let us now consider using the window approach to implement the minimum-laxity-first transmission policy for time constrained communication. One may think that it is very simple: set the window along the axis of messages' LS and use the above approach to manage the window. Would this trivial method work? Let's consider the following scenario.

Example 1: Assume that at time t = 0, there are two messages M1 and M2 on different nodes in the system. $LS_{M1} = 4$ and $LS_{M2} = 16$. The length of M1 is one time unit and M2 is two time units. Also assume that the initial window size is 20, and hence the initial window is [0, 20).

Because both messages are in the window, they are sent out at the same time, causing a collision. Assume that the collision is detected at time t = 1; the nodes abort the transmission. At t = 2, the channel is idle again. Now, with the traditional window technique, the window should be *split* into two small windows [2, 11) and [11, 20), and then be processed separately. Assume that the protocol deals with the window [2, 11) first. Because only M1 is in this window, its transmission is now successful.

The transmission of M1 completes at t = 3 and the channel is idle at t = 4. Assume that while transmitting message M1, a new message, M3, arrives which has an LS of 6 and length 1. According to the traditional window protocol, this message would not take part in the protocol at this time. Hence, the window [11, 20) is used next, and M2 is transmitted.

The transmission of M2 finishes at t = 6 and the channel becomes idle at t = 7. At this time, the current time t is larger than M3's LS, and hence M3 is lost.

The loss of M3 occurs because the traditional window approach cannot *accurately* implement the minimum-laxity-first policy even if the window is set along the axis of the messages' LS. With the minimum-laxity-first policy, M3 should be transmitted before M2. This example is not an exceptional case for the traditional window approach. In the design of our new protocol, this above problem is solved. We do this by letting the lower bound of the window equal the current real time t. After a collision, we *modify* the window size rather than *split* the window. The method we use to modify the window implements the minimum-laxity-first policy by allowing a newly arriving message to take part in the protocol immediately after an ongoing message transmission completes, if the new message has its LS in the current window. In this way, the newly arriving messages with small laxities can always be considered early, rather than waiting until all the old messages in the window have been transmitted. This is a major advantage of our protocol.

3.2 The Time Constrained Window Protocol without Laxity Ties

We now present our new protocol in detail. In our protocol, the window has the form [t, up) where t is the current real time, and up is the upper bound of the current window. We say that *a message, M, is in a window* [t, up) if t $\leq LS_M <$ up. Messages are queued at each node according to their LS, which is the deadline of the message minus the length of the message. On each node, in addition to the message queue, there is a stack that saves the information needed to resolve contentions, making recursive execution of the protocol possible. Each item of the stack is an integer number u which is the upper bound of a (previous) window. Throughout Section 3, we assume that at any time t, different messages in the system have different laxities. That is, there is no tie among them. The extension to the case where two or more messages may have the same laxity is discussed in Section 4.

The pseudo code of our window protocol is presented in Figure 1. At the time of initialization, the upper bound of the window is set to be t+δ where t is the current time, and δ is the protocol parameter for the initial window size. Any node having a message in the window attempts to transmit the message.

At the beginning of each time unit, every node first drops any message from the message queue if its LS is less than the current time t. In addition, the stack on each node is *cleaned*, — any item in the stack whose value is less than or equal to the current time t is discarded. Then, each node calls a subroutine **get_state** to obtain the information on the state of the channel. We define five channel states:

1. `channel_collision`: Two or more messages are being transmitted over the channel.
2. `channel_idle_after_a_collision`: The channel is idle now, and there was a collision over the channel one time unit ago.

3. channel_busy: One message is being transmitted over the channel.

4. channel_idle_after_a_successful_transmission: The channel is idle now, and a message transmission occurred one time unit ago.

5. channel_continue_idle: The channel is idle now, and was also idle one time unit ago.

Depending on the channel state, each node takes the following actions:

1. channel_collision: The nodes which are transmitting messages immediately abort transmission. Then, in the next time unit, each node should observe that the channel is in the state of channel_idle_after_a_collision.

2. channel_idle_after_a_collision: Procedure Contract_Window_and_Send is called. See Figure 2 for the pseudo code. Each node realizes that two or more messages are in the current window so that the window size should be reduced to resolve the collision. Hence, the upper bound of the window, up, is reduced to $t + \lceil (up - t)/2 \rceil$, i.e., the middle point of the old window. Then, a node sends out the collided message again if its LS is in the reduced window.

   It is possible for the old value of up to be equal to or less than the current time t. [2] In this case, Procedure Pop_and_Send is called. For the pseudo code, see Figure 3. In this procedure, if the stack is not empty, the upper bound of the window, up is set to be the value of up which is on the top of the stack. If the stack is empty, a new window with enlarged size δ, where δ is the parameter for the initial window size, is used. In either case, a message can be sent if its LS is in the new window.

[2]This happens if the old window has its size equal to 2.

3. channel_busy: The node transmitting the message continues the transmission. The other nodes wait until the transmission completes. After the transmission completes, each node notices that the channel state is channel_idle_after_a_successful_transmission.

4. channel_idle_after_a_successful_transmission: Procedure Pop_and_Send is called (see Figure 3). At this point, the message in the current window has been processed [3]. That is, if the current window resulted from a previous collision, then the protocol extends the window to its original upper bound before the collision. Otherwise, a new window is created with the additional size δ.

5. channel_continue_idle: Procedure Expand_Window_and_Send is called. The pseudo code for the procedure is in Figure 4. When the channel is in this state, there is no message in the current window. If the stack is empty, each node simply expands the window to increase its size by δ. However, by keeping additional information, we can improve the performance of the protocol. That is, a non trivial inference can be made if the previous window information is retained. Specifically, if we know that the current window was created from a previous window because a collision occurred, and there is no message in the current window (otherwise the channel cannot be continuously idle), then there should be at least two messages in the interval [up, top(stack)) where top(stack) indicates the upper bound of the previous window. Knowing this, the new value of up is set to the middle point of the current up and top(stack), i.e., $\lceil (up + top(stack))/2 \rceil$, rather than expanding it to the old window's upper bound which is on the top of the stack. This avoids one "guaranteed" collision.

   Note that because of the round-up effect, if the old value of up is equal to top(stack)

[3] This would not be true if a newly arrived message has its LS in the current window. However, because the lower bound of the window is always set to be the current time, the new message has a chance to be considered.

-1, the new value of up will be the same as top(stack). In this case, the top of stack in popped out and that value is used as the new value of up.

Once the new window is determined, if a node has its message with LS in the new window, the node transmits the message.

The next example helps us better understand the general behavior of the time constrained protocol just presented.

Example 2 Let us take the same scenario as in Example 1, Page 6. Messages M1 and M2 are in the system at t $= 0$. M1 has its LS equal to 6 and length 1, and M2 has its LS equal to 16 and length 2. We also assume that the initial window size has been chosen to be 20, i.e., $\delta = 20$. At t $= 0$, M1 and M2 both are transmitted over the channel, causing a collision. See Figure 5.a.

At t $= 1$, the collision is detected. The transmissions of M1 and M2 abort. At t $= 2$, The channel is idle after the collision. Now, the window size is reduced. The new window is $[2, 11)$. The old value of up is pushed onto the stack at each node. Top(stack) $= 20$. Because M1 is in the new window, it is transmitted. See Figure 5.b.

At t $= 3$, M3 arrives in the system. Transmission of M1 completes. At t $= 4$, the channel is idle. The protocol now goes back to the previous window $[4, 20)$. The stack is empty. Because both M3 and M2 are in the window, they start to be transmitted, causing a collision. See Figure 5.c.

At t $= 5$, the collision is detected. The transmissions of M3 and M2 abort. At t $= 6$, the channel becomes idle. The window size is reduced. The new window is $[6, 13)$. M3 is transmitted. See Figure 5.d. Only after the transmission of M3, will M2 be transmitted.

3.3 Property of the Time Constrained Protocol without Laxity Ties

We now show that with the protocol discussed above, the minimum-laxity-first policy is preserved.

Lemma 1 At any time t, with the time constrained window protocol without laxity ties, for any two messages M and M', $LA_M(t) < LA_{M'}(t)$ if and only if $LS_M < LS_{M'}$.

Proof This lemma follows from the definitions of LA(t) and LS.

We say that *a message, M, is in the system at time* t if at time t, M has arrived at the system, and M has neither started its successful transmission, nor has it been lost, i.e., M's arrival time $\leq t \leq LS_M = D_M - L_M$ where D_M is the message deadline, and L_M is the length.

Lemma 2 With the time constrained window protocol without laxity ties, at any time t, if a message M starts a successful transmission, LS_M is the minimum among all the LS values of messages which are in the system at time t.

Proof Assume at time t, message M starts a successful transmission in a window [t, up). From the protocol, $t \leq LS_M <$ up. If M does not have the minimum LS among all the messages in the system at t, then, there must be another message M' with $t \leq LS_{M'} < LS_M$. Hence, $t \leq LS_{M'} < LS_M <$ up. This means that both M and M' are in the window [t, up). Consequently, the transmission of M cannot be successful. This is a contradiction. []

THEOREM 1 The window protocol described in this section preserves the minimum-laxity-first policy. That is, with this protocol, at any time t, if a message M starts its successful transmission, M has the minimum laxity among all the messages which are in the

system at time t.

Proof According to Lemma 2, any message successfully transmitted by the protocol has the minimum LS in the system at the time when the transmission starts. Then, following Lemma 1, this message must also have the minimum laxity at the time when its transmission starts. []

In summary, our new window protocol differs from the traditional ones not only in the semantics given to the time axis of the window, but also in the way we manage the window. As a result, in the case where there is no tie on message laxity, the optimal minimum-laxity-first transmission policy is accurately implemented with our protocol.

4 The Time Constrained Window Protocol with Message Laxity Ties

In Section 3, we presented a version of our new window protocol for the case when there is no tie on message laxity. In reality, it is possible that multiple messages (on different nodes) have the same laxity, and hence have the same LS value. Hence, such messages will always cause a collision when they are in a window whose axis is based on messages' LS. Splitting the window as in a traditional window protocol, or reducing the window size as we do in our new protocol cannot break the tie. A practical window protocol must be able to recognize message ties and resolve them. In this section, we extend the time constrained protocol to handle message laxity ties.

The idea to handle message laxity ties is as follows: At first, all the nodes should be able to recognize the tie at some point in time. Once a tie is recognized, every node involved in the tie randomly modifies its message's LS value between the earliest next channel available time, t + 2, and $D_M - L_M$, to try to resolve the tie. In other words the messages have their LS value reduced in time in a random fashion in order to resolve the tie. Later we discuss the impact of this policy on accurately emulating the minimum laxity algorithm.

Clearly, the messages that have collided in a window are the messages with possible laxity ties. The stack used to maintain the window history now needs to keep information of not only the upper bounds of the windows in which collision occurred, but also the identification numbers of messages which caused the collision. In this way, once a tie is recognized by all the nodes, a node is able to know which of its messages is involved in the tie if any. [4] That is, a stack item now is a tuple (u, i) where u is the upper bound of a window which had a collision and i is the identification number of the message which caused the collision. Note that on different nodes, the values of i represent different messages. In the case that a node has no message involved in a collision, the value of i is zero. The functions and procedures used for the operations on the stack are listed in Table 1.

