1. Chapter I

CHAPTER I

Introduction

1.1 Background and Motivations

Multimedia applications require a high network bandwidth and also impose stringent time constraints on the network, such as constraints on the maximum tolerable delay and the variance of the delay. This thesis focuses on network and operating system support for multimedia systems.

Because of recent advances in network technology, high-speed networks are becoming widely available and high-speed processors have become cheaper. Applications such as high volume transaction processing, high-speed file transfers and multimedia are becoming viable.

Multimedia Systems

In multimedia systems, multiple media types such as text, audio and video, are stored, processed and transmitted to applications. Each media type represents a separate *data stream* and each data stream can have multiple sinks for each source. Because there are

multiple data streams and the data streams have requirements for timing delays, they are real-time in nature.

Multimedia applications encompass a diverse group of programs, ranging from information kiosks and interactive demos to interactive video, video on demand, and teleconferencing.

One characteristic common to all multimedia applications is that the multimedia data, which includes voice, motion video, high-quality sound, still images and text, is real-time in nature. Since the data typically represents a continuous stream of either voice or video, it must be delivered to its destinations in a timely fashion. If the data is delayed too long, then it can not be used and must be discarded. Each class of media, such as voice or video, has specific tolerances for its deadlines. If audio data is not delivered by its deadline, the missing data manifests itself as an audible "pop." Similarly, missing video data manifests itself as a noticeable discontinuity. Obviously both of these situations should be avoided. Multimedia data has soft-deadlines because dropped packets result in an undesirable situation such as pops and clicks which represent a quality degradation but pose no threat to lives or property. On the other hand, hard-deadline real-time systems can not tolerate a single missed deadline, because they are often used for control systems in which a missed deadline can have disastrous results.

Even though soft-deadline systems are not as restrictive as hard-deadline systems, they still place a demand on the network to meet their constraints. Multimedia data requires that the network provides high bandwidth, low delay, and low variance in the delay.

High Speed LANs

Multimedia data imposes serious constraints on networks in terms of bandwidth, delays and delay variance tolerance. Ethernet is a popular local area network that is currently widely used for networking workstations together and typically have a maximum bandwidth of 10 megabits per second (Mb/s). Ethernets use a CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol for transmission. This is a decentralized protocol that is appropriate for traditional non-real-time data, which is bursty and can tolerate a high variance in delay. However, it is not adequate for high bandwidth, time-sensitive data such as multimedia. If multimedia services are implemented on an Ethernet, then multimedia data must compete with traditional non-real-time data for bandwidth which often results in insufficient bandwidth or excessive delay or delay variance for the multimedia data. Because of the large installed base of Ethernet, it is cost-effective to modify the protocol to allow it to provide adequate support for multimedia data. This thesis presents three protocols to allow multimedia data to be transmitted over an Ethernet.

Asynchronous Transfer Mode (ATM) is a new high speed network that is getting popular because it is designed to provide a high-speed link between nodes and provide a

guaranteed level of quality of service (QOS), such as bandwidth, delay, and delay variance. It is meant to work with both traditional, non-real-time data as well as real-time multimedia data, and can serve as both a local area network (LAN) and a wide area network (WAN).

Causal Order

In a teleconference, one person speaks which provokes a response from another speaker, and so on. The bursts of conversation have a cause and effect relation. The speakers in a conference-call typically are geographically distant, which introduces variable delays as messages are sent through these networks. With complex multistage network topologies, it is possible for messages to get out of order, due to non-uniform delays at different sites. Because of these variable network delays, message delivery is not guaranteed to be in casual order. We need to enforce a message order so when the conversation bursts are combined, the result is in causal order. Therefore causal ordering of multimedia messages is an important issue in applications like conference calls. In order to maintain causal order, each node must maintain some state information about every other node in the system. In order to keep the state information current, it must be transferred with each message which imposes an overhead for every message sent. This thesis presents a method of ensuring causal order for multimedia data while minimizing the associated message overhead.

Causal Multicast

In a conference-call, typically one speaker speaks while other participants listen. This requires a multicast capability from the underlying network. A *multicast* is a transmission in which there is one sender and multiple recipients. In a teleconference, multimedia data, which consists of multiple data streams, such as text, audio, and video from multiple sources, is multicast to the recipients. Since in a teleconference, the cause and effect relations between messages should be preserved, causal order must be maintained in a multicast environment. *Causal multicasting* is defined as maintaining causal order relations in a multicasting environment and presents additional challenges beyond those associated with just maintaining causal ordering. Not only must each node enforce causal order, but multiple nodes must identically order their data streams while obeying the real-time constraints. Every causal order algorithm imposes a message overhead due to the control information that must be sent with each message to insure causal order delivery. One of the biggest challenges in causal multicasting is the efficient use of bandwidth because the impact of the overhead due to control information increases with the number of recipients. Because of these characteristics, ATM can meet the demands of multimedia data. Existing causal multicast algorithms express the cost of multicasting by associating a weight with each edge and using that to determine the optimal graph for multicasting. We show that other criterion, such as the amount and type of data to be sent, should be considered in order to determine how optimum a graph is for multicasting. The impact a multicast message has on total network bandwidth

depends on the configuration of the network as well as the data being sent and is studied in this thesis. Additionally, an efficient, low cost algorithm to perform causal multicasting is presented.

1.2 Problem Statement and Contributions

This dissertation discusses methods of supporting the transmission of multimedia data over high-speed networks, both at the network and operating system level. Three primary areas are investigated. The first is Priority Ethernet, which involves supporting real-time and traditional data types on Ethernets. The second involves supporting causal order for multimedia data in a unicasting environment. The third discusses support for causal order in a multicasting environment. These are briefly described below.

1.2.1 Priority Ethernet

In traditional Ethernet, CSMA/CD is used to control access. There is no notion of priority, since all sites compete to transmit their data. Ethernet can not support high speed traffic because as the load increases the throughput decreases due to excessive collisions. As a result, it is poorly suited for multimedia data, since real-time data competes equally with non-real-time data for network access.

There are two alternatives available. The first is to migrate to a network technology that can provide high-speed, high bandwidth, and low delay, like ATM or B-ISDN. The second is to adapt existing Ethernet to allow real-time data to coexist with non-real-time

data such that the real-time data is transmitted without any interference from or reduction in bandwidth due to the non-real-time data. Because of the large installed based of Ethernet and since new technology like ATM is not yet fully evolved and initially expensive, we examine methods to support priority transmission on an Ethernet.

To that end, we propose three algorithms for *Priority Ethernet*, an Ethernet that supports multiple classes of priority. The first, called the Synchronous Method, eliminates virtually all collisions by having a well-defined order for data transmission that is established during the *priority information exchange phase* in which nodes have a fixed *a priori* order to transmit their priority data. The second, called the Modified Synchronous Method, is similar to the Synchronous Method, but it relaxes the ordering constraint during the priority exchange phase in an attempt to improve efficiency. Finally, the third, called the Asynchronous Method, eliminates the priority information exchange phase to reduce the time high priority message wait until they can be transmitted. We describe the algorithms, and present simulation data on the performance of the methods and discuss the conditions under which each method performs best.

1.2.2 Real-Time Causal Order for Multimedia

In applications such as a conference call, typically a message from one "speaker" causes another message to be sent from another speaker. This cause and effect order should be preserved upon message delivery. A *delta causal order* relation is defined as a causal order relation restricted to some Δ time interval. Delta causal order is useful to describe

causal order relations for real-time messages with deadlines since the causal order relation should only apply while the message has not passed its deadline.

We describe one type of causal order violation called the *triangle inequality* and argue that this simple version of causal order violation is responsible for the majority of causal order violations. To that end, we propose an algorithm to insure that messages are delivered in delta causal order. The algorithm enforces causal order by adding two vectors, from a matrix maintained at each site, to every message. Element (*i, j*) of the matrix records the last known timestamp when site *i* sent a message to site *j*. The two vectors added to each message are row *i* and column *j* of the dependency matrix at site *i*. By comparing the vectors in an arriving message with the corresponding vectors in the destination site's dependency matrix, missing messages are detected; they are represented by timestamps in the message's vectors that are greater that those in the site's dependency matrix. Real-time values are used for timestamps in order to detect if a message has exceeded its deadline. The message overhead is analyzed and a proof of correctness is presented.

1.2.3 Causal Multicast

The previous discussion on causal order assumed that messages are unicast to other sites, i.e., each message has only one destination. Sending a message to multiple destinations requires separate unicast messages to be sent. The concept of a *multicast* allows a source to send a message to multiple destinations in one message. Applications

like conference calls require multicasting capability from the underlying network, since there is one speaker and multiple listeners. Ensuring causal order message delivery in a multicast environment is more difficult than in a unicast environment. The amount of control information relating to the recipients increases with the number of recipients, thus increasing the total size of messages and increasing the total amount of bandwidth consumed which decreased the bandwidth available for the data. The simplest way to implement a multicast message is to simulate it as a series of unicast messages, in which each unicast is sent to a separate destination of the multicast message. While this allows existing algorithms to work, it is an inefficient and cumbersome way to implement a multicast. Multicast capability allows one message to be sent to many destinations simultaneously and consumes less total network bandwidth than the equivalent implementation using multiple unicasts. There are two areas we examine, with respect to causal multicasts. The first focuses on characterizing the total cost of a given multicast. The second deals with the design of an efficient causal multicast algorithm and we propose a method called the Partition Method. These are briefly discussed below.

Characterizing Multicast Cost

In order to compare the relative efficiency of different causal multicast algorithms, it is necessary to describe the cost of each algorithm. The cost of multicasting a message depends upon both the underlying topology of the communication network and the causal multicast algorithm being used. The network topology determines how much of the path is shared between the recipients of the multicast. The longer the paths that are

shared, the greater the overall multicast efficiency. Since each message has control information appended to it by the causal multicast algorithm, the overall efficiency is also affected by the overhead associated with the algorithm. We propose a framework to describe the cost of a multicast, given the message overhead and the multicast topology. In addition, a measure of the efficiency of the multicast is presented. The efficiency is expressed in comparison to the cost of implementing a multicast as a sequence of unicasts, which can be thought of as the worst-case scenario.

Partition Method

The causal order algorithm that we propose in this dissertation (briefly described in Section 1.2.2) only provides delta causal order but it does not provide direct support for multicasting. We present an algorithm called the Partition Method, which is designed to have a low cost and support multicasting. This algorithm divides the list of destinations into several smaller groups of destinations, or partitions, where each partition has a relatively high degree of path sharing. The efficiency of a multicast within a partition is higher and we can use one multicast for each partition. The causal order algorithm of Section 1.2.2 becomes a special case of the Partition Method in which there is only one partition. The Partition Method algorithm is described and compared to existing causal multicast algorithms. An analysis of the algorithm for several different sample topologies is presented. In addition, the efficiency of the algorithm with respect to the size of the partitions is analyzed.

1.3 Organization of Dissertation

The dissertation is organized as follows: Chapter II presents Priority Ethernets. Current Ethernet technology is described and the issues and objectives involved in a priority protocol are discussed. Three techniques for priority transmission are proposed. A qualitative analysis and simulation results comparing the performance of these methods are presented.

Chapter III focuses on maintaining causal order for multimedia data. A low-cost algorithm that maintains delta causal order by preventing delta causal order violations due to the triangle inequality is presented. A proof of correctness is also presented. Performance issues are discussed, including message overhead and complexity, as well as relaxing constraints on the requirements of clock synchronization.

Chapter IV introduces causal multicast and focuses on how to characterize the cost of a multicast based on the communication network topology and the causal algorithm used. A cost analysis is presented that compares the costs of two existing algorithms.

Chapter V presents a new method for causal multicast called the Partition Method. The cost of the Partition Method is analyzed and compared to the two methods discussed in Chapter IV. The performance is evaluated for several different topologies as well as for partitions that contain different numbers of elements. Chapter VI presents the conclusions and future directions for research.

2. Chapter II

CHAPTER II

Priority Ethernets: Multimedia Support on Local Area Networks

2.1 Introduction

Multimedia is becoming important in several domains [12,13,20,21,34]. By allowing the flexibility of presenting more than one type of information, such as voice, music, still and motion video, in addition to normal text, multimedia computing is entering into office space, businesses, and other professional, non-academic environments.

There are many potential uses of multimedia in the business realm including presentations, lectures, conferences, discussions, and shared work environments. Multimedia provides more information and interaction than that is possible or practical with conventional facilities. Conferences can link people that are too distant to meet face-to-face and provide a better forum for the exchange of ideas than a 3-way telephone call. Shared applications allow multiple people to work on a project.

While the types of multimedia applications are diverse, they share a common underlying trait: “The multimedia data that is being exchanged from node-to-node is real-time in nature and needs to be delivered in a timely fashion. Multimedia data is isochronous, that is, it is sensitive to delays and requires a constant amount of bandwidth, as opposed to bursty traffic.” [17] Multimedia data requires a high bandwidth, low end-to-end delay an low delay jitter or variance. Different media types of course have different tolerances, but they all have deadlines and data past its deadline can not be used [17,47]. Current approaches of transmitting regular data traffic which is bursty in nature and can tolerate a high variance in delay [43,45] are not suitable for multimedia traffic which is isochronous [47]. In multimedia systems, intelligent transmission of packets based on a data priority or deadline that will result in a much lower number of discarded packets is required [2,17].

Most business organizations and academic institutions already have LANs (local area networks) installed. A typical environment consists of workstations or PCs connected by a LAN, typically an Ethernet. Multimedia is getting more and more important to these organizations. As far as the computing power is concerned, the nodes possess sufficient computing resources to support multimedia applications. However, as far as communication is concerned, traditional Ethernets do not support high speed multimedia traffic [4,9,32]. Traditional Ethernet protocols do not support real-time, isochronous traffic.

These organizations and institutions have two options for adding multimedia capabilities. First is to attempt to utilize their existing networks by adding some devices and/or software in order to successfully transmit multimedia streams. The other option is to replace the underlying network infrastructure with new, high-speed network technologies, such as ATM [6,15]. However, currently ATM networks are expensive and there is not much experience with this technology. Therefore, a practical solution is to add capabilities to the existing LANs in order to successfully transmit multimedia streams.

The chapter is organized as follows. Section 2 describes the current state of the art in Ethernet including previous work on priority protocols. Section 3 describes the issues involved in providing a priority protocol on an Ethernet. Section 4 proposes three priority protocols. Section 5 presents a qualitative analysis of the three techniques and Section 6 presents the results of a simulation study of the proposed protocols. Finally, Section 7 summarizes.

2.2 State of the Art

The ISO Reference Model of Open System Interconnection [23] (also known as the ISO 7-layer reference) divides communication into seven different conceptual layers. The highest is the Application Layer, where different programs or applications communicate with each other. The lowest level is the Physical Layer, which is concerned with the actual physical data transmission and the hardware involved. This layers deals with the

specification of electrical signals and their timing sent over a physical wire. Above this level is the Data Link Layer or the level 2 layer. This layer specifies how data is sent from one host to another host over a single physical network. It includes the concept of data frames, checksums, and some error handling. Ethernet communication falls in this layer. Internet Protocol (IP), Transmission Control Protocol (TCP), as well as most other protocols involved in interprocess communication, represent higher level protocols. When several networks are connected together, IP provides point-to-point delivery of packets [11]. IP addresses are divided into two parts, the network address and the host address. In theory, a maximum of 254 hosts can be attached to a single Ethernet configured as a Class C subnet. However, in practice, the number of hosts is much smaller. Ethernets are a baseband bus communication medium that use CSMA/CD (carrier sense multiple access/collision detect) [30]. A node listens to the channel before it attempts to transmit, to avoid transmitting while another node is transmitting. If a node finds the channel is currently busy, it waits until the Ethernet becomes free and then begins transmitting. This scheme is known as 1-persistence. A collision occurs when two or more nodes attempt to transmit packets on the Ethernet simultaneously. All packets involved in a collision cannot be used and must be retransmitted. Collision detection allows a node to detect if its packet going out on the bus is being corrupted by another node that started transmitting almost simultaneously. When a collision occurs, each node waits for a random amount of time (called an exponential backoff, due to the mean time of the delay or backoff doubling with every consecutive collision) and then

tries again. This paradigm attempts to give all nodes an equal chance to transmit their data, while avoiding subsequent collisions due to repeated retransmission attempts [22,30,43,45]. There is no notion of priority at this level. Thus, transmission of multimedia data on an Ethernet does not get any preference and must compete equally with the transmission of regular data traffic.

Ethernet hardware, i.e., network interface, is designed to minimize the load on the CPU. The interface delivers only those packets to its node that are sent to it or broadcast to the entire net and ignores packets intended for other nodes. Broadcast packets are received by all nodes on an Ethernet, but are not propagated to adjoining Ethernets when the system consists of multiple interconnected Ethernets. The network interface can be configured to be in "promiscuous mode" in which it receives all packets sent on the bus [44]. However, this is likely to cause an unacceptable load on the CPU to process and discard all of the undesired packets.

Thus, Ethernet was designed to efficiently transfer regular data traffic for which variable delay is acceptable (e.g., transaction processing [28]). It is not suitable for multimedia traffic which can be highly bursty, and requires constant delay, fast (or real-time) delivery, and a large bandwidth. Priority protocols have been proposed in [5, 10, 47]. These are reservation based, which typically divide the bandwidth into timeslots and allow transmission during certain periods restricted to certain priorities. These approaches require new Ethernet interface hardware that can perform timing based on state changes sensed on the Ethernet. Other approaches are parametric and change the

persistence of the transmitter based on the priority of the data to transmit [41]. These approaches do not provide any guarantees that high priority packets get transmitted before low priority packets. In addition, they do not prevent collisions among data within the same priority class.

Token ring networks support the concept of priority data transfer [3] and suitability of priority data transfer on a token ring to multimedia traffic has been studied by Amer et al. [2]. Their study reveals that the priority mechanism on a token ring considerably improves the quality of multimedia data transfer.

2.3 Priority Protocols for Ethernet

2.3.1 Objectives

The main problem in existing Ethernets with regard to the integration of multimedia streams is that current CSMA/CD protocols are not appropriate for the demands multimedia data place on the network. Since there is no priority mechanism, on traditional Ethernets the transmission of multimedia data must compete equally with the transmission of regular data. Note, however, that multimedia streams must be received at the presentation node in time such that there is no interruption (or gap) in presentation from user's point of view (consider a moving picture presentation here). That is, the Ethernet must be able to deliver multimedia packets with a small and constant delay (in real-time). Clearly, the communication protocols of the Ethernet that are designed to efficiently transfer regular data traffic, which are bursty in nature and can tolerate

variable delay, must be modified to suit multimedia applications. One viable approach to handle this problem is to have a priority protocol, where high priority data gets precedence over low priority data. By using a protocol for priority transmission, multimedia data can be given a higher priority than regular data and thus allowing multimedia data to be delivered in a timely fashion over the Ethernet.

In this chapter, we develop a set of protocols that enable priority transmission of data on an Ethernet. That is, if a set of nodes are trying to transmit packets, the node with the highest priority packet succeeds. Development of such protocols involves changes to the Data Link protocol layer of the Ethernet to carry out packet transmission in priority order, and changes to higher layer protocols to allow user processes to define, specify, and interpret the priority of data.

Thus, this chapter focuses on the problems associated with supporting multimedia on Ethernet environments as well as qualitatively studying the performance aspects of the possible alternatives.

2.3.2 Issues Involved

There are two main issues involved in the design of priority protocols of an Ethernet, namely, functional decomposition of the protocol and the structuring of the protocol in the layered architecture.

Functional decomposition of the protocol

This deals with decomposing the protocol functions in components and identifying what functions the various components of the protocol must perform. There are two distinct functions of the protocol.

- *Dissemination of state information of nodes.* At any time, the node with the highest priority packet on an Ethernet is allowed to transmit its data on the Ethernet. In order to decide which node has the highest priority packet, each node must have knowledge of the highest priority packet of the other nodes on the Ethernet. Therefore, timely dissemination of this information is an important issue. Each node maintains a queue containing the priority of the highest priority packet every other node wishes to transmit.

- *Transmission of packets in priority order.* When a node wants to transmit a packet, in addition to the existing CSMA/CD checks, it compares the priority of its packet with that of the other nodes on the Ethernet. This is done by examining the queue containing the priorities. If it has the highest priority, it is allowed to transmit the packet immediately. Ties are broken by an *a priori* ranking of the nodes.

Structuring issues

These issues deal with assigning protocol functions to various layers (determining what layers are responsible for what). Priority cannot be implemented at the Network Layer

(level 3) or higher layers because implementation of priority at these layers will guarantee enforcement of priority only locally. It will not guarantee transmission of packets in the order of priority at global level (among all nodes). Also, in this case the processing cost and time required to pass information up and down the protocol layers is likely to be high. Therefore, the concept of priority needs to be implemented at the lowest level possible, specifically, the Data Link Layer, which corresponds to the Ethernet device drivers.

Ideally, the network interface, the Ethernet hardware controller, should be "intelligent" and be able to implement a priority protocol on its own (a hardware-oriented solution). However, if that is not possible, priority protocol can be implemented in the device driver controlled by the node's operating system (a software solution). An intelligent interface requires new, sophisticated hardware, and will be more expensive than the existing Ethernet controllers, while a device driver change only requires a software modification. There is a trade-off, however. If priority is implemented in the device driver, the device driver must then interpret all the broadcast priority information packets. This requires interrupting the CPU to process each of these packets. Clearly, the load on the CPU will be much greater in this case than if all the information were handled at the hardware interface level. Device drivers are responsible for writing and reading data to and from the hardware interface buffer. Since the device drivers are interrupt driven, the amount of work they do must be minimized. A context switch is

required every time a packet is received and additional time is required to pass information to and from the different protocol layers.

While the basic mechanisms of a priority protocol must be supported at the lowest protocol levels, other parts of it can be located at higher levels. There are cost benefit tradeoffs with respect to placing the support for the priority protocol at each level and these issues are explored in Section 2.5.

2.4 Priority Protocol Transmission Techniques

We now present three methods to implement Priority Ethernet. The first method is the Synchronous Method, the second is the Modified Synchronous Method and the third is the Asynchronous Method. First, we present issues involved in priority transmission, followed by the data structures and notations.

Issues

There are two main issues to be resolved: first is the dissemination of the priority information of the packets at the nodes and second is the transfer of the packets in priority order. After the priority information of packets at all nodes is disseminated, the rules for transmitting packets on Ethernets in the order of priority are relatively straightforward, i.e., a node waits until its packet has the highest priority. However, dissemination of the priority information of nodes to all the nodes efficiently is a complex issue. Dissemination of priority information should take precedence over the

transmission of data on an Ethernet. This is difficult to implement on the Ethernet that does not support priority communication to start with. There are several factors involved that may influence the performance, including the number of nodes, the amount of network traffic, and the distribution of the traffic among various priorities involved.

There are two issues involved in the dissemination of the priority information. The first is transmitting the priority of the highest priority message for every node on an Ethernet. The second is disseminating the information that a node has finished transmitting its data packet or packets so that the node with the next highest priority can start transmission. In this section, we present three methods for disseminating node priority information for the priority protocols on Ethernets.

2.4.1 Data Structures and Notations

The following data structures are used in all of the three methods. Note that the terms "site" and "node" are used interchangeably.

- Each node has two priority queues, called *pending* and *priority*. The *pending* queue holds all the messages a site wants to send. The *priority* queue contains the priority of the message at the front of each site's *pending* queue and represents the global view of the system for that site. Elements in these queues are stored in priority order, with the elements with the highest priority at the front of the queue.

- Message priorities can assume three values, *low*, *high*, and *none*, where *low* indicates the site wishes to send a low priority message, *high* indicates the site wishes to send a high priority message and *none* indicates it does not wish to send a message.

- All three methods use two types of broadcast messages. The first is *PRIO-REQ*, which requests all sites to broadcast their current priority. The second message, *PRIO-IS*, is used by a site to inform other sites of the priority of its highest priority message and contains a field that stores this priority information. In addition, there is a *YOUR-TURN* message which informs a site that it may now begin transmitting its message. This message is sent (point-to-point, not broadcast) by a site that finishes transmitting its message to the site with the next highest priority message. The third method uses an *IM-DONE* broadcast message to inform all sites that the last message in the queue has been transmitted.

- Constant t_{max} is twice the maximum time required to propagate a message across an Ethernet.

2.4.2 Synchronous Method

In the synchronous method of distributing the state information, after transmission of an arbitrary number of packets, there is a round of priority information exchange. In this round, each node sends a broadcast packet informing other nodes of the highest priority

packet it wishes to transmit. Nodes broadcast their status information in an *a priori* fixed order. For example, node *i* broadcasts its information after it receives node *i-1*'s state information, after which node *i+1* broadcasts its state. Provisions need to be made to handle adding and dropping nodes. This method should virtually eliminate Ethernet contention during the priority information exchange phase; however, some channel bandwidth is taken up by exchange of priority information and there is a processing overhead due to these broadcast packets moving up and down the protocol layers.