4.1 Recognition and Resolution of Message Laxity Ties

In a traditional window protocol, a tie is recognized when the window size is reduced to one and there is still a collision. As we will see, in our protocol, due to the style of window management, a tie can be recognized both during window contraction and during window expansion.

[4] The details of how this can be done will be explained in the later part of this section. Here, we concentrate on the changes to the stack.

4.1.1 Recognition and Resolution of Message Laxity Ties During Window Expansion

In our protocol, a tie can be recognized when a window expands to its upper limit. To help understand this, let's consider the following example.

Example 3 At t = 0, there are two messages (on different nodes) in the system. Each message has a unit length. M1 and M2 have LS of 10 (a tie!). Assume that the initial window size is 20, so the initial window is [0, 20). M1 and M2 both start transmitting, causing a collision. See Figure 6.a.

At t = 1, the collision is detected. Transmission of M1 and M2 abort. At t = 2, the channel is `idle_after_the_collision`. Each node pushes the old value of up onto stack. Top_u(stack) = 20. The window size is reduced. The new window is [2, 11). Because both M1 and M2 are in the window, they start transmitting, again causing a collision. See Figure 6.b.

At t = 3, the collision is detected. The transmission of M1 and M2 abort. At t = 4, the channel becomes `idle_after_the_collision`. The old value of up is pushed onto the stacks at each of the nodes. The window is reduced to [4,8). Top_u(stack) = 11. No message is in the current window. See Figure 6.c.

At t = 5, all the nodes notice that the channel is `continuing_idle`. The window expands. The new window is [5, 10). Top_u(stack) still is 11. Again, no message is in the current window.

At t = 6, the channel is `continuing_idle`. If we were using the version of the protocol for the case of no laxity tie, described in the last section, the window would be expanded to [6, 11). Because both messages are in the window, they are sent, causing the collision. Then

the window would be reduced and expanded as above, until time 10 passes after which both messages, M1 and M2, are lost.

From this example, we see that when the channel is in the state of continuing_idle and the current window [t, up) is to be expanded to its upper limit, i.e., the old value of up = top u(stack) - 1, two or more messages may have the same laxity which is equal to top_u(stack) - 1. The reason is that two or more messages were in an old window with upper bound of top_u(stack), causing a collision and forcing the upper bound of the old window to be pushed onto the stack. Now that no message is in the window [t, up) = [t, top_u(stack) - 1), we conclude that the two or more collided messages may have their LS equal to top_u(stack) - 1. Contraction or expansion of the window cannot resolve this kind of tie. We propose the following extension to Procedure Expand_Window_and_Send to resolve this kind of tie.

The pseudo code of the *extended Procedure Expand_Window_and_Send* is shown in Figure 7. In the extended Procedure Expand_Window_and_Send, if the value of up is equal to top_u(stack) - 1, the expansion on the window stops. A probabilistic scheme is invoked: The messages, which collided in the window with the upper bound of top_u(stack), have their ids equal to top_i(stack) (on different nodes). If such a message exists in a node, the node draws a random real number in the interval of (0, 1). In the case that the random number drawn is larger than P — a protocol parameter, the message is sent. Otherwise, the message's LS is modified for a future consideration of transmission. This is done by assigning LS a random value between the earliest next channel available time, t + 2, and $D_M - L_M$, which is the latest time this message could be sent. It is clear that in this version of the protocol, message parameter — LS — is used for the purpose of controlling the transmission. It is initialized to $D_M - L_M$.

With the above extension, let's consider the situation in Example 3 again.

Example 3 (cont.) See Figure 8. Assume the same scenario as described in the original Example 3 until t = 6.

At t = 6, the channel is in the state of `continuing_idle`. The old value of up is 10 and top_u(stack) = 11. Because top_u(stack) - 1 = 10 = up, the probabilistic scheme is invoked. The nodes which have M1 and M2 draw the random number independently. Assume that based on the random numbers drawn, the node with M1 decides to transmit again and the node with M2 decides to modify M2's LS to 9. M1 is successfully transmitted over the channel. The top of stack is popped as the new value of up. Note that although M2 is still in the window, the protocol explicitly prohibits the transmission of M2 at this time. See Figure 8.d.

At t = 7, the transmission of M1 completes. And at t = 8, the channel state is `channel_idle_after_a_transmission`. The top of the stack is popped as the new value of up. Hence, the new window is [8, 20). Now, M2 can be transmitted. See Figure 8.e.

Note that if at t = 6, it is randomly decided that both M1 and M2 are to be sent again. A collision would occur. Then, the protocol would recognize the situation again, use the same random scheme to try to break the tie again. This process repeats until the tie is actually broken or the messages are lost.

4.1.2 Recognition and Resolution of Message Laxity Ties During Window Contraction

In our protocol, as in traditional window protocols, a tie can be recognized when the window size is reduced to one and there is still a collision.

When the channel is in the state of channel_idle_after_collision, Procedure Contract_Window_and_Send (for which, the original pseudo code is in Figure 2, and the discussion is on Page 9) is called. In this procedure, the window size is reduced by letting up $= t + \lceil (up - t)/2 \rceil$. If two or more messages have the same laxity, the window will (eventually) be reduced to a point where up $= t + 1$. At this point, if up is further reduced by the formula up $= t + \lceil (up - t)/2 \rceil$, the new value of up is equal to the old one. This means that the window has been reduced to its minimum and the messages in window $[t, up)$ have LS equal to the current time t. Hence, a collision would occur again if the new window, which is the same as the old one, is just simply used. To resolve the tie, we extend Procedure Contract_Window_and_Send as follows (for the pseudo code, see Figure 9): if up $= t + 1$, the protocol does not reduce the window size, but invokes a probabilistic scheme: Each node which has a message in the current window draws a random real number in the interval of (0, 1), and if the number drawn is larger than a pre-defined probability P (which is a parameter of the protocol), the node transmits the message. In the case that the random number is not larger than P, if the message's laxity is zero, the message is discarded because there is no chance to transmit it before its time constraint. Otherwise, the message's LS is modified for possible future transmission.

4.2 Property of the Extended Protocol

For the extended protocol, the tied message's LS values are modified, and hence the minimum-laxity-first policy is not always preserved as in the case when there is no tie. However, the following theorem shows that in the extended protocol, an individual message M will still be sent according to the minimum-laxity-first policy if the tie among messages

with larger laxity is resolved after message M's arrival, or if the tied messages have a smaller laxity than M.

Lemma 3 For the extended protocol, at time t, consider any 2 messages M and M' which do not have a tie. If

1. M' has no laxity tie with any other message, or
2. M' does have a laxity tie with one or more other messages but the tie has not been resolved at time t

then,

$$LS_M < LS_{M'} \text{ if and only if } LA_M(t) < LA_{M'}(t).$$

Proof When the first condition is true, this lemma holds from Lemma 1. When the second condition is true, this lemma holds because only *after* a tie resolution, can an involved message's LS be reduced. []

Lemma 4 With the extended protocol, if at any time t, the protocol is not performing the tie resolution (i.e., not using the probabilistic scheme), and at the same time t, a message M starts its successful transmission, then M has the minimum LS value among all the messages which are in the system at time t.

Lemma 4 simply says that if the extended protocol is not resolving ties, then Lemma 2 is still true. The proofs of both Lemmas 2 and 4 are similar. The proof for Lemma 4 is hence omitted.

THEOREM 2 Assume that a message M arrives at the system at time t_1. Let t_2 ($> t_1$) be the earliest time at which the channel is idle, and t_3 be the time at which message M

leaves the system (either being transmitted or lost). Then for any message M' transmitted in the time interval $[t_2, t_3)$, $LA_{M'} \leq LA_M$, unless M' has a laxity tie with other messages and the tie is resolved at $t_0 < t_2$.

Proof For any message M' without a laxity tie with other messages, according to Lemma 1 and Lemma 4, M' may be transmitted in the time interval of $[t_2, t_3)$ only if $LA_{M'} < LA_M$. Hence, the theorem holds for this kind of message.

Now, consider the messages with laxity tie. Let $\{M_1, M_2, \ldots, M_n\}$, $n \geq 2$, be a set of messages which have a tie on their laxity, i.e., $LA_{M_1} = LA_{M_2} = \ldots = LA_{M_n}$. Also, their laxity is larger than that of message M, i.e., $LA_{M_1} > LA_M$. We need to show that if the tie is not broken at t_2, then no message in $\{M_1, M_2, \ldots, M_n\}$ could be transmitted in the interval of $[t_2, t_3)$.

First, we show that it is impossible for the tie resolution for $\{M_1, M_2, \ldots, M_n\}$ to occur in the time interval of $[t_2, t_3)$. If this is not true, i.e., the tie is resolved at t_0 and $t_2 \leq t_0 < t_3$, then the extended protocol, at t_0, uses either the extended Procedure Expand_Window_and_Send (Figure 7) or the extended Procedure Contract_Window_and_Send (Figure 9). If at t_0, the extended Procedure Expand_Window_and_Send is used, then the old window has the form of $[t_0, LS_{M_1})$. And there should be no message in that window. However, because $t_2 \leq t_0$ and $LS_M < LS_{M_1}$, message M is indeed in window $[t_0, LS_{M_1})$. Hence, it is impossible that at t_0 the extended procedure Expand_Window_and_Send is used. It is also not possible that at t_0 the extended Procedure Contract_Window_and_Send is used. If the extended Procedure Contract_Window_and_Send is used, the window size must be one. And the lower bound of the window, i.e., the current time, t_0, is equal to the original LS value of tied messages. This certainly cannot be true for $M_1, M_2, \ldots,$ and M_n because before the tie resolution, $LS_{M_1} = LS_{M_2} = \ldots = LS_{M_n} > LS_M \geq t_3 - 1 \geq t_0$. Hence, in any

case, the tie resolution for $\{M_1, M_2, \ldots, M_n\}$ cannot happen in the time interval of $[t_2, t_3)$.

Therefore, if the laxity tie among $M_1, M_2, \ldots,$ and M_n is not resolved before t_2, the resolution will not happen before t_3. Consequently, by Lemma 3, since $LA_{M_1} = LA_{M_2} = \ldots = LA_{M_n} > LA_M$, in the time interval of $[t_2, t_3)$, $LS_{M_1} = LS_{M_2} = \ldots = LS_{M_n} > LS_M$ holds. Then by Lemma 4, no message in the set of $\{M_1, M_2, \ldots, M_n\}$ can be successfully transmitted in the time interval of $[t_2, t_3)$. Hence, the theorem is proven. []

In summary, with the extended protocol, the minimum-laxity-first cannot always be guaranteed because of tie resolutions which modify the LS values. However, with Theorem 2, we see that under certain conditions for an individual message, M, the minimum laxity policy can still be preserved. The conditions are that messages with larger laxities (1) have no laxity ties, or (2) have their ties un-resolved by the time M arrives.

5 Performance Evaluation by Simulation

In this section, we evaluate the performance of the new window protocol with its extensions. We first introduce two baseline protocols to be used for comparison purposes. Then, we present the simulation model and discuss the simulation results.

5.1 Base-line Protocols

The first baseline protocol is called *Centralized Minimum Laxity message transmitted first*, abbreviated as CML. In this protocol, transmission of all messages is assumed to be scheduled by a centralized controller. This controller contains perfect knowledge about the nodes

and the channel and experiences no communication overhead. It schedules the transmissions of messages such that the message with the minimum laxity is transmitted first. Obviously, this algorithm is an ideal one, not realizable in practice. We use it to provide an upper bound on performance.