In this method, each node informs every other node of its highest priority message during the priority exchange phase. After that phase, every node has an identical copy of the *priority* queue and the node with the message at the front of the queue transmits its message and removes its entry from its *priority* queue. After having finished its transmission, that node informs the node with the message now at the front of its *priority* queue that it may now transmit its message (via a *YOUR-TURN* message). Upon receiving a *YOUR-TURN* message, a node deletes all entries in the queue up to and including its entry, transmits its message and informs the next node (via a *YOUR-TURN* message). This continues until there are no entries left in the *priority* queue, at which point a priority exchange phase is initiated. If a node wants to send a message when its *priority* queue is empty, it initiates the priority exchange phase (via a *PRIO-REQ*) message.

The Protocol

To send a message *M* from site *i* to site *j*

Add *M* to the *pending* queue.

If the *priority* queue is empty, then

Broadcast *PRIO-REQ* message.

When a *PRIO-REQ* message is received by *i* from *j*

Delete all entries from the *priority* queue.

If site $i = 1$ (i.e., *i* is the first site), then

Broadcast a *PRIO-IS* message containing the priority of the message at the top of the *pending* queue or *none* if the queue is empty. If the queue is not empty, the move that message from the *pending* queue to the *priority* queue.

When a *PRIO-IS* message is received by *i* from *j*

If the priority of the *PRIO-IS* message is not equal to *none* then

Insert an entry in the *priority* queue for site *j* with the priority contained in the message.

If $j = i - 1$, then

Broadcast a *PRIO-IS* message containing the priority of the message at the top of the *pending* queue or *none* if the queue is empty. If the queue is not empty, then move that message from the *pending* queue to the *priority* queue.

If $j = N$ and the element at the front of the *priority* queue is from site *i*, then

Transmit the message.

Remove the entry that corresponds to that message from the *priority* queue.

Transmit a *YOUR-TURN* message to the site that is at the top of the *priority* queue, or transmit a *PRIO-REQ* message if the *priority* queue is empty..

When a YOUR-TURN message is received by i from j

Remove all elements from the front of the *priority* queue until the message at the front of the queue is from node *i*.

Transmit the message at the top of the of the *priority* queue.

Remove the element corresponding to that message from the *priority* queue.

Transmit a *YOUR-TURN* message to the site that is at the top of the *priority* queue, or transmit a *PRIO-REQ* message if the *priority* queue is empty.

2.4.3 Modified Synchronous Method

In the Modified Synchronous Method, the priority information exchange phase occurs after a fixed number of data messages are transmitted; however, rather than having a synchronous exchange of the priority information, nodes on the Ethernet transmit their priority messages in the priority messages exchange phase using CSMA/CD, with exponential backoff to handle collisions.

In this method if a site does not have any messages to send, it does not transmit any priority information in the priority exchange phase, unlike the first approach in which a site must explicitly send a message to set its priority to *none*. However, since the number of sites which participate in the information exchange round may vary, it is difficult to know when this phase has ended. The priority information exchange phase is

considered ended if $t_{TimeOut}$ time units have elapsed since the last successful transmission of priority information. Therefore, the duration of the period is dynamically adjusted based on current demand. This makes this approach highly desirable for heterogeneous traffic environments, since nodes that have nothing to send are not required to transmit anything during this phase.

A non-1-persistent scheme should be used when sites broadcast their *PRIO-IS* message. In this scheme, sites pause for a random interval before broadcasting their *PRIO-IS* message to avoid collisions (originally, the delay was based on the site number, but that gives an unfair advantage to the low numbered sites). If a 1-persistent scheme is used, this method will suffer from excessive collisions ("broadcast storms"), resulting in a very low throughput.

In addition, the priority information exchange round is time-based in this approach; so *PRIO-IS* messages should only be sent during this round, and *PRIO-REQ* messages should only be sent during the data transmission phase (when the *priority* queues are empty). Transmission of these messages can be delayed beyond their respective phases due to Ethernet contention or packet collision and result in a message that is transmitted too late (referred to as a "late message"). The underlying CSMA/CD protocol must prevent the transmission of *PRIO-IS* messages outside of the priority information exchange phase, and *PRIO-REQ* messages during the priority information exchange phase. We adopt the policy of dropping such late message. A late *PRIO-REQ* message implies that another site initiated a priority information exchange phase, in which case the

late *PRIO-REQ* message is redundant and can be safely dropped. A late PRIO-IS message implies that the data transmission phase has already begun, and that all of the sites on the Ethernet have already determined the data transmission order. Since the site with the late message missed the priority information exchange phase, it must wait until the next round before it can transmit its priority. The data packet will still be at that site in the following round, so a new *PRIO-IS* message will be generated then. Therefore, the late *PRIO-IS* message can be safely dropped.

The synchronous method does not suffer from these problems, since the priority information exchange round is based on the number of priority information messages transmitted, not time-based. The asynchronous method has no priority information exchange round; so it is not vulnerable to these timing issues.

The Protocol

To send a message *M* from site *i* to site *j*

Add *M* to the *pending* queue.
If the *priority* queue is empty, then
    Send *PRIO-REQ* message.

When a *PRIO-REQ* message is received by *i* from *j*

Delete all entries from the *priority* queue.
If the *pending* queue is not empty, then

Broadcast a *PRIO-IS* message containing the priority of the message at the top of the *pending* queue, after pausing for a random delay period.

When a *PRIO-IS* message is received by *i* from *j*

Insert an entry in the *priority* queue. Remove the corresponding entry from the *pending* queue if $j = i$ for site *j* with the priority contained in the message.

If $t_{TimeOut}$ time units have elapsed since the last *PRIO-REQ* message was received at site *i*

and the element at the front of the *priority* queue is from *i*, then

Transmit the message.

Remove the entry that corresponds to that message from the *priority* queue.

Transmit a *YOUR-TURN* message to the site that is at the top of the *priority* queue.

When a *YOUR-TURN* message is received by *i* from *j*

Remove all elements from the front of the *priority* queue until the message at the front of the queue is from node *i*.

Transmit the message at the top of the of the *priority* queue.

Remove the element corresponding to that message from the *priority* queue and the *pending* queue.

Transmit a *YOUR-TURN* message to the site that is at the top of the *priority* queue or send a *PRIO-REQ* message if the *priority* queue is empty.

2.4.4 Asynchronous Method

The third approach is to allow asynchronous broadcast of node priority information. No distinct round of priority information exchange occurs. Instead, the broadcast of priority information is asynchronously embedded in the data transmission. That is, the priority data compete with the regular data for network bandwidth using the CSMA/CD protocol to resolve conflicts. In this approach, when a node gets a data packet for transmission from a user level process, it immediately broadcasts the priority of the packet provided the packet has a priority greater than anything pending transmission at that node. The amount of traffic and utilization of the Ethernet by the high and low priority traffic will influence the effectiveness of this method.

In this method, each node informs every other node of its highest priority message asynchronously, when the node wants to send the message. All sites receive the priority information messages, *PRIO-IS*, since they are broadcast on the Ethernet; therefore, all nodes add the new entry to their *priority* queue. The node with the highest priority message is the first to transmit its message. When it is finished transmitting, it informs the node with the next highest priority (via the *YOUR-TURN* message) that it can begin transmitting its message. The *YOUR-TURN* message contains two data fields, *highq* and *lowq*, which contain the ids of the sites at the front of the high priority queue and low priority queue, respectively. With this information, when a site receives a *YOUR-TURN* message, it deletes all entries at the front of its *priority* queues up to *highq* and *lowq* (for the high priority and low *priority* queues, respectively), since messages in front of them

have already been transmitted. The site then transmits its messages and sends a *YOUR-TURN* message to the next site, and so on. If, after transmitting its message, there is no other site waiting to send a message, the node then broadcasts an *IM-DONE* message, which causes all other sites to delete all entries in their *priority* queue up to and including the entry for the source of the *IM-DONE* message. Note that there is no *PRIO-REQ* message used by this method.

The Protocol

To send a message *M* from site *i* to site *j*

Add *M* to the *pending* queue.
If there is no higher priority message in the *pending* queue then
    Broadcast *PRIO-IS* message containing the priority of *M*.
    Move *M* from *pending* queue to *priority* queue.
If *M* is at the front of the *priority* queue, then
    Transmit *M*.
    Remove *M* from the *priority* queue.
    If the *priority* queue is not empty, then
        Set *highq* to the site at the top of the high *priority* queue
        Set *lowq* to the site at the top of the low *priority* queue
        Send a *YOUR-TURN* message, including *highq* and *lowq*, to the site that is at the top of the high *priority* queue. If the high *priority* queue is empty, then send the message to the site that is at the top of the low *priority* queue.
    else

Broadcast an *IM-DONE* message with priority field set to priority of *M*.

When a *PRIO-IS* message is received by *i* from *j*

Insert an entry in the *priority* queue for site *j* with the priority contained in the message.

When an *IM-DONE* message is received by *i* from *j*

Remove all elements from the front of *priority* queue corresponding to the priority of the *IM-DONE* message up to and including site *i*.

If message at front of the *priority* queue is from site *j*, send *YOUR-TURN* to site *j*. (Note: can be optimized so actual data transmission begins, rather than self-sending a *YOUR-TURN* message.

When a *YOUR-TURN* message is received by *i* from *j* containing *highq* and *lowq*

Remove all elements from the front of the high *priority* queue until the message at the front of the queue is from node *highq*.

Remove all elements from the front of the low *priority* queue until the message at the front of the queue is from node *lowq*.

Transmit the message at the top of the of the *priority* queue (corresponding to site *i*).

Remove the element corresponding to that message from the *priority* queue.

If the *priority* queue is not empty, then

Set *highq* to the site at the top of the high *priority* queue.

Set *lowq* to the site at the top of the low *priority* queue.

Send a *YOUR-TURN* message, including *highq* and *lowq*, to the site that is at the top of the high *priority* queue. If the high *priority* queue is empty, then send the message to the site that is at the top of the low *priority* queue.

else

broadcast an *IM-DONE* message with priority field set to priority of *M*.

2.4.5 Dissemination of Completed Transmission

Recall that in order for the next highest priority node to begin transmission, it must be informed when the highest priority node has finished transmitting its packet. We have identified two approaches to disseminating the information that a node has finished transmitting its data. The first approach is to have every node monitor the Ethernet and update their *priority* queue every time a node successfully transmits data. For current Ethernet interfaces, this requires the hardware to be set in "promiscuous mode" which may put a heavy load on the CPU because it must do context switches and pass the packets up and down the protocol layers. However, if the interface were intelligent and could update the queue without the CPU having to process all the packets, this can be a viable approach. In the second approach a node, after successfully transmitting data, transmits a packet to the site with the next highest priority indicating that it has finished transmitting data. New priority information about the node can be included in this packet.

The problem of dissemination of completed transmission has been solved in the previous three methods with the use of the *YOUR-TURN* message. This message is sent point-to-point to the node with the next highest priority message. Since all sites have identical

priority queues and a well defined order of transmission, a site is properly notified when it is that site's turn to transmit. In addition, the ordering of the queue uniformly determines the next site to transmit. For all three methods, there is an overhead of a packet that is transmitted after every data message. The larger the size of the data, the smaller the overhead is with respect to the total data transmitted. However, in the worst case when each data message occupies one physical Ethernet packet or frame, the number of messages transmitted on the Ethernet are doubled.

2.5 Qualitative Analysis of the Methods

In this section, we qualitatively analyze these methods for the following qualities: efficiency, potential collisions, potential improvements from using a smart interface, fairness and message overhead.

Efficiency

The synchronous approach can suffer from delays during the information exchange phase. Since each site must wait until its turn to broadcast its priority information, every delay incurred by each site adds to the time required to complete the entire exchange phase. These delays can be caused by the time the device driver of node's operating system takes to process the priority information packets before the node can transmit its priority information. In addition, sites that have no message to send must still send a *PRIO-IS* message with the priority field set to *none*. In high, uniform traffic, when most sites have message to send, this method is very efficient.

The second approach alleviates these delays by allowing the network interface to transmit the priority information packets as soon as the Ethernet is free during the priority information exchange phase. However, delays can be incurred due to packet collisions and retransmissions during this phase. In addition, in this method a node must wait for a time-out period before the information exchange phase ends. If the time-out value is too small, some sites will not be able to transmit their priority information and therefore not be able to transmit their data in the following data transmission phase. If the time-out value is too large, time is wasted waiting for the information exchange phase to end and network bandwidth utilization suffers. The second method is superior to the first in heterogeneous traffic, where few sites transmit messages at a higher rate than others. If all sites do have a message to send in every phase, the second method is no more efficient than the first. The number of nodes on the Ethernet, the data traffic patterns, and the latency from the device drivers in the first method will affect the relative performance of the first two methods.

The third method is better for low traffic, since a high priority message does not have to wait for the priority information exchange phase before it can be transmitted. In the other two methods, if a high priority message arrives at a node to be transmitted just after a priority exchange phase, it will have to wait for all of the messages to be transmitted before the its priority information can be exchanged with the other sites. In this regard, the third method is the most fair; however, it is also prone to the most collisions, which is discussed subsequently.

Potential collisions

Collisions directly relate to wasted bandwidth on an Ethernet and should be minimized. The first method should be free from collisions since the priority information exchange phase is completely synchronous and only the site with the highest priority message will transmit during the data transmission phase. However, there is lower bandwidth utilization, since there will be "wasted time" on the net during the delays incurred by processing the priority information messages. The only way a collision can occur is if two sites simultaneously send a *PRIO-REQ* message after the system has been idle. The second method should reduce delays from the time wasted due to the priority information exchange by allowing asynchronous broadcast of the priority information messages during the priority information exchange phase, although delays due to collisions are possible. If all sites attempt to transmit their *PRIO-IS* message as soon as they receive a *PRIO-REQ*, the collision rate dramatically increases. This is caused by the clustering effect of sites simultaneously sensing an idle net immediately after the *PRIO-REQ* and each then attempting to send their *PRIO-IS*. Using a non-1-persistent scheme for transmitting the *PRIO-IS* message, such as waiting a random interval with the mean delay equal to the site id, should eliminate the clustering. During actual data transmission, there should be no collisions, since only the highest priority site on the net will attempt to transmit its data. The third method can suffer from more collisions and subsequently delays due to collisions than the other two methods since priority

information messages can be broadcast at any time. However, this method will not waste time due to the state information exchange rounds that the first two methods have.

Potential improvements from using a smart interface

A smart interface is capable of managing the local *priority* queue and update it appropriately when it receives *PRIO-IS* and *PRIO-REQ* broadcast messages, without intervention from the node's CPU. The first method benefits the most since the synchronous operations of the priority information exchange phase consume CPU resources. For every *PRIO-IS* message received, each node's CPU must decide if it is its turn to send the node's priority now or not. Every *PRIO-IS* message must go up and down the protocol layers before it can be processed. In addition, each message generates context switches which consume additional CPU cycles. With a smart interface, all the details of when to broadcast its priority information are handled by the smart interface, which would already be tracking which sites have already broadcast their priority information and if it is this node's turn to broadcast its priority now or not.

The other benefit of having a smart interface would be shared by all three methods. That is, time is saved that would be needed to process the broadcast messages, both the time spent for context switches (at least two per message) and the CPU time needed to process the broadcast message itself. This time would be proportional to the number of sites on the local Ethernet, since each site must broadcast its priority.

In addition, the network interface would be able to process the messages sent when a site finishes transmitting its data, i.e., the *YOUR-TURN* message.

Fairness

The priority information exchange phases used by the first two methods can be invoked either after every message, or after some fixed number of messages. This parameter will affect the responsiveness to high priority versus starvation issue of the algorithm. If the value of the parameter is low, then the priority exchange phase occurs more often; so it is more likely that a site with a high priority message can send it sooner. However, other sites, with lower priority message might face starvation. If the parameter is large, the information exchange phase occurs less often and more of the sites will eventually be able to send their messages; however, a site with a higher priority message will be forced to wait until after the next information exchange phase until it can transmit its data.

A reasonable compromise is to set the parameter to the number of sites that want to send a message at the priority exchange phase, that is, for the synchronous method the value is the number of sites that did not send a *PRIO-IS* message with a priority value set to *none* and for the modified synchronous method the value is the number of sites that send a *PRIO-IS* message.

Message overhead

All three methods should have a comparable message overhead, since one control message is required to establish the priority and one is required to relinquish it, although

the collision rate differs for each method. The methods only differ in the way they disseminate their priority information. The synchronous method has fixed rounds of priority information exchange in which each site must transmit the priority of their highest priority message. In the modified synchronous method, only sites that have a message to send are required to participate in the priority information exchange round. Because of this, the length of the priority information exchange rounds can vary. The asynchronous method has no round of priority information exchange; instead, each site transmits their priority when they have a packet to transmit. The synchronous method has a slightly higher message overhead than the other two methods, since each site must send a *PRIO-IS* even if it has no message to send. As the load increases and the number of sites with no message to send decreases, the difference in the message overhead between the synchronous method and the other two methods should decrease towards zero.

2.6 Simulation Study

The performance of the three proposed protocols, namely, the synchronous method, the modified synchronous method, and the asynchronous method, was studied via extensive simulation. The performance of the basic Ethernet, with CSMA/CD only, was also simulated to evaluate the performance enhancement due to the three proposed protocols. We describe the system model used in the simulation and then present results from the simulation study.

2.6.1 System Model

The system is modeled by a set of nodes sending messages to other nodes. Each node is modeled by a set of processes, specifically a *receiver*, *transmitter*, and *generator* process. Additionally, each node has two queues, a *pending* queue which holds messages generated by this node that are waiting for transmission, and a *priority* queue which contains the priority information of other nodes on the net (refer to Figure 1). The receiver process is responsible for receiving messages from the network, processing the protocol messages, and taking the appropriate actions, such as adding an entry to the *priority* queue or initiating a message transmission. The transmitter process is responsible for actually sending a message out on the Ethernet. It models the CSMA/CD protocol. The generator process creates messages to send and puts them in the *pending* queue. The generation of the data is modeled by a Poisson process and a Markov Modulated Poisson Process (MMPP) [18] described in the next section.

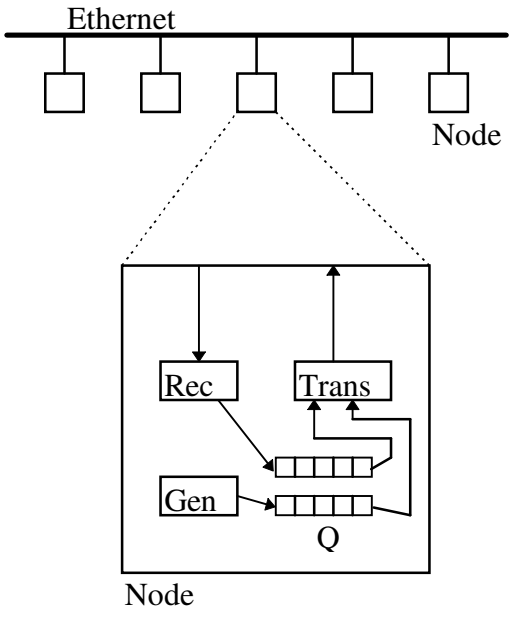

Figure 1: Processes Modeled in a Node in the Ethernet Simulation

Traffic Model

The generator process generates both bursty and non-bursty traffic. Non-bursty traffic is modeled as a Poisson process, where packets have an exponentially distributed arrival rate with a mean of $\lambda_{nonburst}$, which is varied to change the applied load. Bursty traffic source is modeled by a 2-state Markov modulated Poisson process, where the burst size (number of messages in a burst), burst arrival rate and average bandwidth are variable.

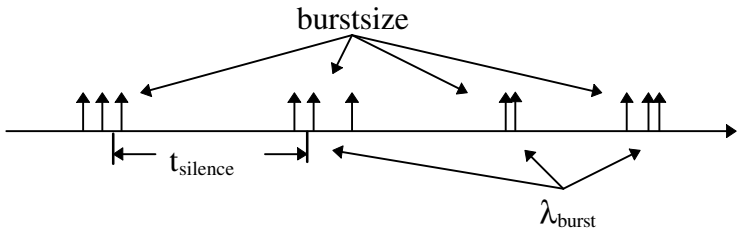

Figure 2: Bursty Traffic

Figure 2 shows an example of bursty traffic. The bursty traffic consists of an *on* period during which *n* packets are generated, where *n* has an exponential distribution and a mean of *burstsize* during that period. Packets are generated with an arrival rate of λ_{burst}. After the *on* period, there is a *silence* period which has an exponential distribution and a mean of $t_{silence}$. *Burstsize* and λ_{burst} are varied to change the applied load. $t_{silence}$ is derived from the other parameters and is specified in a later section as Equation [5].

For simulation runs with only Poisson traffic, i.e., 0% bursty traffic, a fixed fraction of the packets were marked as high priority and the remaining packets were marked as low priority. The combined Poisson and bursty traffic model more closely represents combined real-time and non-real time traffic, with the bursty source representing the real-time traffic. For simulation runs with bursty traffic, the bursty data was marked as high priority, and the non-bursty data was marked as low priority.

The combined bursty/non-bursty model was chosen to model both real-time and non-real-time traffic that would be seen on an Ethernet. File transfers and other non-real time

data are modeled by the Poisson process. A 2-state Markov-modulated Poisson process was used to model variable bit rate (VBR) real-time data, such as packetized voice processes [18]. The applied load is determined by selecting values for the mean of the on-off times (i.e., silence) and the arrival rate for the "on" state (i.e., λ_{burst}).

For the non-uniform traffic cases (refer to Section 2.6.5), one site was responsible for 50% of the total applied load and the remaining sites were responsible for the other 50%.

Assumptions

Nodes on the Ethernet were assumed to be spaced at equal distances physically on the net. The data packet size is variable, however, for an individual run, all packets are assumed to be the same size. If 16 consecutive attempts to transmit a packet result in collisions, then the packet is dropped by the sender. The traffic was uniformly distributed to every node, i.e., every node has the same λ values.

How the Simulation was Run

The simulation programs were written in C++ using the CSIM [39] package to simulate multiple processes, virtual time, and ease the collection of data. The programs were run on HP and Sun workstations using PERL scripts to facilitate varying different parameters and parsing the output. All four models were run with various combinations of sites and applied loads and traffic types combinations (Poisson and bursty).

The simulation was run to collect statistics for each data point until 50,000 messages were successfully transmitted; however, for data points in the vicinity of unstable regions of the graph, the number of messages was increased to 100,000.

The stability of the system was determined by the average queue length; when the system becomes unstable there is a sharp increase in the average queue length, which is typically 2 orders of magnitude. Data in this region was discarded, since the response time reports the time for only those packets that were successfully transmitted. Undelivered packets (with an infinite waiting time) are not computed, so the response time reported when the system is unstable can actually decrease as the load increases, due to the first few packets being delivered quickly before the low priority traffic is essentially cut off by the excessive high priority traffic.

Performance Measures Collected

The following performance measures were collected: average response time for high priority data, average response time for low priority data, average high priority queue length, average low priority queue length, total number of messages transmitted (including control messages), total number of collisions, total number of packets dropped by sender (due to excessive collisions), and effective data throughput.

The average response time represents the average of the time it takes from the initial creation of the packet by the generator process until the end of successful transmission (i.e., transmitted without collisions). Since Ethernet does not distinguish between high

and low priority data, the response time computed with the basic Ethernet simulation represents the average response time of both high and low priority packets. The priority protocol simulations report results for high and low priority data separately. Applied load, as a percentage of the maximum Ethernet bandwidth, was used as the independent variable for the graphs.

Parameter values

The maximum bandwidth of the Ethernet was set to be 10 Mbps and the maximum time to propagate a message from one end of the net to the other was set to be 51.2 microseconds. The simulator allowed the data size of a packet to be varied from a minimum of 64 bytes to a maximum of 1536 bytes, however, only 100 and 1000 bytes packets were used in our simulation runs.

The number of sites simulated varied from 4 to 12. The non-bursty traffic source was modeled by a Poisson source with a variable arrival rate, $\lambda_{nonburst}$. By varying $\lambda_{nonburst}$, the applied load on the Ethernet was varied up to 90% of the maximum Ethernet bandwidth. The value of $\lambda_{nonburst}$ represents the aggregate arrival rate for all sites on the net. The arrival rate for an individual site is $\lambda_{nonburst} / sites$, where *sites* is the number of sites in the simulation. In the initial simulations, traffic was modeled with only the Poisson source, with part of that traffic was marked as high priority (as determined by the variable *highprio*). For these runs, the traffic consisted of 20% high priority and 80% low priority data.

The bursty traffic generator was added later to more accurately model the data traffic. In those simulations, the high priority data came from the bursty source and the low priority data came from the Poisson source. For the combined Poisson and bursty source simulations, different values, from 10 - 50% of the data was chosen to be high priority (bursty). The applied load was varied from 10% to 90% of total possible network bandwidth and the average burst size was set to 5 messages. The bandwidth during the "on" period of the bursty source was set to 0.5 Mb/s, to model the typical bandwidth for a compressed video source.