The second baseline protocol is called *Virtual Time CSMA-L* [34,35]. It has been previously shown that in terms of various performance metrics and stability, in a wide range of hard real–time communication environments, the virtual time CSMA-L protocol performs well and is better than a traditional CSMA protocol. We use it here to serve a baseline and show that the new time constrained window protocol is even better.

In the virtual time CSMA-L protocol, each node maintains two clocks: a real–time clock and a virtual time clock. Messages waiting to be transmitted are queued in the order of their LS values. Whenever a node finds the channel idle, it resets its virtual clock to equal the real clock. The virtual clock then runs at a higher rate, $\eta \geq 1$, than the real clock. A node transmits its first message waiting in the queue when the time on the virtual clock is equal to the LS value of that message. If there is a collision, a probabilistic scheme is invoked for resolution. It is clear that if there are no collisions, this protocol also implements the minimum-laxity-first transmission policy.

One actually may think of the virtual time protocol as a simplified form of the window protocol in which the "window" keeps expanding at a constant rate until either a message transmission starts, or a collision happens. In the latter case, the probabilistic resolution scheme is used immediately, rather than modifying the window.

5.2 Simulation Model

A simulation model is developed and used to evaluate the performance of the above protocols. The simulation program is written in Simscript II.5, and runs in an ULTRIX environment on MicroVAX-II. The simulation model is parameterized by the distributions of message arrivals, transmission times, and laxities.

In the simulations reported in this paper, messages arrive as a Poisson process. Message lengths are exponentially distributed with mean = 100 (corresponding to $\alpha = 0.01$), or mean = 10 (corresponding to $\alpha = 0.1$). Message laxities are uniformly distributed from [0, 2*AL] where AL is the average laxity.

In each simulation run, statistics are reset at the simulation time equal to max[100*mean of message inter-arrival time, 2*mean of message laxities] simulation time units. Then the statistics are collected after another (5,000 * mean of message inter-arrival time) simulation time units. We observed that with this setting for the simulation length, the collected data are within 90 % confidence interval.

We consider the case of *infinite* population of nodes in the system [14,29]. That is, in the simulation model, it is assumed that a message always arrives at a node where no message is waiting. Note that the infinite population presents the worst case for a multi-access network. It maximizes the number of messages in a window, and hence causes maximum number of collisions.

The system load, L, is defined as

$$L = \text{Message Arrival Rate} * \text{Mean of Message Length}.$$

In our simulations, when the system load and the mean of message length are defined, the message arrival rate is decided by the above formula.

For simplicity, in all the simulations, parameter P, the probability of immediate retransmission in the resolution of message laxity tie, takes a value of 0.5.

For each run of the simulation, the following performance measures are collected:

1. *Ratio of Message Lost*, ML which is defined as

$$ML = \frac{\text{TNML}}{\text{TNMT} + \text{TNML}},$$

   where TNML is the total number of messages lost, and TNMT is the total number of messages transmitted.

2. *Effective Channel Utilization*, ECU, defined as

$$ECU = \frac{\text{Total Time Units Channel is Used for Transmitting Messages}}{\text{Total Time Units Simulated}}.$$

3. *Collision Channel Utilization*, CCU, which is

$$CCU = \frac{\text{Total Time Units Channel is Wasted due to Collisions}}{\text{Total Time Units Simulated}}.$$

4. *Normalized average Transmitted Message Length*, NTL, which is defined as

$$NTL = \frac{\text{Average Length of Transmitted Messages}}{\text{Average Length of Arrived Messages}}.$$

A good protocol should minimize the message loss, ML and the collision channel utilization, CCU, and maximize the effective channel utilization, ECU. A protocol which does not have a bias toward short or long message should have its NTL close to 1. As shown in [35] and also confirmed in this simulation study using the minimum-laxity-first policy results in NTL very close to 1. That is, the minimum-laxity-first policy has no bias based on the message length. Consequently, when using the minimum-laxity-first policy, ECU directly depends on ML and can be decided by (1 - ML) * L = ECU. We also find that

CCU for the window protocol is very small when the ML is close to the optimal (minimum) point. Based on these facts, to save space and time, in this report, we concentrate only on the performance metric ML. We would like to refer our enthusiastic reader to [36] for the complete set of data from the simulations.

5.3 Simulation Study Results

We discuss the simulation results in two parts. The first part shows the effect of the environmental parameters on the performance of the protocols, and the second shows the parameter sensitivity of the protocol.

5.3.1 Environment Parameters' Effect on Performance

In this part of the simulation, we study how the application environment parameters such as the average message laxity (AL), system load (L), and the normalized end-to-end delays (α) affect the performance of the baseline and the new window protocol.

Two cases of simulation studies are conducted. In the first case, α is 0.01, and in the second case, $\alpha = 0.10$. In both cases, the system load changes from 0.1, to $0.5, 1.0$, and 2.0, and the mean of message laxity changes from 1 to $10^{4.5}$ in a logarithmic scale. Figures 10 and 11 present the results, plotting the percentage of message loss (ML) vs. the mean of message laxities (AL) for the CML, window, and virtual time protocols.

From Figures 10 and 11, we see that, when the laxity increases or the system load decreases, the performance of each protocol improves as expected. However, after a certain increase in laxity, the improvement is saturated, i.e., the message loss does not decease even though the laxity is further increased.

We note that when the load is light (L = 0.1), the laxity is tight (up to 10^2), and α is small ($\alpha = 0.01$), the performance of three protocols is very close. See Figures 10 and 11. On the other hand, when the system load is heavy, the message laxity is large, or α is large, the performance often is different. Our new window protocol is often better and never worse than the virtual time protocol. This reconfirms the general belief held by the computer communication community that a deterministic collision resolution scheme such as that found in window protocol performs better than a probabilistic one such as the one found in the virtual time protocol.

A careful reader may further notice that the differences between CML and the window protocol, and CML and the virtual time protocol do not always decrease as the laxity increases. Actually, in many instances (see Figures 10.c, 11.b, 11.c, and 11.d), it increases. Although this sounds contradictory to one's intuition, the explanation is quite simple: this phenomenon is due to the fact that when the laxity increases, the number of active messages in the system increases. The reason why increasing laxity increases the number of active messages is as follows: When the messages which have long laxities arrive, they typically wait longer in the system because the protocol favors the short laxity messages and the long laxity messages have a lot of time to still meet their time constraints. The increased number of active messages in the system causes more collisions and hence the performance of protocols such as the window and virtual time, which always takes certain time (cost) to solve a collision, does not improve with increase in laxity as much as the ideal baseline protocol CML. We call this phenomenon the *laxity abnormality in real-time scheduling.* We will observe a similar phenomenon again in the next sub-section.

5.3.2 Sensitivity of Protocol Parameters

In the simulations reported above, the protocol parameters — δ for the window protocol and η for the virtual time protocol — are selected in such way that the best performance of the system is achieved. The sensitivity of the system performance to the selection of the protocol parameters is the subject of this sub-section.

We conduct two cases of simulation studies of the window protocol to test its sensitivity to the protocol parameters. The first case is for $\alpha = 0.01$, and the second case is for $\alpha = 0.10$. In each case, the average laxity (AL) takes values of 10, 100, and 1000, and the system loads (L) are 0.1, 0.5, 1.0, and 2.0 respectively. The value of δ, the parameter for the initial window size, changes from 10^0 to 10^4 in a logarithmic scale. Figures 12 and 13 show the results.

For the purpose of comparison, we also show the simulation results for the virtual time protocol in Figures 14 and 15, where the simulation parameters are set the same as for window protocol except that instead of varying δ, we vary η, the rate at which virtual clock runs, changes from 10^0 to 10^4.

From Figure 12, we notice that in the case when the normalized end-to-end delay is small, i.e., $\alpha = 0.01$, the performance of our window protocol is very stable in terms of the selection of the value δ: As the δ increases from 10^0, the message loss decreases and reaches its minimum no later than when $\delta = 10^2$. Then the message loss stabilizes at the minimum value up to $\delta = 10^4$ where the simulation stops.

Now consider the case where α is large, i.e., equal to 0.1 (Figure 13). When the system load is light ($L = 0.1$) or medium ($L = 0.5$), the sensitivity of the window protocol with respect to the choice of δ is almost the same as in the case where $\alpha = 0.01$. When the load

is high ($L = 1.0$), we see that the performance is stable for the values of $\delta = 10$ to 100. Even when the system is extremely overloaded ($L = 2.0$), there are still certain ranges of δ values in which the system is stable.

The virtual time protocol has been reported to be insensitive in terms of the protocol parameter η [20,21,34,35]. However, if we compare each of corresponding cases of the two protocols, i.e., Figures 12 and 14, and Figures 13 and 15, we find that the new window protocol does much better. With the window protocol, the range in which the system is stabilized is always much greater than with the virtual time protocol.

In these sensitivity tests, we can observe another kind of the laxity abnormality. In Figures 13.d, 14 and 15, we see that when the laxity increases, message loss may even increase. The situation becomes worse when the system is highly loaded (L = 1) or overloaded (L = 2). We believe that the same explanation as stated in Section 5.3.1 still applies: As the laxity increases, the number of active messages in the system increases. Hence, the chance of collision increases which increases the overhead of these algorithms. Hence the performance degrades with respect to the ideal baseline where it is assumed that it takes zero time to solve a collision. A further observation is that when the protocol parameter is far from the optimal value, the system is more likely to be in an abnormal state. However, if we compare the corresponding cases of the window and the virtual time protocols, we see that at least in these simulations, the laxity abnormality is observed much less often for the window protocol than for the virtual time protocol. Indeed, in Figures 12 to 15, the laxity abnormality appears only once (Figure 13.c) for the window protocol, but 7 times (Figures 14.b, 14.c, 14.d, 15.a, 15.b, 15.c, and 15.d) for the virtual time protocol — giving us another reason to favor the new window protocol.

From the data in Figures 12 and 13, we also notice that for a given α, one can always

easily choose a δ value independently from the system load and message laxities such that the performance is at or very close to the minimum value. For example, when $\alpha = 0.01$, any value for δ between 10^2 and 10^4 is very good. And when $\alpha = 0.10$, a δ value around 10 may keep system in an optimal performance stage. Because in practice, α is often a fixed parameter of the network working environment, the above fact indicates that the implementation and maintenance of the protocol would not be difficult in terms of the parameter setting.

5.4 Summary of the Performance Observations

We now summarize the performance observations as determined by the simulation studies:

1. Most of the time, the new window protocol performs very close to that of CMLF which is an un-realizable and ideal protocol that provides the upper bound on the performance.

2. The window protocol often performs better and never worse than the virtual time protocol which in turn has previously been shown to work well and be better than a general CSMA protocol [34,35].

3. The window protocol is extremely insensitive to the choice of parameter δ. For a given α, one may identify good values of δ independently of the system load and message laxity. Moreover, the window protocol is often much more stable than the virtual time protocol though the latter has been shown to be stable most of the time [20,21,34,35].

4. The laxity abnormality, i.e., the phenomenon that when the laxity is relaxed, the system performance may not be improved, or may even become worse, is observed for both the window and virtual time protocols. However, with the window protocol, the

abnormality occurs less frequently, or to a lesser degree if it does occur, than with the virtual time protocol.

6 Final Remarks

We have proposed and studied a new window protocol suited for time constrained communication. The new protocol differs from the traditional window protocol approach in the sense that it explicitly takes message's time constraints into account, in particular, the message deadlines. The management of window is done in such a way that a newly arriving message always has a chance to be considered for the transmission if its latest time to send (so that it reaches its destination before the deadline) is less than that of the active messages in the system. As a result, our new protocol accurately implements the optimal minimum-laxity-first policy if there is no tie among message laxities. In case there is a laxity tie among some messages, the minimum-laxity-first policy still can be preserved for an individual message M if the tied messages have a smaller laxity or if the tie has not been resolved at the time message M arrives. Simulation studies shows that it performs well in a wide range of environments, even under overloaded conditions.

Turning our attention to implementation support for our protocol, an exact implementation requires the synchronization of the clocks at all nodes. In a distributed system, clock synchronization is an interesting and challenging problem. However, our protocol is robust in the sense that it will continue functioning even if clocks are not perfectly synchronized. Of course, in this situation, the performance of the protocol may deteriorate because mes-

sage transmission will not be exactly according to the minimum-laxity-first transmission policy. (In fact, as the protocol stands, laxity ties distort the message transmission policy.) Further studies are needed to evaluate the effect of a given maximum skew of the clocks on the desired performance metrics.

In this paper and in our previous studies [34,35], we have used virtual time and window protocols to implement specific transmission policies suitable for hard real–time communication. This method can be generalized to many other network applications. For example, in [8], a special window protocol is introduced for load balancing. And in [37], a virtual time protocol is used to implement a transmission policy suitable for distributed simulation. Currently, we are working on generalized virtual time and window protocols in which the transmission policy is parameterized and hence can be adjusted to suit a given network application.

References

[1] N. Abramson, "The ALOHA System — Another Alternative for Computer Communications", *AFIPS Proceedings of Fall Joint Compu.*, Vol. 37, 1970.

[2] J. I. Capetanakis, "Tree Algorithm for Packet Broadcast Channels", *IEEE Transactions on Communications*, Vol. Com-27, No. 10, October 1979.

[3] I. Chlamtac, W. R. Franta, and D. Levin, "BRAM: The Broadcast Recognizing Access Method", *IEEE Transactions on Communications*, Vol. COM-27, No. 8, August 1979.

[4] G. Fayolle, P. Flajolet, M. Hofri, and P. Jacquet, "Analysis of a Stack Algorithm for Random Multiple-Access Communication", *IEEE Transaction on Information Theory*, Vol. IT-31, No. 2, March 1985.

[5] A. Grami, K. Sochraby, and J. Hayes, "Further Results on Probing", *Proceedings of the IEEE International Communications Conference*, 1982.

[6] L. W. Hansen and M. Schwartz, "An Assigned-Slot Listen-Before-Transmission Protocol for a Multiaccess Data Channel", *IEEE Transactions on Communications*, Vol. COM-27, No. 6, June 1979.

[7] J. F. Hayes, "An Adaptive Technique for Local Distribution", *IEEE Transactions on Communications*, Vol. Com-26, No. 8, August 1978.

[8] J. Y. Juang and B. W. Wah, "Unifies Window Protocols for Contention Resolution in Local Multi-Access Networks", *Proceedings of the Third Annual Joint Conference of the IEEE Computer and Communications Societies*, April 1984.

[9] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", *Proceedings of IEEE Real-Time Systems Symposium*, December 1985.

[10] L. Kleinrock, "Packet Switching in Radio Channels: Part 1 — Carrier Sense Multiple-Access Modes and their Throughput-Delay Characteristics", *IEEE Transactions on Communications*, Vol. COM-23, No. 12, December 1975.

[11] L. Kleinrock and Y. Yemini, "An Optimal Adaptive Scheme for Multiple Access Broadcast Communication", *Proc. ICC*, 1978.

[12] L. Kleinrock and M. O. Scholl, "Packet Switching in Radio Channels: New Conflict-Free Multiple Access Schemes", *IEEE Transactions on Communications*, Vol. Com-28, No. 7, July 1980.

[13] J. F. Kurose, M. Schwartz, and T. Yemini, "Controlling Window Protocols for Time-Constrained Communication in a Multiple Access Environment", *Proceeding of the 8th IEEE International Data Communication Symposium*, 1983.

[14] J.F. Kurose, "Time-Constrained Communication in Multiple Accesses Networks", *Ph.D. Dissertation, Columbia University*, 1984.

[15] J.S. Meditch and D. H. Yin, "Performance Analysis of Virtual Time CSMA", *Proceedings of IEEE Infocom'86*, April 1986.

[16] J. Misra, "Distributed Discrete-Event Simulation", *Computing Surveys*, Vol. 18, No. 1, March 1986.

[17] A.K. Mok, and M.L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment",*Proc. of the Seventh Texas Conference on Computing Systems*, November 1978.

[18] A.K. Mok and S. A. Ward, "Distributed Broadcast Channel Access", *Comput. Network*, Vol. 3, November 1979.

[19] A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", *Ph.D. Dissertation*, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1983.

[20] M.L. Molle, "Unifications and Extensions of the Multiple Access Communications Problem" *Ph.D. Dissertation*, University of California at Los Angeles, July 1981.

[21] M.L. Molle and Lenonard Kleinrock, "Virtual Time CSMA: Why Two Clocks are Better than One", *IEEE transactions on Communications*, Vol. COM-33, No. 9, September 1985.

[22] S. Panwar, D. Towsley, and J. Wolf, "Optimal Scheduling Policies for a Class of Queues with Customer Deadlines to the Beginning of Service" *submitted for publication.*

[23] S. Panwar "Time Constrained and Multiaccess Communications" *Ph.D. Dissertation*, University of Massachusetts at Amherst, February 1986.

[24] K. Ramamritham, J.A. Stankovic, and W. Zhao, "Distributed Scheduling of Hard Real-Time Tasks under Resource Constraints in the Spring System", submitted for publication.

[25] K. Ramamritham, "Channel Characteristics in Local Area Hard Real-Time Systems", *Comput. Networks and ISDN Syst.*, to be published.

[26] E. H. Rothauser and D. Wild, "MLMA — A Collision-Free Multi-Access Method", *Proc. IFIP Congr. 77*, 1977

[27] M. Scholl, "Multiplexing Techniques for Data Transmission over Packet Switched Radio Systems", *Ph. D. Dissertation*, University of California at Los Angeles, 1976.

[28] J.A. Stankovic, "A Perspective on Distributed Computer Systems", *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984.

[29] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc., 1981.

[30] D. Towsley and G. Venkatesh, "Window Random Access Protocols for Local Computer Networks", *IEEE Transactions on Computers*, Vol. C-31, No. 8, August 1982.

[31] B. W. Wah and J. Y. Juang, "An Efficient Protocol for Load Balancing on CSMA/CD Networks", *Proceedings of the Eighth Conference on Local Computer Networks*, Oct. 1983.

[32] Y. Yemini, "On the Channel Sharing in Discrete-Time Packet Switched, Multiaccess Broadcast Communication", *Ph.D. Dissertation*, University of California at Los Angeles, 1978.

[33] W. Zhao, "A Heuristic Approach to Scheduling Hard Real-Time Tasks with Resource Requirements in Distributed Systems", *Ph.D. Dissertation*, The University of Massachusetts at Amherst, 1986.

[34] W. Zhao, and K. Ramamritham "A Virtual Time CSMA Protocol for Hard Real-Time Communication", *Proceedings of IEEE Real-Time Systems Symposium,* 1986.

[35] W. Zhao, and K. Ramamritham "Virtual Time CSMA Protocols for Hard Real-Time Communication", *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 8, August 1987.

[36] W. Zhao, et al "Performance Evaluation of a Window Protocol for Hard Real-Time Communication", *Tech Report, Department of Computer and Information Science, University of Massachusetts*, August 1987.

[37] W. Zhao,"Using the Continuous Simulation Clock to Control the Discrete Distributed Simulation", *21st Simulation Conference*, March 1988.

[38] T. Znati and L. Ni, "A Prioritized Multiaccess Protocol for Distributed Real-Time Applications", *Proceedings of IEEE 7th International Conference on Distributed Computing Systems*, September 1987.

Name	Operation
Reset(stack)	Make the stack empty.
Clean(stack)	Pop out and discard any item (u, i) if u < t.
Empty(stack)	Return true if the stack is empty.
Top_u(stack)	Return the u value of the top item in the stack if any.
Top_i(stack)	Return the i value of the top item in the stack if any.
Push(stack, (u, i))	Push (u, i) onto the stack.
Pop_u(stack)	Pop the top item of the stack and return the value of u.
Pop i(stack)	Pop the top item of the stack and return the value of i.

Table 1: Procedures and Functions for Stack Used in the Extended Protocol

```
Initialization:

    up := t + δ
    empty(stack); (* make the stack empty *)
    send out the first message if its LS is between [t, up);

At the beginning of each time unit:

    drop any message M if D_M - L_M < t;
    clean(stack); (* pop out and discard any item u if u < t
    *)
    get_state(state);
    CASE state OF
    channel_busy_due_to_a_collision:

        Abort the message transmission if any;

    channel_idle_after_a_collision:

        Contract_Window_and_Send(up, t);

    channel_busy_due_to_a_message_transmission:

        Continue the transmission if any;

    channel_idle_after_a_successful_transmission:

        Pop_and_Send(up, t);

    channel_continue_idle:

        Expand_Window_and_Send(up, t);

    END (* CASE *)
```

Figure 1: The Time-Constrained Window Protocol

```
Procedure Contract_Window_and_Send(up, t);

BEGIN
      IF up > t
      THEN
      BEGIN
         push(stack, up);
         up = t + ⌈ (up - t)/2 ⌉; (* reduce up *)
         send out the message if its LS is in [t, up)
      END
      ELSE (* up ≤ t *)
         Pop_and_Send(up, t); (* See Figure 3 *)
END;
```

Figure 2: Procedure Contract_Window_and_Send

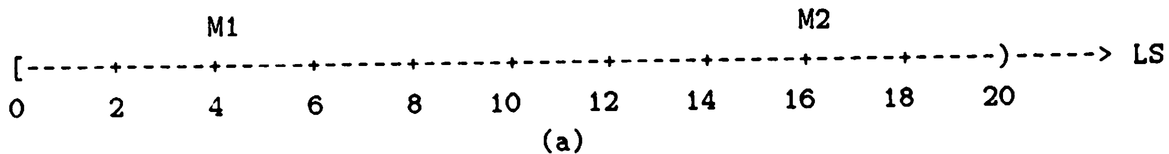
```
Procedure Pop_and_Send(up, t);
BEGIN

    IF not empty(stack)
    THEN

        up = pop(stack)

    ELSE

        up = max(up, t) + δ;

    send out the first message in the queue if its LS is in
    [t, up);

END;
```

Figure 3: Procedure Pop and Send

```
Procedure Expand_Window_and_Send(up, t);

BEGIN
   IF empty(stack)
   THEN
   BEGIN
      up = t + δ;
      send out the first message if its LS is in [t, up);
   END
   ELSE
      IF up < top(stack) - 1
      THEN
      BEGIN
         up = ⌈ (up + top(stack))/2 ⌉;
         send out the first message if its LS is in [t, up)
      END
      ELSE  (* the old up = top(stack) - 1 *)
      BEGIN
         up = pop(stack);
         send out the first message if its LS is in [t, up)
      END
END;
```

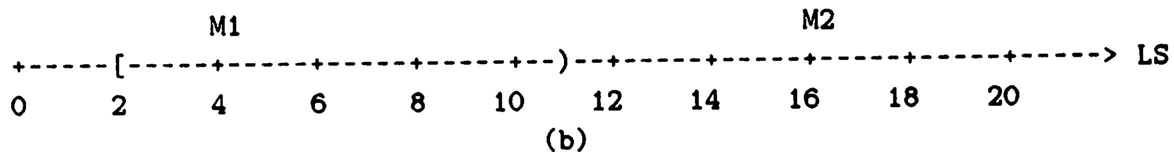
Figure 4: Procedure Expand_Window_and_Send

t = 0: The channel is idle. The initial window = [0, 20).
M1 and M2 both start to be sent out, causing a collision.