By utilizing the applied load ($bandwidth_{applied\ load}$), percent of high priority/bursty data (*highprio*), and the size of the data packets (*packetsize,* in bytes), $\lambda_{nonburst}$, for non-bursty, low priority traffic, is determined by:

$$\lambda_{nonburst} = bandwidth_{applied\ load} * (1 - highprio) / (packetsize * 8) \quad [1]$$

In order to compute the average silence period for the bursty source, first the amount of time the bursty source transmits (average) must be determined. Given the average number of messages in a burst (*burstsize*) and the arrival rate when the bursty source is "on" (λ_{burst}), this period, defined as $t_{talktime}$, is:

$$t_{talktime} = burstsize / \lambda_{burst} \quad [2]$$

λ_{burst} is determined by the size of the data packets and the transmission bandwidth when the bursty source is "on" ($bandwidth_{onburst}$), which must be specified.

$$\lambda_{burst} = bandwidth_{onburst} / packetsize / 8 \qquad [3]$$

The total bandwidth for a bursty source can be expressed in terms of the total applied load or in terms of the silence and talking times. Therefore:

$$bandwidth_{burst} = bandwidth_{applied\ load} * highprio$$

$$= bandwidth_{ethernet} * (t_{talktime} / (t_{talktime} + t_{silence})) \qquad [4]$$

Finally, by regrouping equation [4] and solving for $t_{silence}$, the silence period can be expressed as:

$$t_{silence} = bandwidth_{ethernet} * t_{talktime} / (bandwidth_{applied\ load} * highprio) - t_{talktime} \qquad [5]$$

Thus, given the total applied bandwidth, percent of high priority data, average number of messages in a burst, Ethernet speed and λ_{burst}, $\lambda_{nonburst}$ and the silence period for the bursty source can be determined from equations [1] and [5], respectively.

2.6.2 Basic Ethernet Simulation

First, a basic Ethernet was simulated for different numbers of nodes and different traffic and the response time was computed. Figure 3 shows the response time for traffic with Poisson distribution, for an applied load up to 90% total network bandwidth (9 Mbps). As the arrival rate increases, the response time increases exponentially, due to an increase in the number of collisions. The network can be considered unstable (and unusable) when the response time is more than one second. When the network becomes unstable, the average queue size at each site jumps from the tens to the thousands.

Figure 4 shows the response time for combined Poisson and bursty traffic, for 20% bursty traffic for an applied load up to 90% total network bandwidth (9 Mbps). The graph is very similar to the 0% bursty traffic graph. In both figures, the 4 site case becomes unstable just under 70% load and the 12 site case becomes unstable around 37% load with the other cases falling in between this range.

In both simulations, when the number of sites on the Ethernet is decreased, the maximum usable throughput available on the Ethernet increases. This is due to the fact that the probability of two packets colliding is proportional to the number of sites. The more sites attempting to transmit on an Ethernet, the better the chance that two of those sites attempt to transmit simultaneously resulting in a collision and wasted bandwidth. As the applied load increases, collisions are more likely to occur, so the amount of bandwidth lost to collisions increases and the average response time increases. An Ethernet with many sites becomes unstable due to collisions before an Ethernet with few sites.

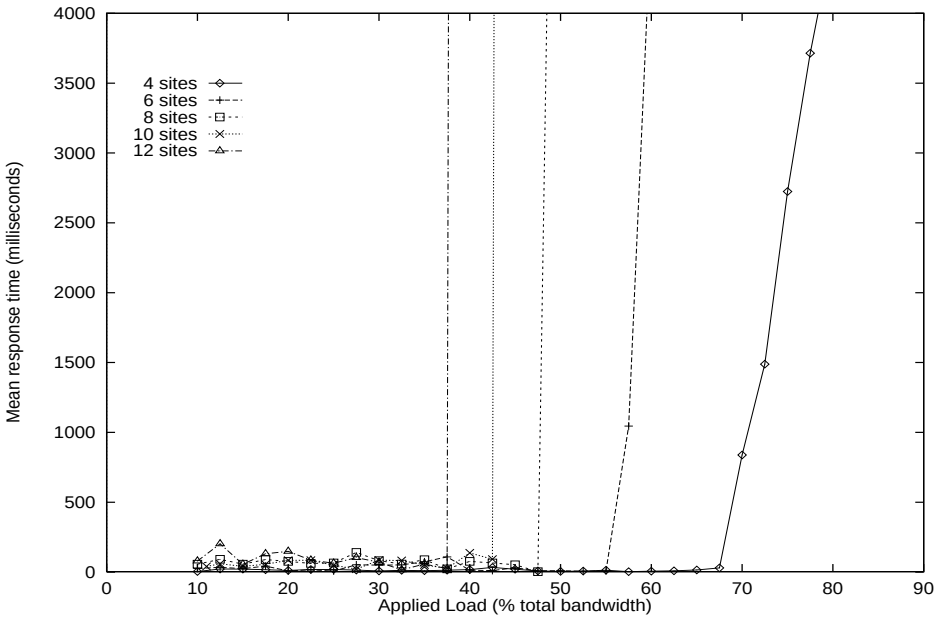

Figure 3: Ethernet Simulation for Poisson Traffic Only.

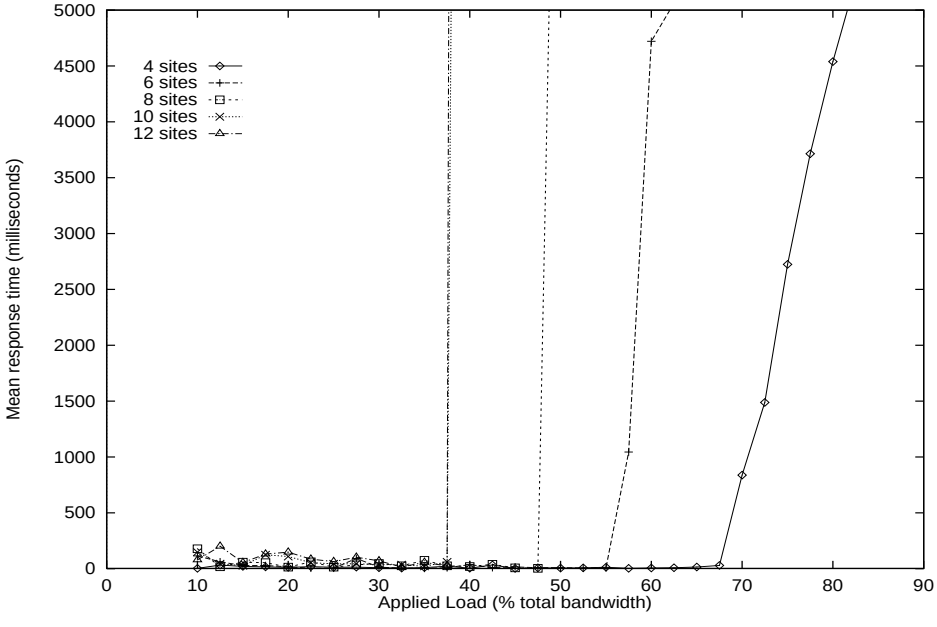

Figure 4: Ethernet Simulation with 20 % Bursty Traffic

2.6.3 Synchronous Method Simulation

Figures 5, 7, 9, 11, and 13 show the response time for high priority traffic and Figures 6, 8, 10, 12, and 14 show the response time for low priority traffic for 0%, 10%, 20%, 30%, 40% and 50% bursty traffic, respectively. The simulations were run for 4, 6, 8, 10, and 12 sites for an applied load ranging from 10% to 90% of the total network bandwidth (10 Mbits/sec). The following observations were made.

Observation 1: In general, the synchronous method sustains a higher throughput with lower response time than an Ethernet with only CSMA/CD.

For the basic Ethernet, only the 4 site case could sustain more than 60% applied load. But, the synchronous method sustained 60% or more applied load for all cases (refer to Figures 8, 10, 12, and 14). The improved performance for both high and low priority data in the synchronous method, as compared to the basic Ethernet (in Figure 3 and Figure 4), is due to the low the collision rate, typically on the order of 1%. The synchronous method is reservation based, with a well-defined order for both data and priority information transmission. The only way a collision can occur is if two sites attempt to initiate a priority information exchange phase simultaneously after the Ethernet has been idle, which should occur infrequently.

Observation 2: High priority data can sustain higher throughput, but at the expense of the low priority data.

The high priority data has a lower response time and greater maximum sustained throughput than the low priority data in all of the figures. For each pair of figures for low and high priority data, at the point the low priority curve increases exponentially, the high priority curve tends to level off. For example, in Figure 7, this point occurs at 70%, and in Figure 11, it occurs at 80%. This is due to the fact that beyond this point only the high priority data is being transmitted and the low priority data is completely choked off. Therefore, the effective applied load decreases since the low priority data no longer contributes to the load. Beyond this point, the Ethernet is already unstable and unusable, since no low priority data is getting through.

Observation 3: The response time for high priority data improves as the load initially increases from 10% to 20%.

In the synchronous method, collisions can only occur when the Ethernet has been idle. The Ethernet becomes idle more often at low loads. As the load increases, it becomes idle less often and the probability for a collision decreases, which results in a lower response time. Beyond 20% applied load, the reduction is overwhelmed by the increase in the applied load.

Observation 4: As the amount of bursty traffic increases, the maximum sustainable throughput for low priority traffic increases.

For 10% bursty traffic, the results show that the system sustains 50 - 60% applied load, for 30% bursty traffic, the system sustains 70 - 80%, and for 50% bursty traffic, the

system sustains around 80%, for low priority data (refer to Figures 8, 12, and 14). For a given applied load, as the percent of bursty, high priority traffic increases, the percent of non-bursty, low priority traffic decreases. So there is less competition among the low priority data for bandwidth, which results in a higher throughput. Even though the amount of bursty traffic increases, which reduces the bandwidth available to low priority data, the decrease in non-bursty traffic has a greater impact on the low priority throughput.

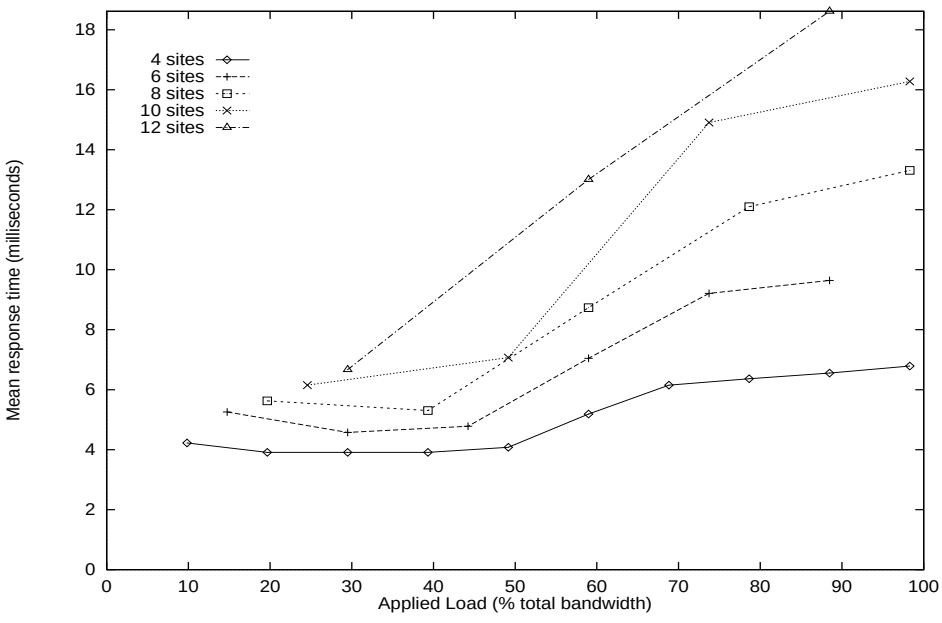

Figure 5: Response Time for High Priority Traffic for the Synchronous Method (0% Bursty Traffic).

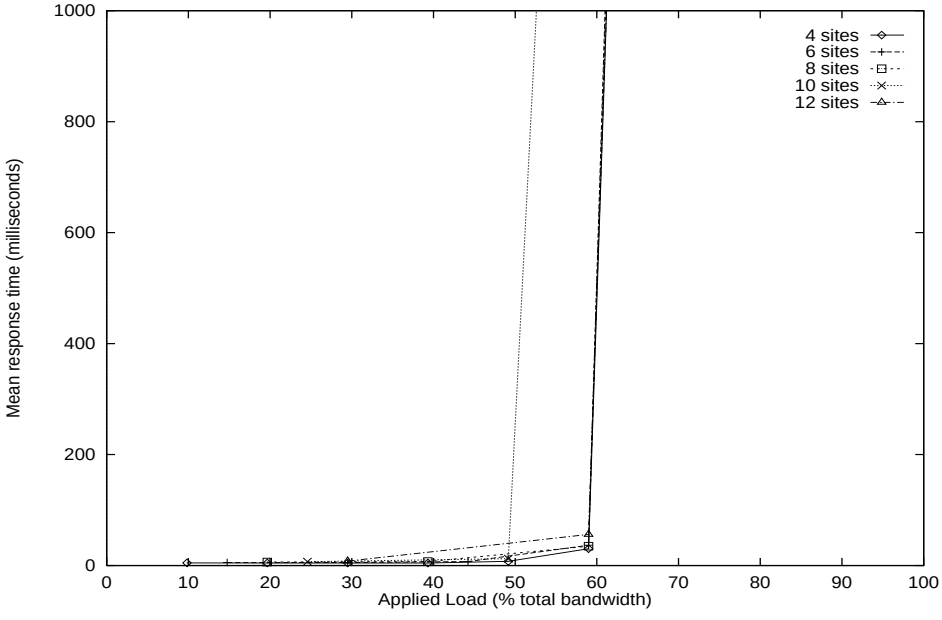

Figure 6: Response Time for Low Priority Data Using the Synchronous Method (0% Bursty Traffic).