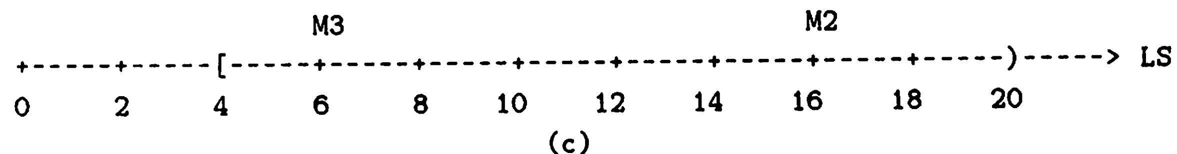
(a)

t = 2: The channel is idle after the collision. The window is reduced. The
new window = [2, 11). top(stack) = 20. M1 is to be transmitted.



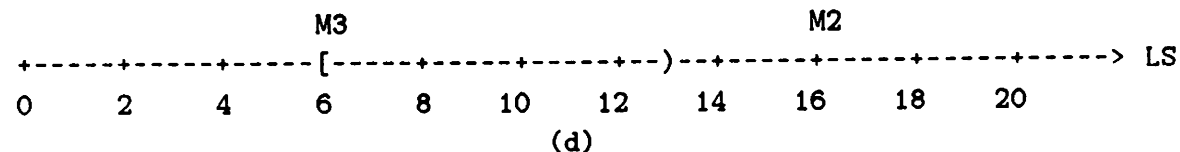
(b)

t = 3: M3 arrives at the system. Transmission of M1 completes.
t = 4: The channel is idle. Protocol goes back to the previous window
[4, 20). Both M3 and M2 are to be transmitted, causing a collision.



(c)

t = 5: The collision is detected. The transmissions of M3 and M2 abort.
t = 6: The channel becomes idle. The window size is reduced.
The new window = [6, 13). M3 is to be transmitted.

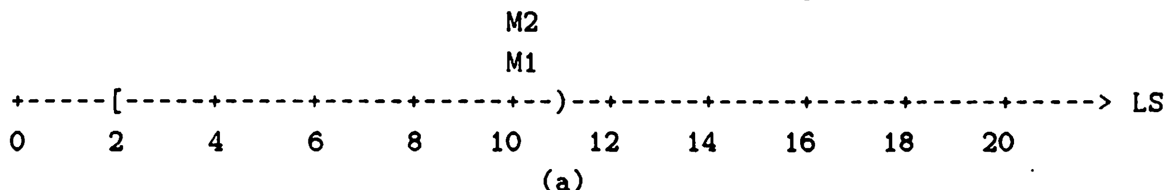


(d)

After the transmission of M3, M2 will have a chance to be transmitted.

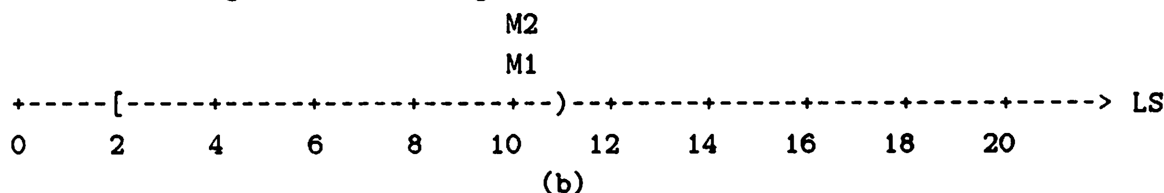
Figure 5: Example 2 — Using the New Window Protocol

t = 0: The channel is idle. The initial window = [0, 20).
M1 and M2 both start transmitting, causing a collision.



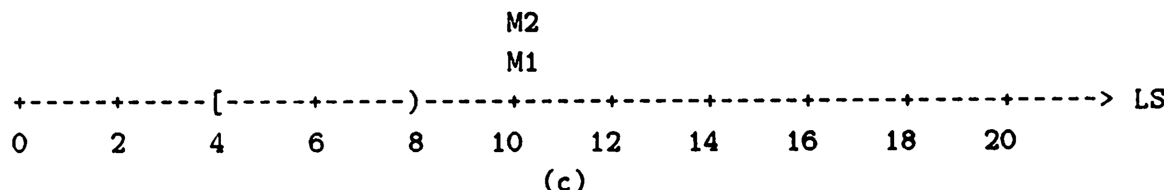
(a)

t = 2: The channel is idle after the collision. top_u(stack) = 20.
The new window = [2, 11). M1 and M2 are transmitted,
causing a collision again.



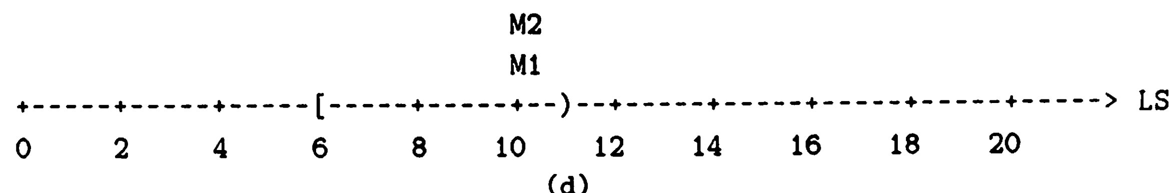
(b)

t = 3: The collision is detected. The transmission of M1 and M2 abort.
t = 4: The channel is idle. The new window = [4, 8). top_u(stack) = 11.



(c)

t = 5: The channel is continues to be idle. The window expands.
The new window = [5, 10). top_u(stack) = 11.
t = 6: The channel is continues to be idle. If the protocol in
Section 3 is used the window expands. Then both M1 and M2 are
in the window [6, 11). When both are sent, a collision occurs.



(d)

Then, the window will be reduced, expanded, repeating the above
steps until the time passes 10 --- after which both M1 and M2 are
lost!!

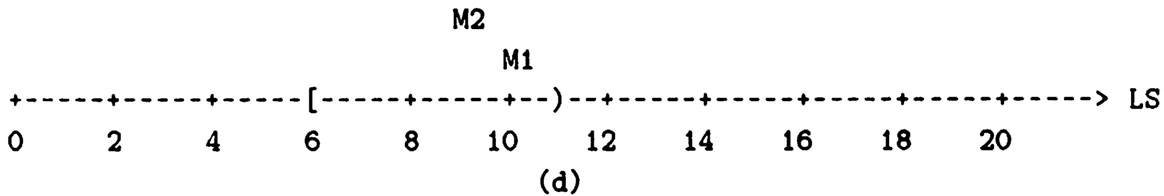
Figure 6: Example 3 — A Message Laxity Tie

```
Procedure Expand_Window_and_Send(up, t);

BEGIN
  IF empty(stack)
  THEN
  BEGIN
     up = t + δ;
     send out the first message if its LS is in [t, up);
  END
  ELSE
     IF up < top_u(stack) - 1
     THEN
     BEGIN
        up = ⌈ (up + top u(stack))/2 ⌉;
        send out the first message if its LS is in [t, up)
     END
     ELSE  (* a tie on message's LS *)
     BEGIN (* use a random scheme to make a decision *)
        IF message M with I_M = top_i(stack) is in the queue
        THEN
           IF Random(0, 1) > P
              THEN
                 send out message M again
              ELSE
                 LS_M = Random(t+2, D_M - L_M) (* Not to be sent !  *)
        up = pop_u(stack);
     END;
END;
```

Figure 7: Extended Procedure Expand_Window_and_Send

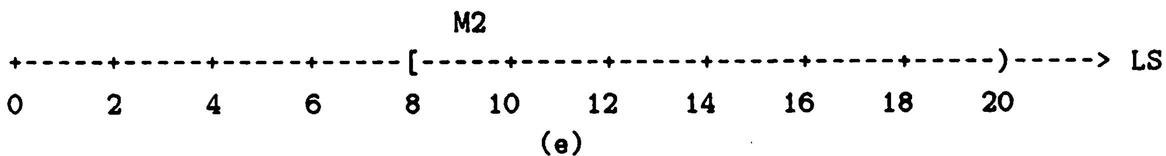
Assume the extended protocol is used. Example 3 continues
from Figure 8.d --- {\tt t }= 6.

t = 6: The channel is continuing idle. Because the old value of up
is equal to 10 = top_u(stack) - 1, the probabilistic scheme is used.
Assume randomly it is decided that M1 is to be sent and M2's LS is
modified to 9. The top value of the stack is popped, the new window
is [6, 11). Protocol explicitly prohibits the transmission
of M2 at this time although it is in the window.



(d)

t = 8: The transmission of M1 has completed. The old window = [8, 11).
top_u(stack) = 20. After popping the stack, the new window is
[8, 20). M2 is considered for transmission.

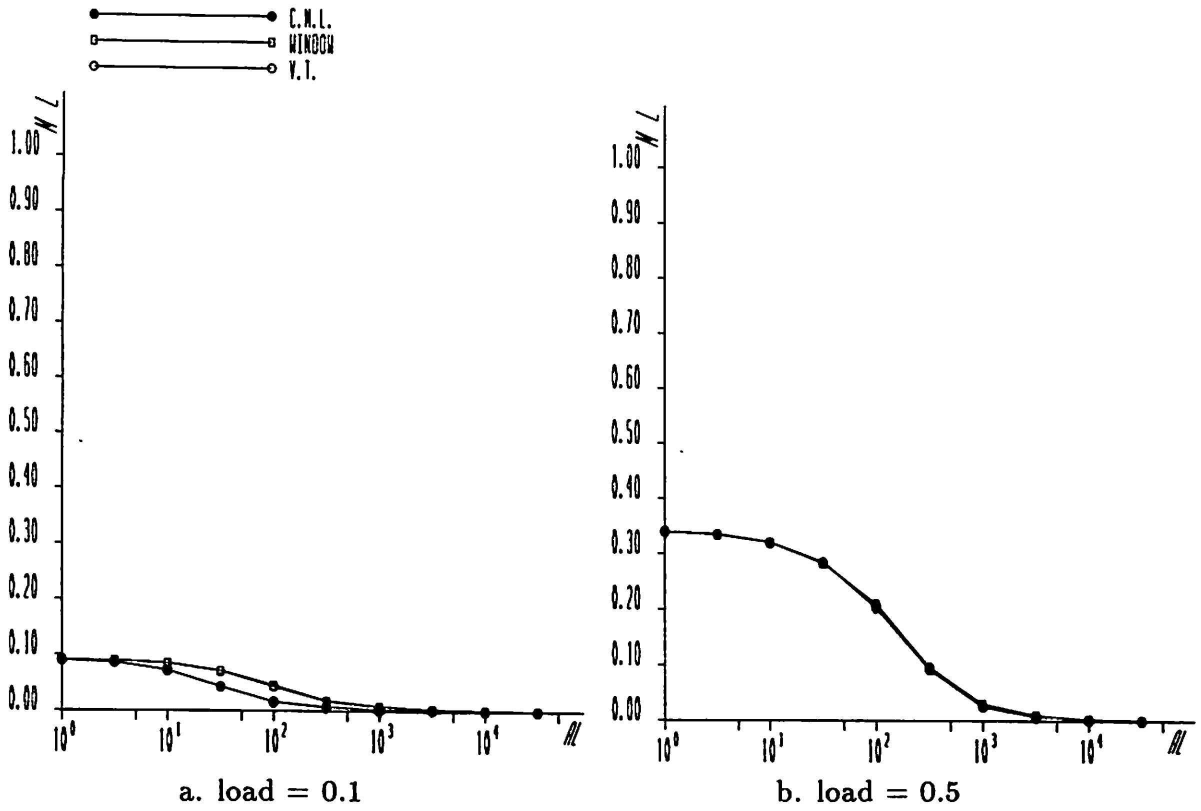


(e)

Figure 8: Example 3 — Using the Extended Protocol

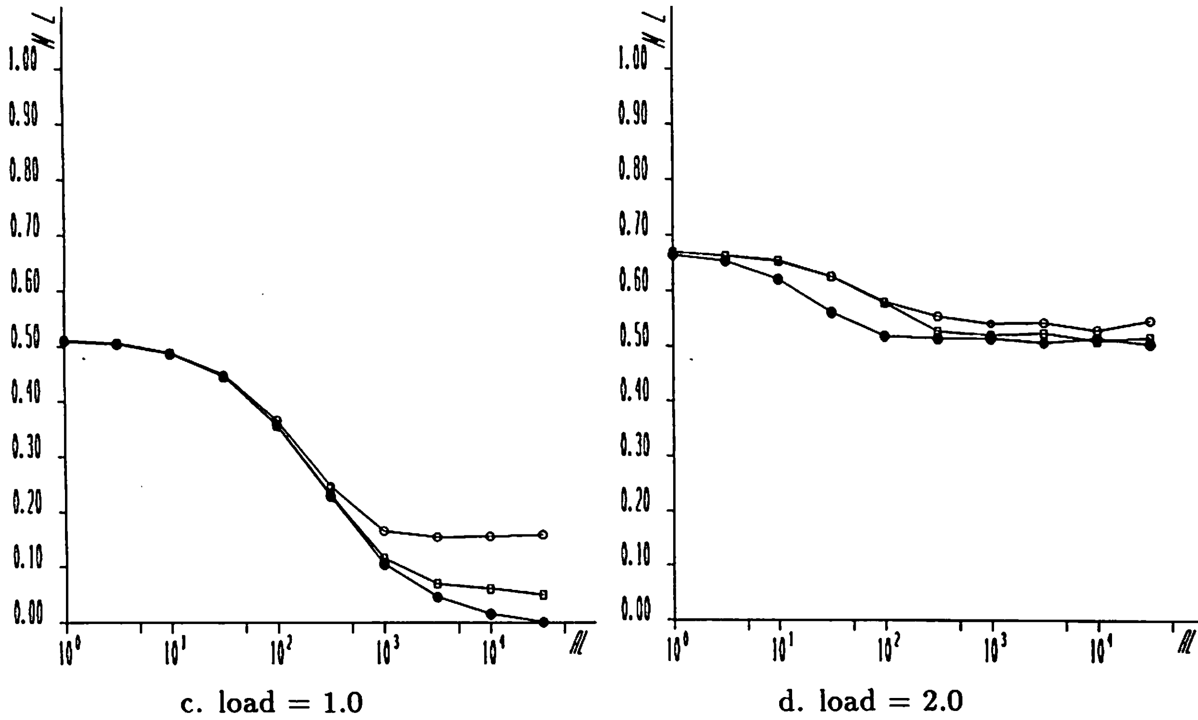
```
Procedure Contract Window and Send(up, t);

BEGIN
01      IF up > t
02      THEN
03         IF up > t + 1
04         THEN
05         BEGIN
06            IF this node is involved in the collision
07            THEN
08               i = I_M (* M is the collided message *)
09            ELSE
10               i = 0;
11            push(stack, (up, i));
12            up = t + ⌈ (up - t)/2 ⌉; (* reduce up *)
13            send out the message if its LS is in [t, up)
14         END
15         ELSE  (* up = t + 1 *)
16         BEGIN (* use the random scheme to make a decision*)
17            IF message M with I_M = top i(stack) is in the queue
18            THEN
19               IF Random(0, 1) > P
20               THEN
21                  Send message M again
22               ELSE
23                  IF LA_M = 0
24                  THEN
25                     discard message M
26                  ELSE
27                     LS_M = Random(t+2, D_M - L_M)
28         END
29      ELSE (* up ≤ t *)
30         Pop_and_Send(up, t); (* See Figure 3 *)
END;
```

Figure 9: Extended Procedure Contract_Window_and_Send



a. load = 0.1

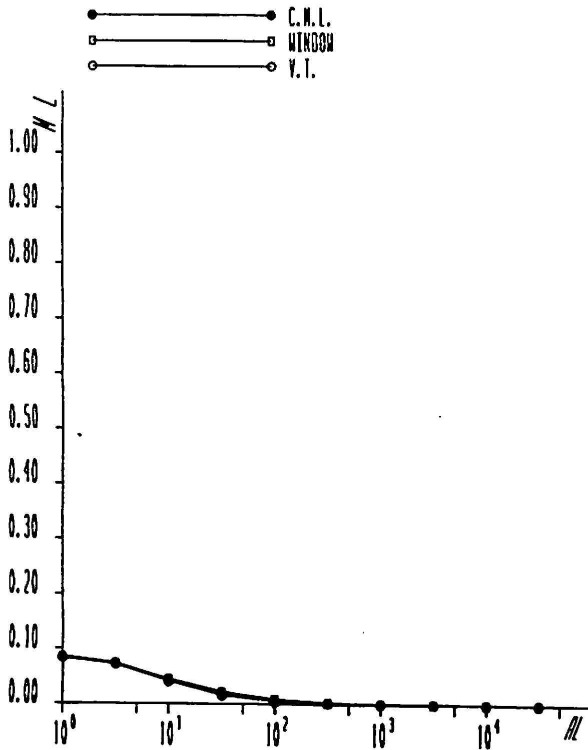
b. load = 0.5



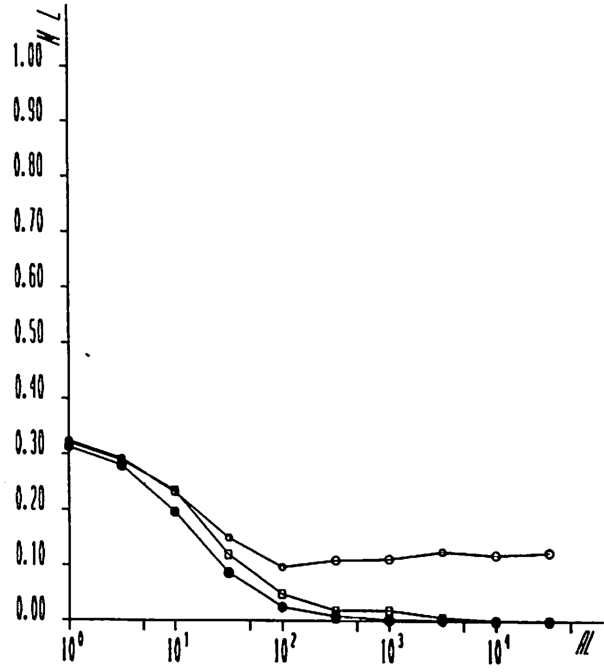
c. load = 1.0

d. load = 2.0

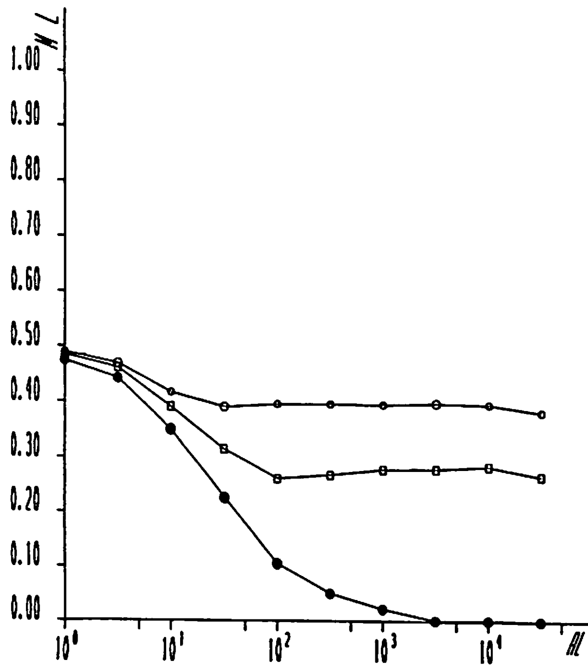
Figure 10: Effect of Environment Parameters I — $\alpha = 0.01$