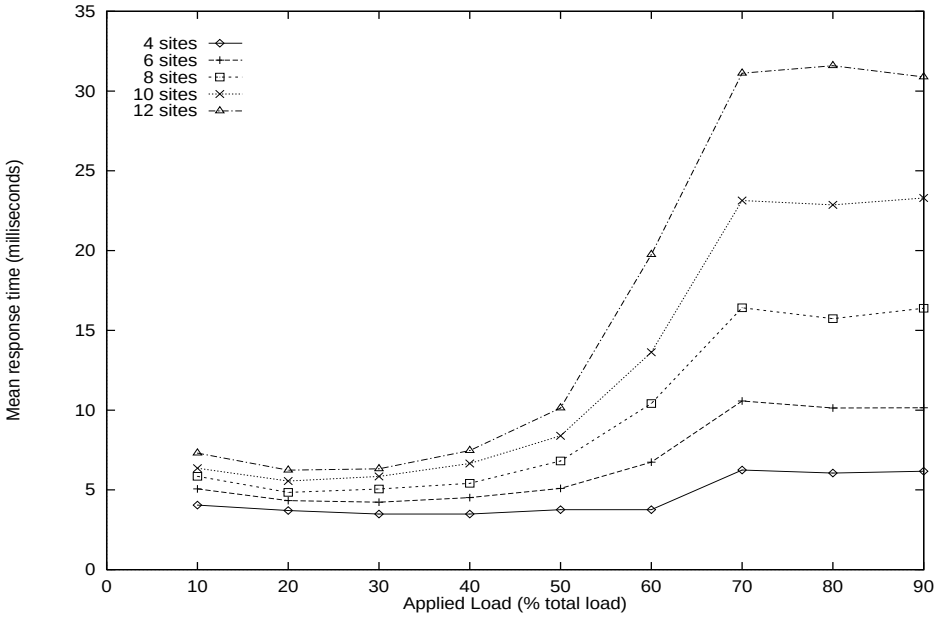

Figure 7: Response Time for High Priority Traffic for Synchronous Method (10% Bursty Traffic)

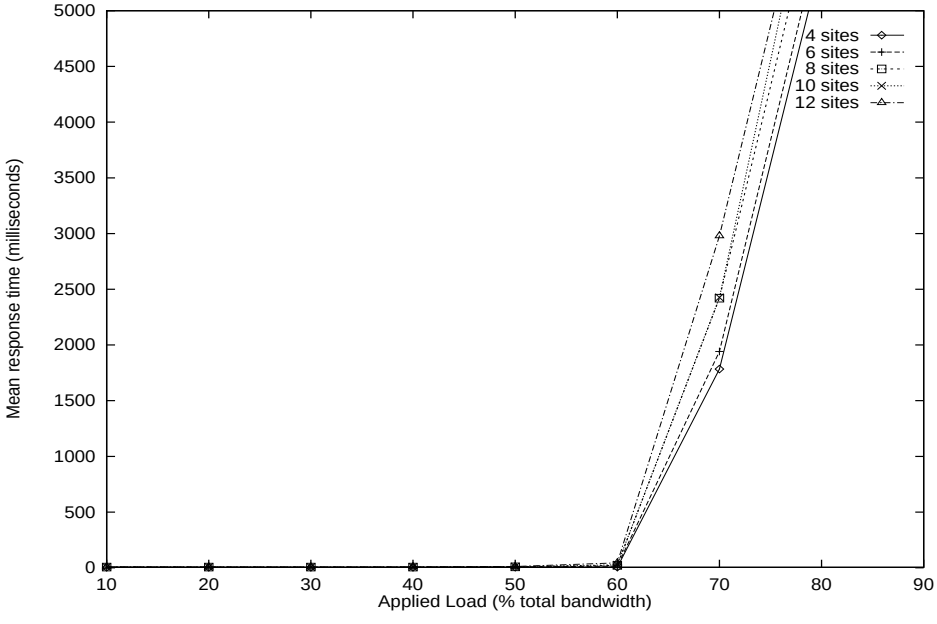

Figure 8: Response Time for Low Priority Data for Synchronous Method (10% Bursty Traffic)

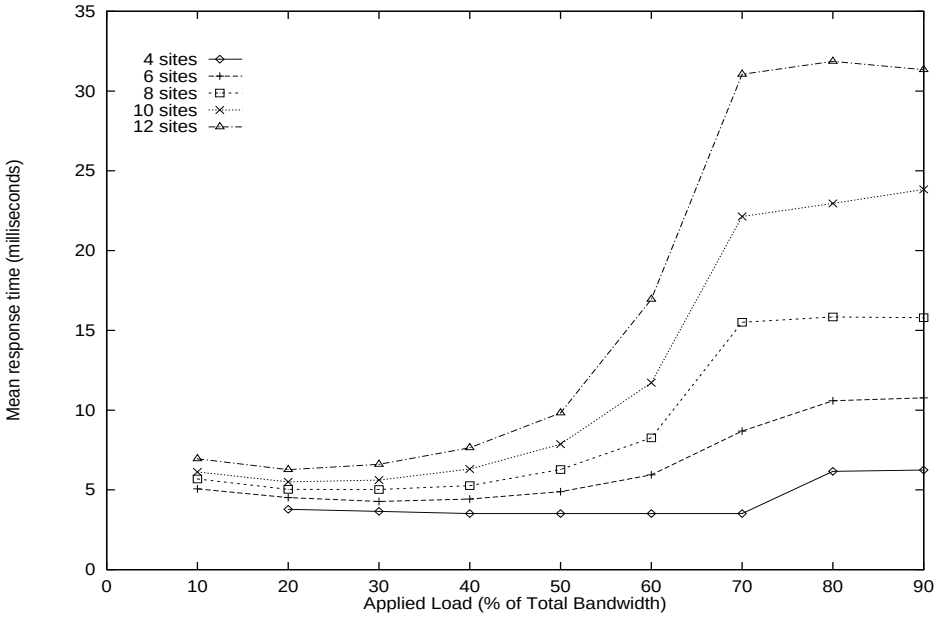

Figure 9: Response Time for High Priority Traffic for the Synchronous Method (20% Bursty Traffic)

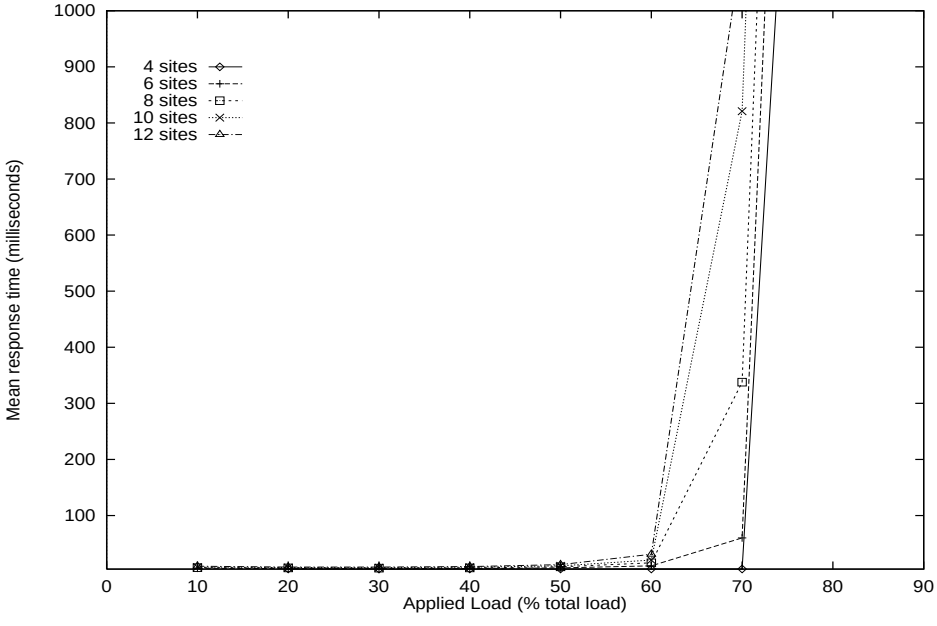

Figure 10: Response Time for Low Priority Traffic for the Synchronous Method (20% Bursty Traffic)

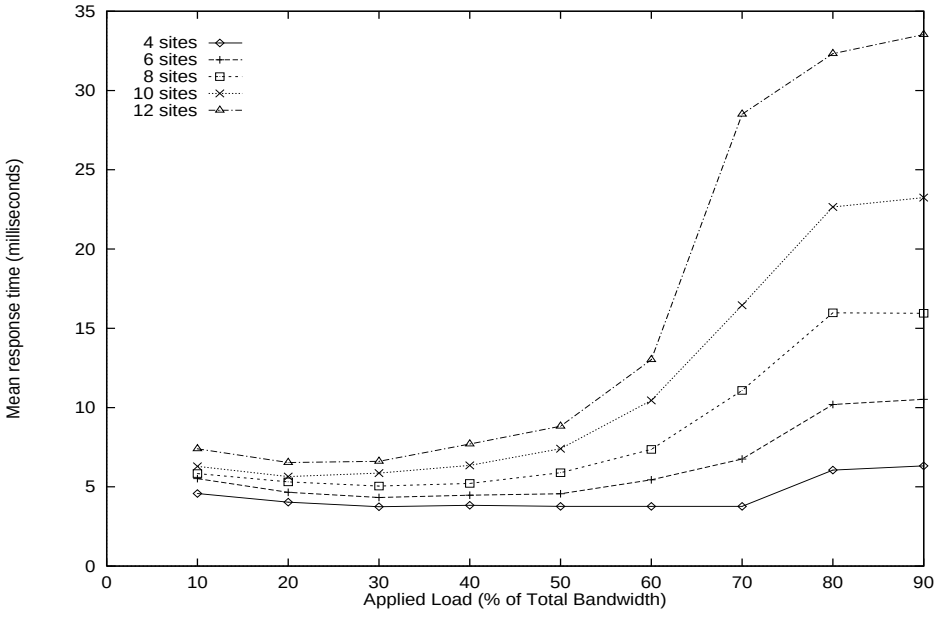

Figure 11: Response Time for High Priority Traffic for Synchronous Method (30% Bursty Traffic)

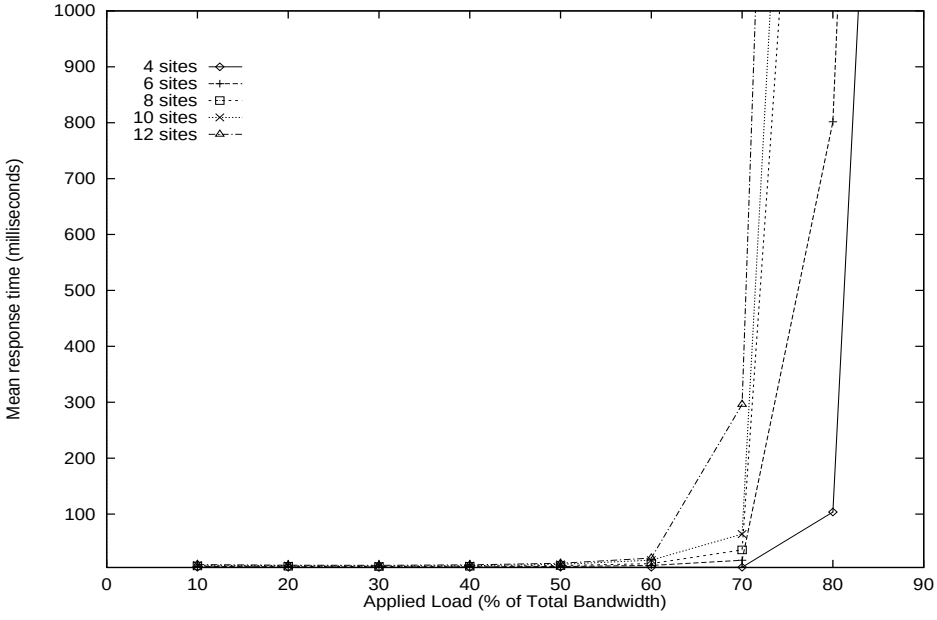

Figure 12: Response Time for Low Priority Traffic for Synchronous Method (30% Bursty Traffic)