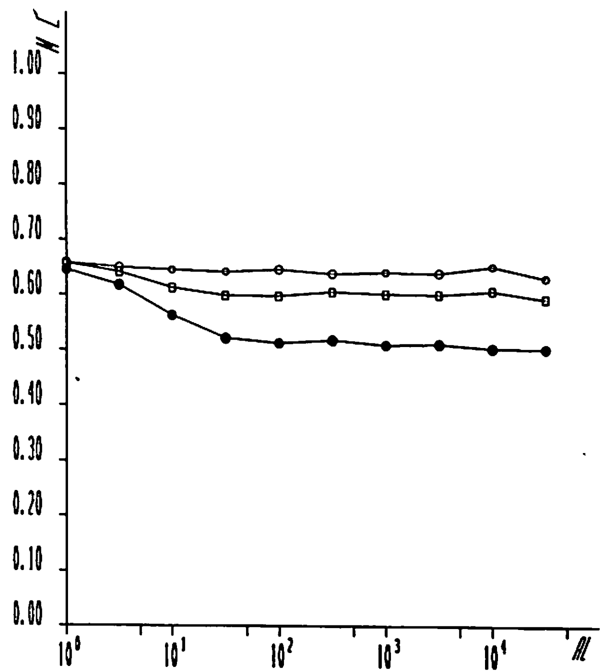
a. load = 0.1



b. load = 0.5

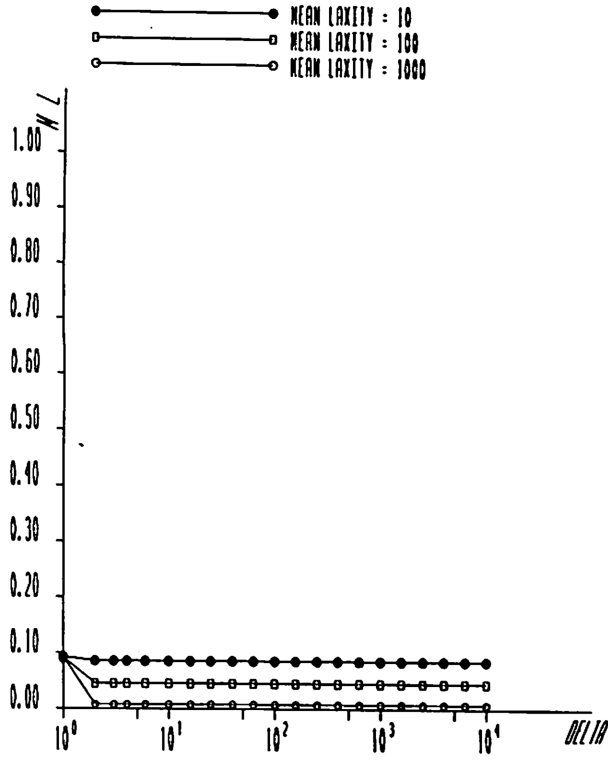


c. load = 1.0

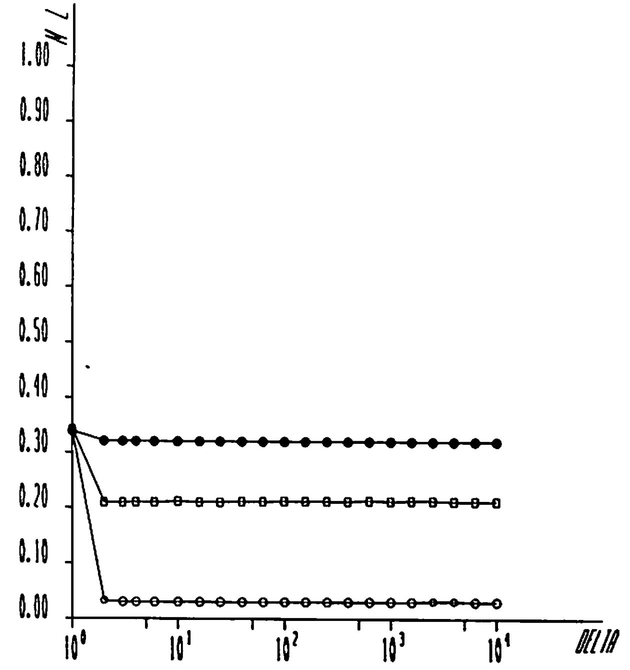


d. load = 2.0

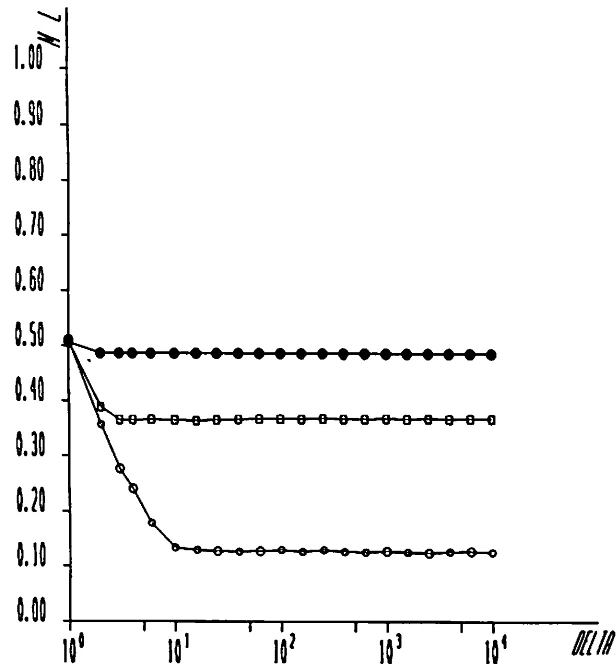
Figure 11: Effect of Environment Parameters II — $\alpha = 0.10$



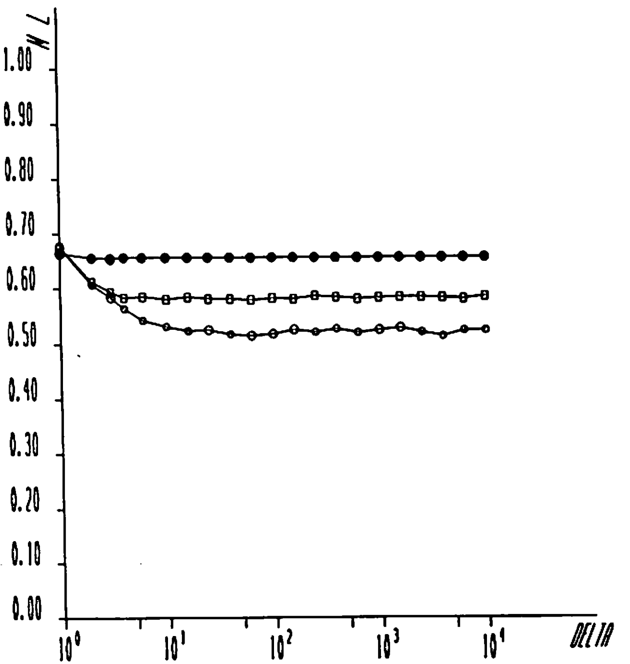
a. load = 0.1



b. load = 0.5

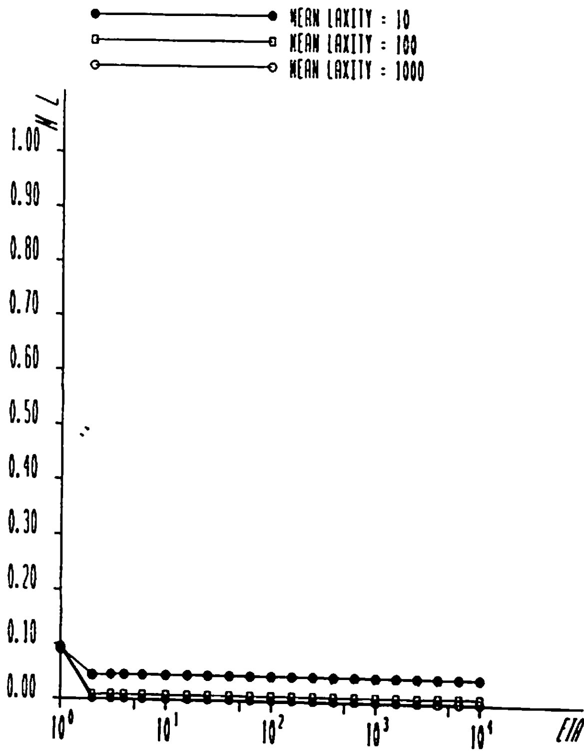


c. load = 1.0

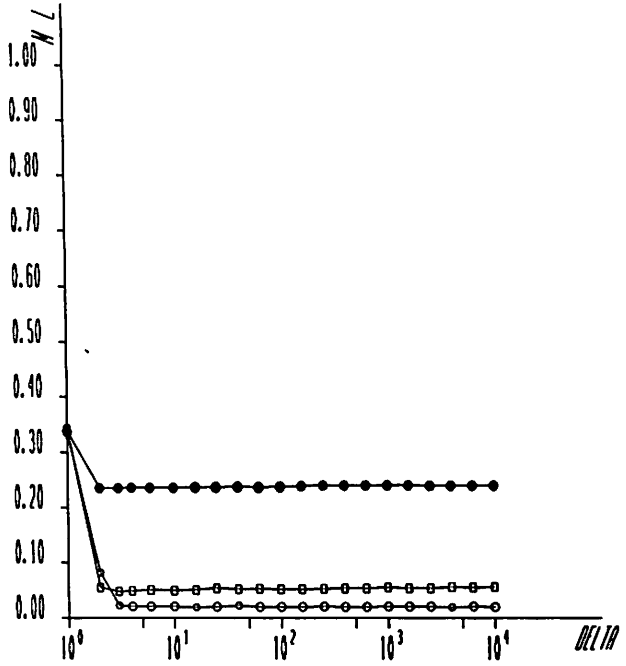


d. load = 2.0

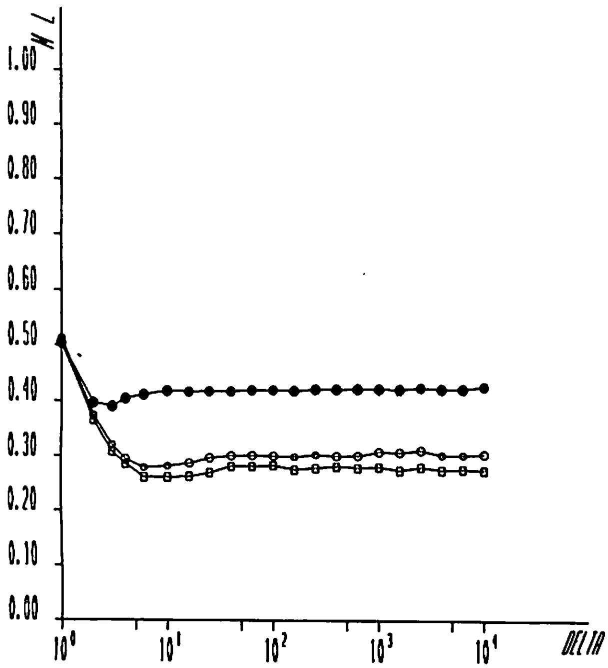
Figure 12: Sensitivity of δ for Window Protocol I — $\alpha = 0.01$



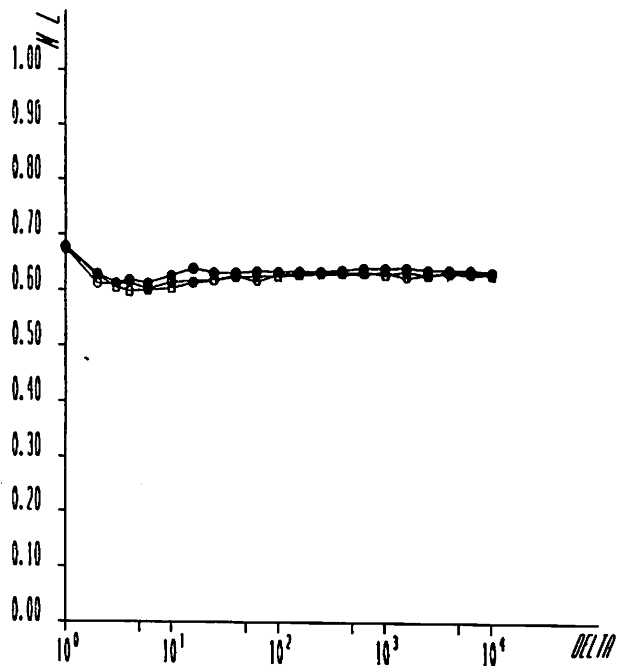
a. load = 0.1



b. load = 0.5

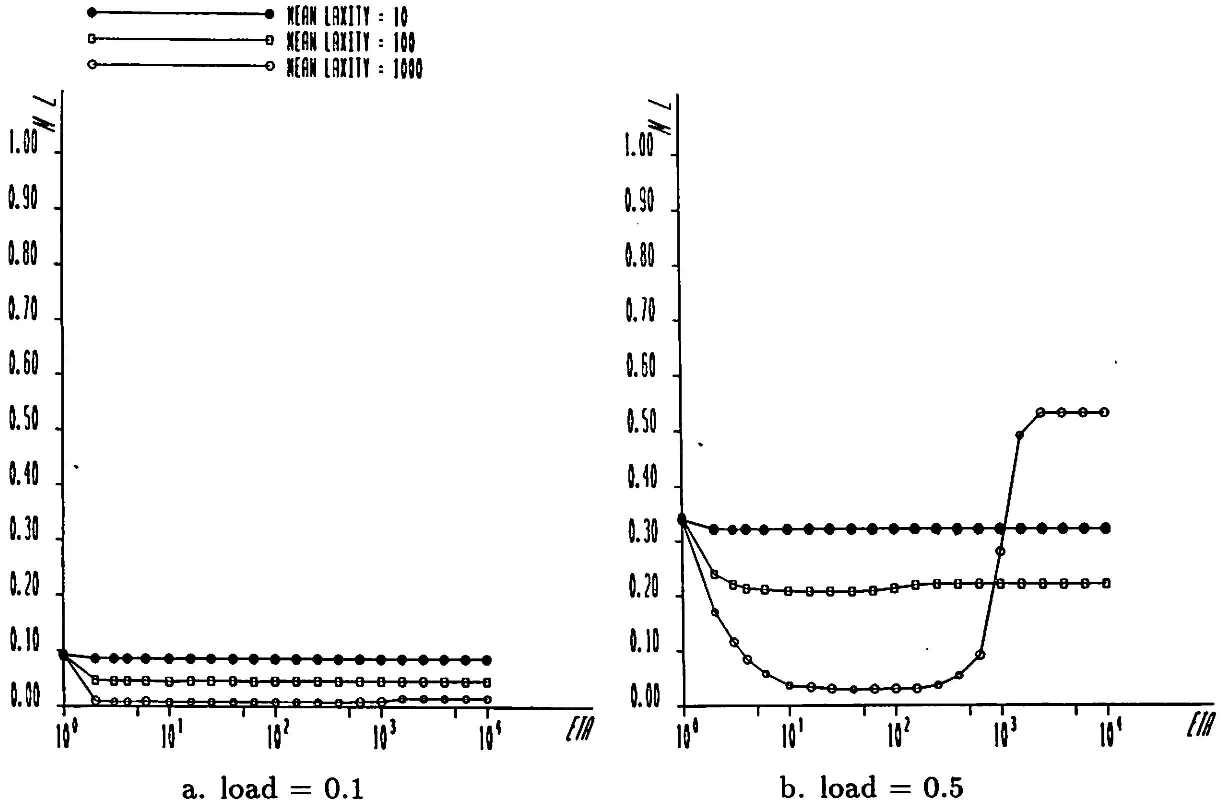


c. load = 1.0



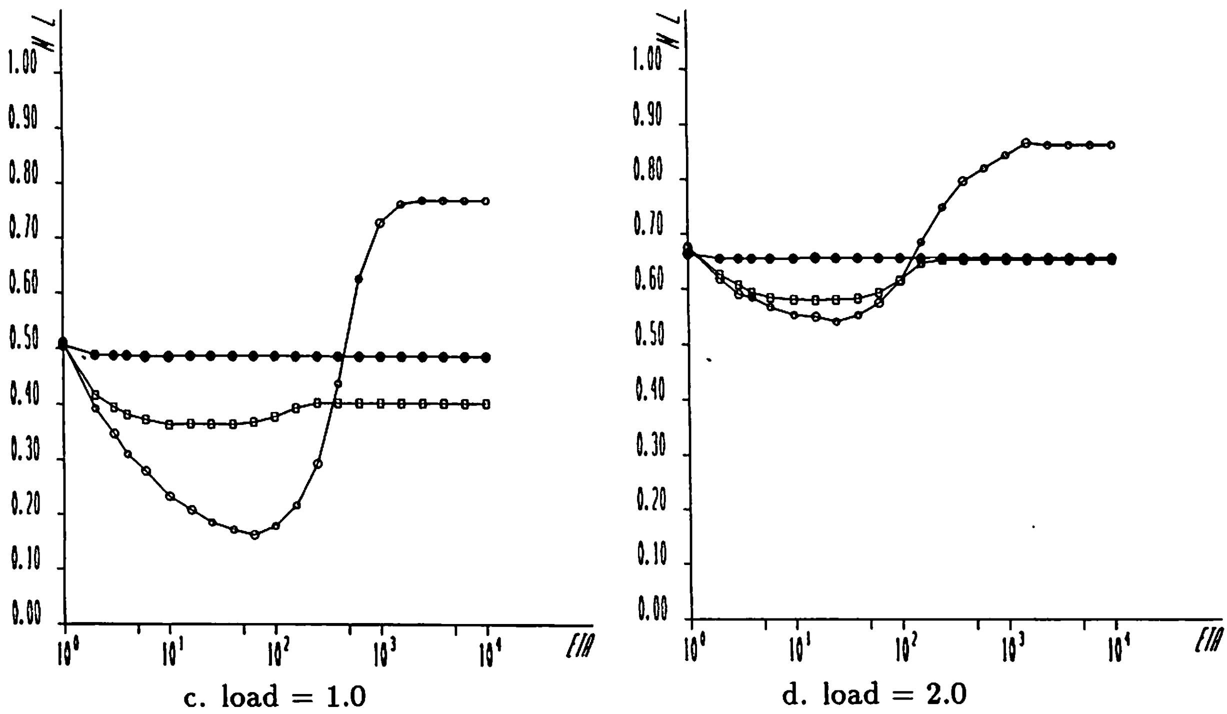
d. load = 2.0

Figure 13: Sensitivity of δ for Window Protocol II — $\alpha = 0.10$



a. load = 0.1

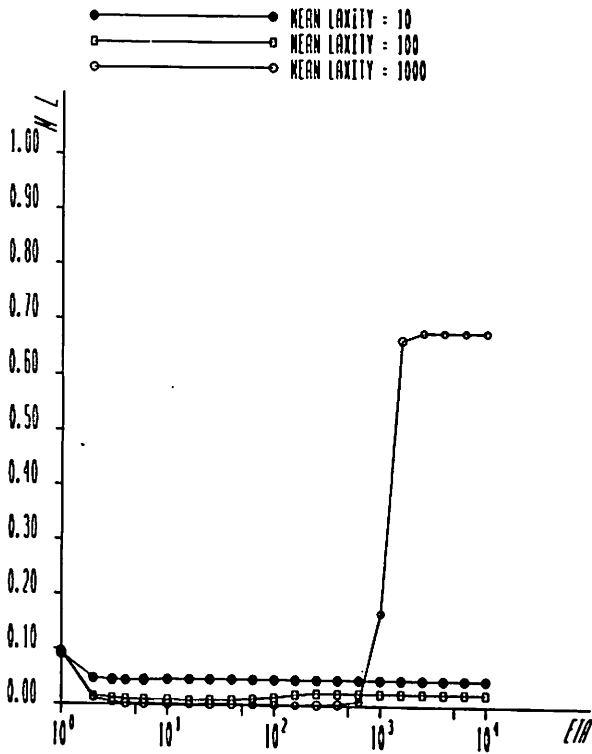
b. load = 0.5



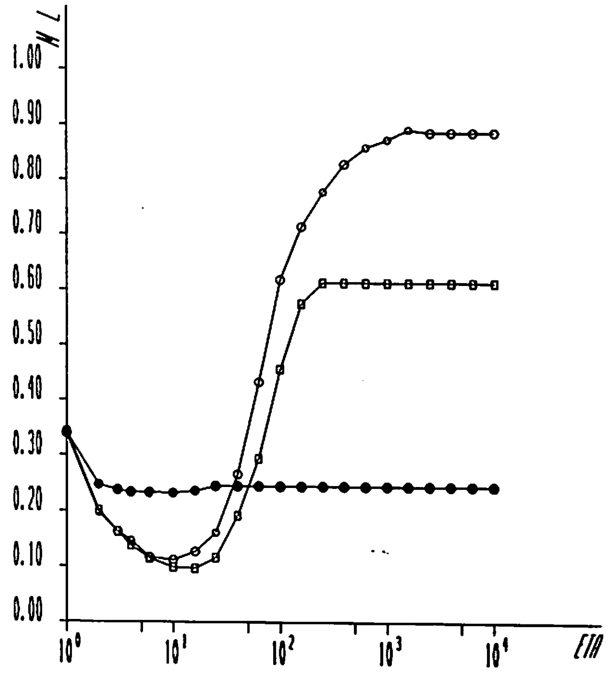
c. load = 1.0

d. load = 2.0

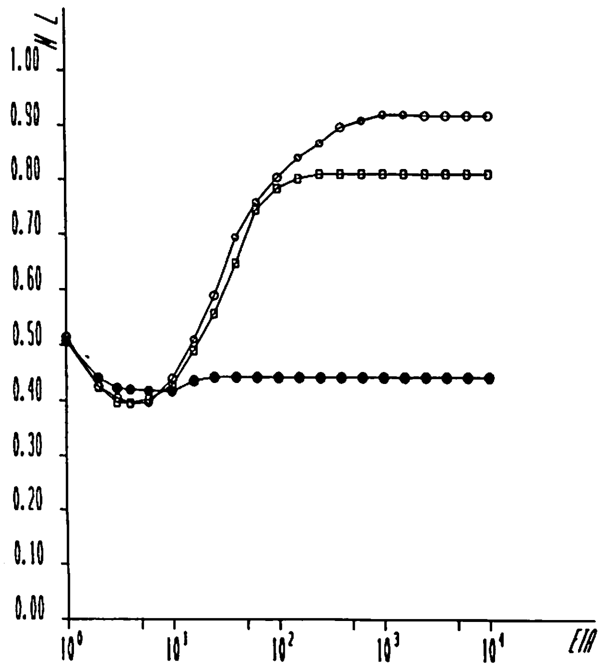
Figure 14: Sensitivity of η for Virtual Time Protocol I — $\alpha = 0.01$



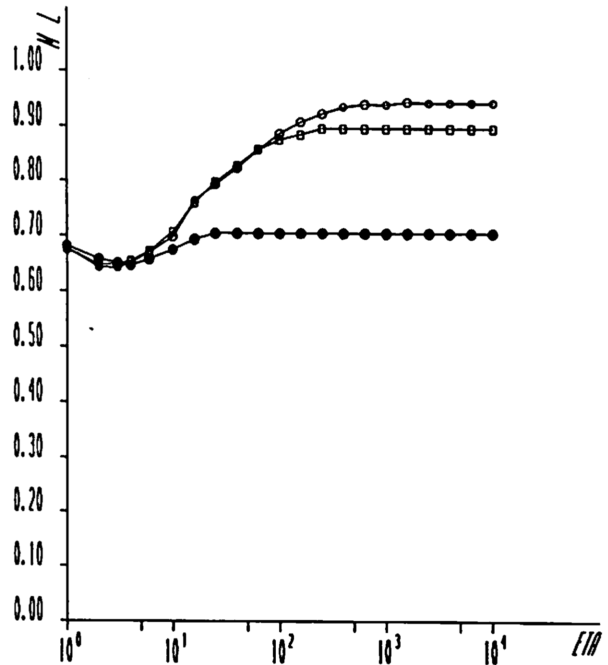
a. load = 0.1



b. load = 0.5



c. load = 1.0



d. load = 2.0

Figure 15: Sensitivity of η for Virtual Time Protocol II — $\alpha = 0.10$