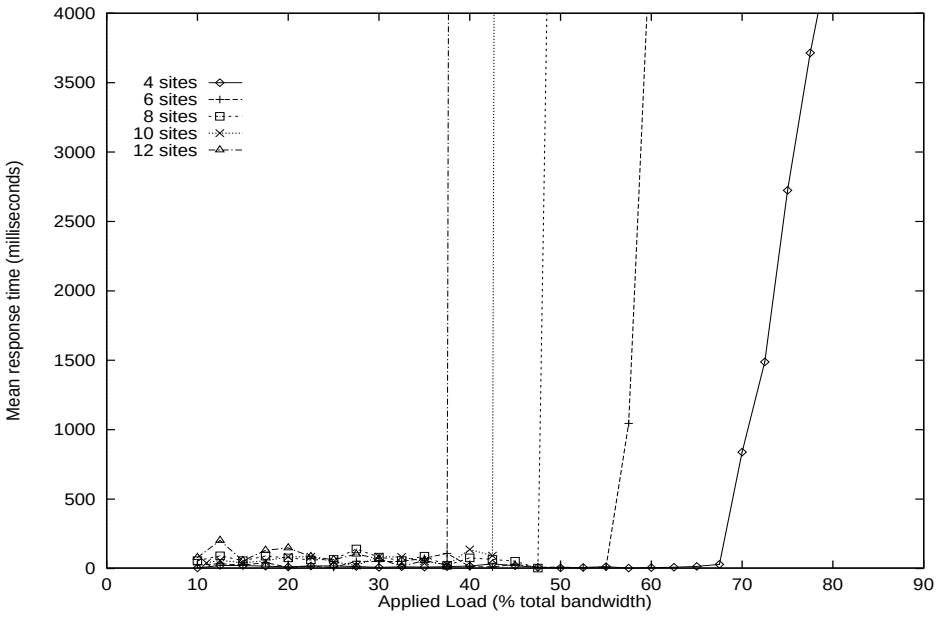

Figure 13: Response Time for High Priority Traffic for Synchronous Method (50% Bursty Traffic)

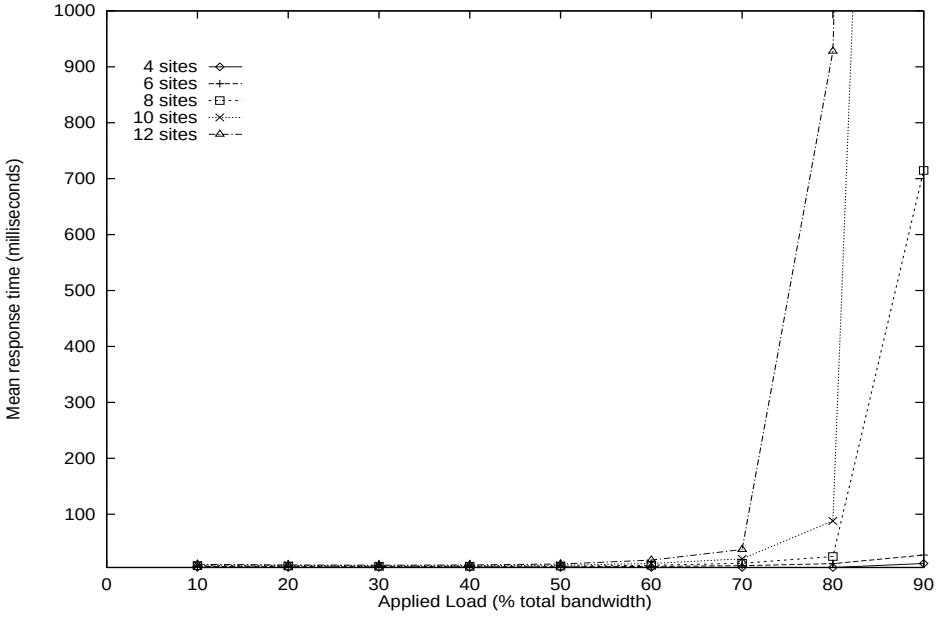

Figure 14: Response Time for Low Priority Traffic for Synchronous Method (50% Bursty Traffic)

2.6.4 Modified Synchronous Method Simulation

Figures 15, 17, and 19 show the response time for high priority traffic and Figures 16, 18, and 20 show the response time for low priority traffic for 20%, 30%, and 50% bursty traffic, respectively. The simulations were run for 4, 6, 8 and 10 sites (the 12 site case was not run due size limitations on some of the CSIM resources) for an applied load ranging from 10% to 90% of the total network bandwidth (10 Mbits/sec). The following observations were made.

Observation 1: For most cases, the performance of the modified synchronous method is better than that of the basic Ethernet.

The basic Ethernet becomes unstable approximately at 67% for 4 sites and 47% for 10 sites (in Figure 3 and Figure 4). In all but the 4 site case, the modified synchronous method sustains a higher total throughput than the basic Ethernet. For 50% bursty traffic (in Figure 19 and Figure 20), in all cases, including the 4 site case, the modified synchronous method has a higher sustained throughput than the basic Ethernet. In general, the response time is low, typically under 50 milliseconds. The improved performance of this method over the basic Ethernet is due to the lack of collisions among data packets. Although, unlike the synchronous method, priority information packets can collide, the order of the data packet transmission is controlled, thus preventing collisions.

Observation 2: For a uniform traffic distribution, the synchronous method has a slightly lower response time and higher throughput than the modified synchronous method.

For 20% bursty traffic, the synchronous method and modified synchronous method have a maximum throughput for low priority data of approximately 70% and 60%, respectively (Figure 10 and Figure 16). For the high priority data, the response time for the two methods is close, but the synchronous method is slightly lower. For example, for 80% applied load, the high priority response time for the synchronous and asynchronous method ranges from 6 - 30 milliseconds, and 10 - 50 milliseconds, respectively (Figure 9 and Figure 15). For 50% bursty traffic, the maximum throughputs for the two methods are 80 - 90% and 70 - 80%, respectively (Figure 14 and Figure 20), and the high priority response time for 80% applied load ranges from 4 - 30 milliseconds, and 10 - 60 milliseconds, respectively (Figure 13 and Figure 19).

The collision rate is higher in this method than in the synchronous method because the *PRIO-IS* messages can collide with each other, resulting in wasted bandwidth. In addition, under high uniform loads, the modified synchronous method must still wait for the priority information exchange phase to time out, which further wastes bandwidth. Note that even if all sites were to respond, there is a high probability that two of the responses would collide and the round would still end by a time out. If a site cannot transmit its *PRIO-IS* message during a priority information exchange phase, then it must wait for the next cycle before it can transmit its priority information, which increases the

response time. The result is that the synchronous method has a lower response time and greater sustained maximum throughput than the modified synchronous method.

Under a non-uniform traffic load, where only a few sites have many of messages to transmit while most sites have few messages to transmit, the modified synchronous method should perform better than the synchronous method. Because the priority information exchange phase is time based in the modified synchronous method, this can be very brief if only a few sites have messages to send, while the synchronous method requires that every site respond with their priority information, which can waste bandwidth.

Observation 3: In general, high priority data has a lower response time than low priority data.

In the modified synchronous method, like the synchronous method, high priority data has lower response time than low priority data, except for low applied loads (less than 30%), when the two values are typically within 0.1 milliseconds of each other.

The response time for high priority data is very low, typically under 20 milliseconds, and has a characteristic shape similar to the high priority data response time for the synchronous method. The high priority curves increase rapidly and then level off. The point at which they level off corresponds to the point at which the low priority response time increases exponentially. At this point, the amount of high priority data prevents any low priority traffic from being transmitted, so the entire applied load comes from the

high priority data. So, if 20% of the traffic is from the bursty (high priority) source and it is the only data transmitted, the applied load is now only 1/5th of its previous value, which causes the response time curve to level off. As the load increases, the curve would again rise exponentially, although the applied load in the simulations do not exceed 90% of the total network bandwidth.

Observation 4: As the amount of bursty traffic increases, the maximum sustainable throughput for low priority data increases.

The response time for the low priority data increases exponentially and becomes unstable at 50 - 60% applied load for 20% bursty traffic, and at 70 - 80% applied load for 50% bursty traffic (refer to Figure 16 and Figure 20). This effect is the same as in the synchronous method. As the amount of bursty traffic increases, for a given load the amount of non-bursty traffic decreases. So the lower bandwidth for non-bursty traffic results in less competition for bandwidth among low priority data, which allows a higher sustained throughput for the low priority data.

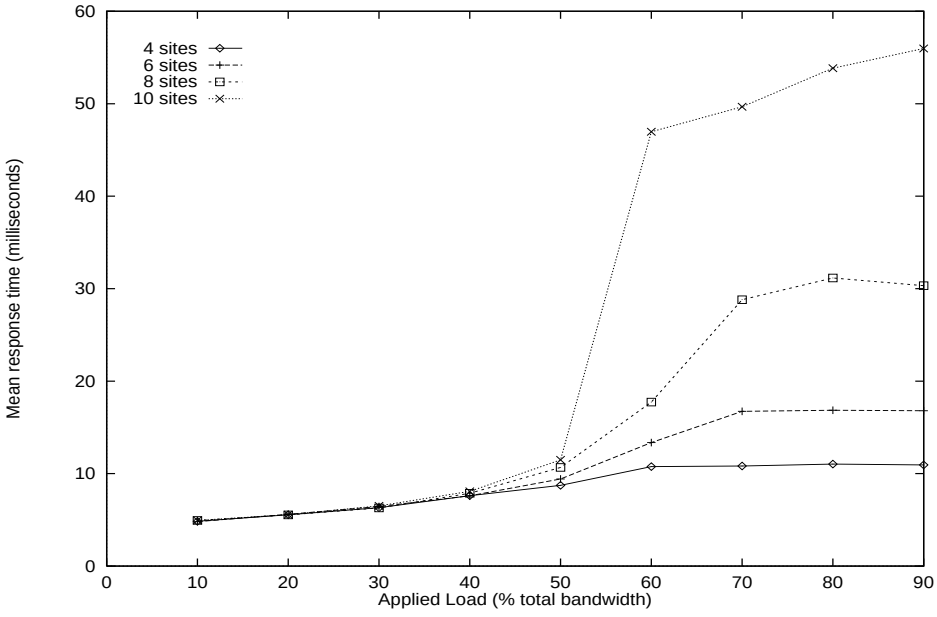

Figure 15: Response Time for High Priority Traffic for Modified Synchronous Method (20% Bursty Traffic)

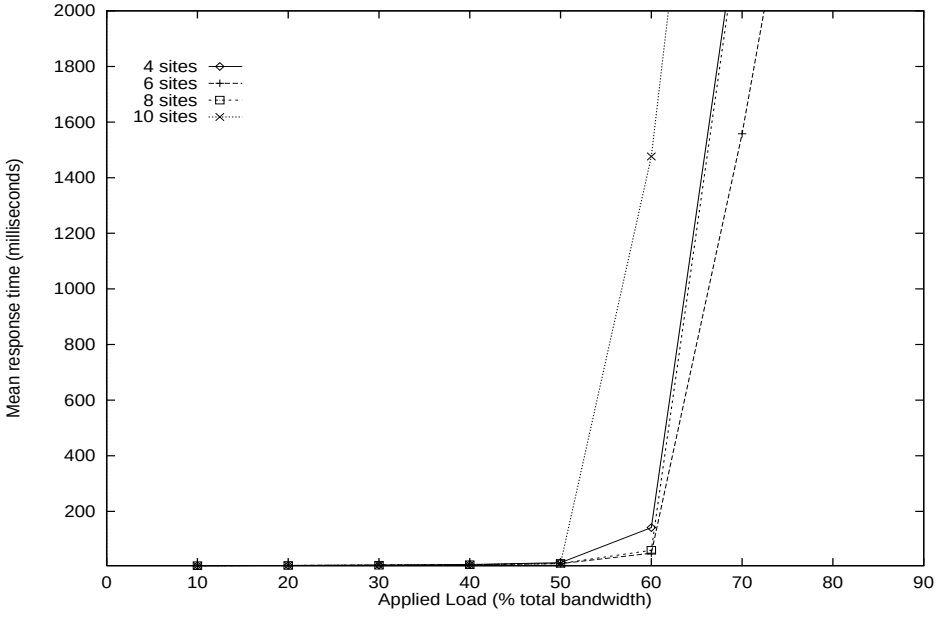

Figure 16: Response Time for Low Priority Traffic for Modified Synchronous Method (20% Bursty Traffic)

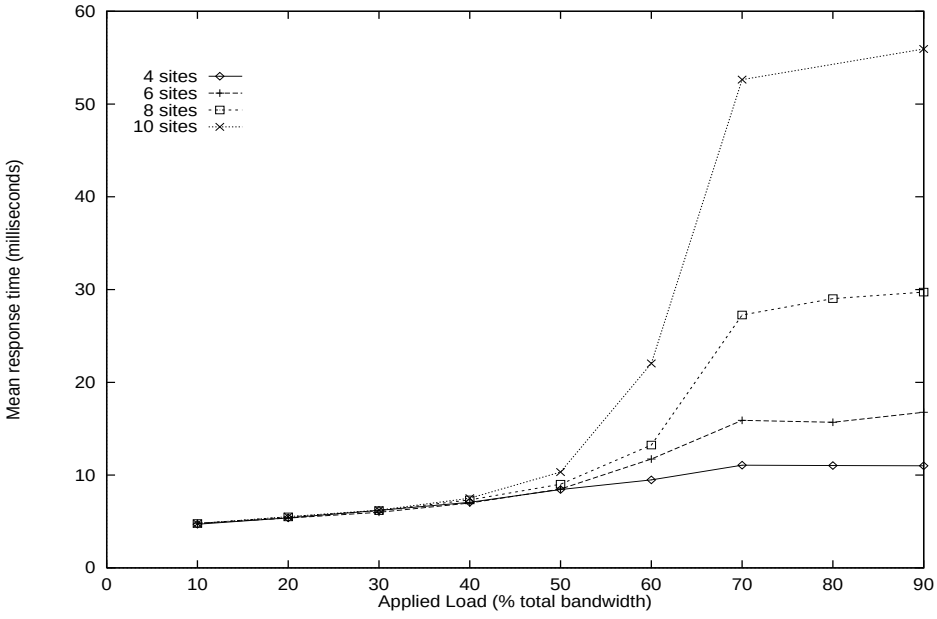

Figure 17: Response Time for High Priority Traffic for Modified Synchronous Method (30% Bursty Traffic)

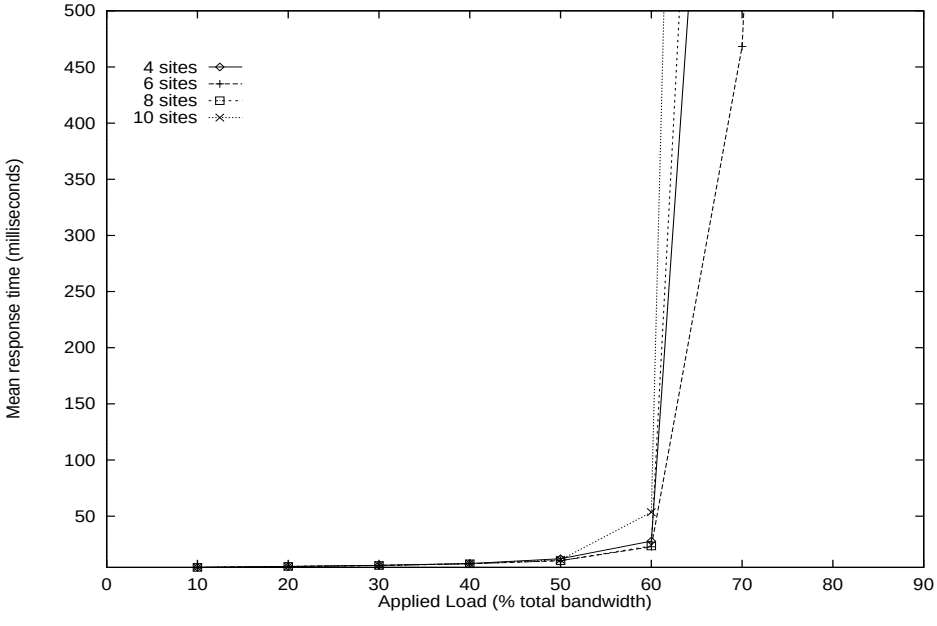

Figure 18: Response Time for Low Priority Traffic for Modified Synchronous Method (30% Bursty Traffic)

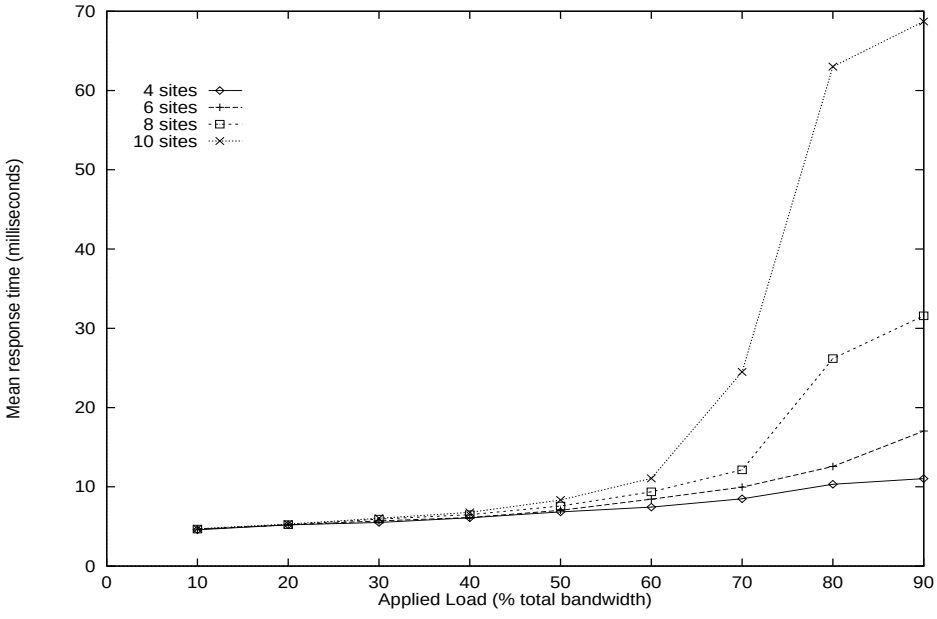

Figure 19: Response Time for High Priority Traffic for Modified Synchronous Method (50% Bursty Traffic)

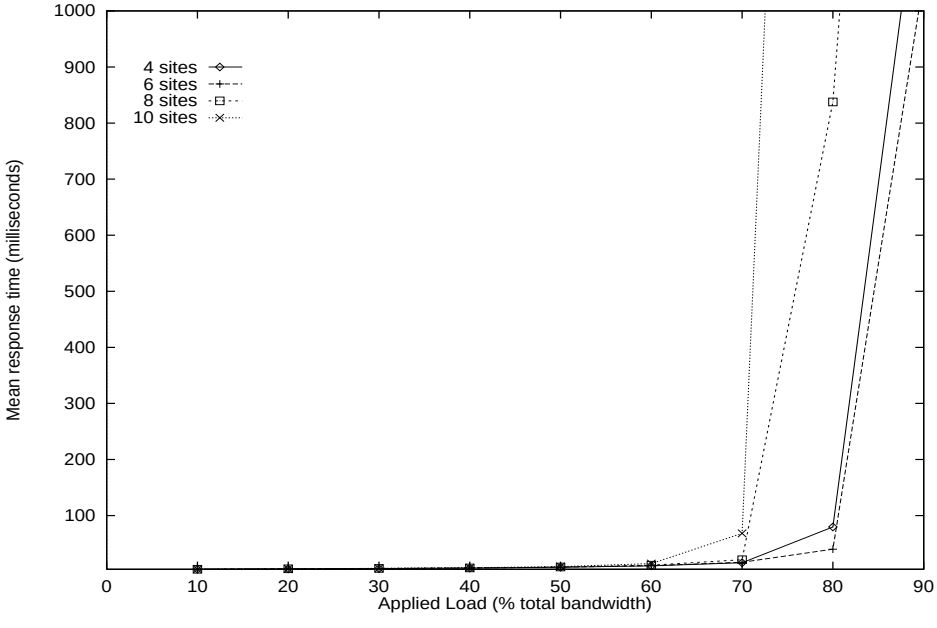

Figure 20: Response Time for Low Priority Traffic for Modified Synchronous Method (50% Bursty Traffic)

2.6.5 Asynchronous Method Simulation

Figures 21, 23, and 25 show the response time for high priority traffic and Figures 22, 24, and 26 show the response time for low priority traffic for 2%, 20%, and 50% bursty traffic for a uniform load distribution, respectively. Figures 27 and 29 show the response time for high priority traffic and Figures 28 and 30 show the response time for low priority traffic for 20% and 50% bursty traffic for a non-uniform load distribution, respectively. The simulations were run for 4, 6, 8, 10, and 12 sites, for an applied load ranging from 10% to 90% of the total network bandwidth (10 Mbits/sec). The following observations were made.

Observation 1: The asynchronous method has a low response time, although the maximum sustained throughput for low amounts of bursty traffic can be worse than the basic Ethernet.

Figure 21 and Figure 22 show the response time of the asynchronous method for 2% bursty traffic for high and low priority data, respectively. The response time tends to be low, however, it becomes unstable quickly, between 30 - 40% applied load, while only the 10 and 12 site case in the basic Ethernet become unstable that quickly.

Figure 23 and Figure 24 show the response time of the asynchronous method for 20% bursty traffic for high and low priority data, respectively. The network tends to be stable

until around 40 - 50% utilization. In the basic Ethernet simulation, only the 4 and 6 site cases were stable beyond 50% utilization.

Figure 25 and Figure 26 show the response time of the asynchronous method for 50% bursty traffic for high and low priority data, respectively.. The network now becomes unstable between 40 - 60% applied load, and the high priority data has a slightly lower response time than the low priority data.

Observation 2: For uniform traffic, the asynchronous method has a lower maximum sustained throughput than the synchronous and modified synchronous methods.

For 20% bursty traffic, the maximum sustained throughputs are approximately 70%, 65%, and 60% for the synchronous, modified synchronous and asynchronous methods, respectively. For 50% bursty traffic, the values are 80%, 75%, and 65% for the synchronous, modified synchronous and asynchronous methods, respectively. The maximum sustained throughput for the asynchronous method is lower than the previous methods due to collisions. Because the priority information competes with the data for bandwidth, more bandwidth is wasted due to collisions in this method, resulting in a lower maximum sustained throughput.

Observation 3: The maximum sustained bandwidth increases as the percentage of bursty traffic increases.

As the percent of bursty traffic increases, the maximum sustained throughput increases, similar to the behavior seen in the synchronous and modified synchronous methods.

Once again, as the percentage of bursty traffic increases, the percentage of non-bursty traffic decreases, which decreases the competition for bandwidth among low priority data and allows a higher sustained throughput for low priority data. The reason there is an increase in the maximum sustained throughput for high priority data is that the number of collisions decreases as the amount of bursty traffic increases. *PRIO-IS* messages are transmitted when a data packet is created if there is nothing ahead of it in the *pending* queue, and it can collide with the data packets from other sites. For a given applied load, bursty traffic tends to arrive in small groups at a site close together, with a low overlap between bursts at different sites, while Poisson arrivals are distributed over a much larger period of time and have a greater chance of overlapping the arrivals at other sites. A *PRIO-IS* message can not collide with a data message from the same site, since a message can not interfere with the other messages in its burst. So, the probability for a collision is lower with bursty traffic than with Poisson traffic. Even though the clustering effect of the bursty traffic improves the maximum bandwidth, the maximum bandwidth for the asynchronous method is still less than that of the synchronous and modified asynchronous methods for uniform traffic due to the greater number of collisions.

Observation 4: The maximum sustained throughput is greater for non-uniform load distributions. Figures 27-30 have a maximum sustained throughput of 50-70%, as compared to 40-60% for the uniform load distribution cases. In the non-uniform load cases, 50% of the bandwidth came from one site and the other 50% came from the remaining sites. There are fewer sites transmitting in the non-uniform cases, so there is a

lower probability of collision and less bandwidth wasted by collisions. Therefore, a greater maximum throughput can be sustained.

In general, the performance for the asynchronous method is method is very dependent upon the type of traffic applied. For uniform traffic, the asynchronous method does not perform as well as the synchronous method. This is due to the number of collisions in the asynchronous method, since the priority information competes with the data for the bandwidth. However, for a highly non-uniform data traffic, the asynchronous method has good performance. Since there are no priority information exchange rounds, this method can quickly respond to high priority messages with a low response time and a high maximum sustained throughput.

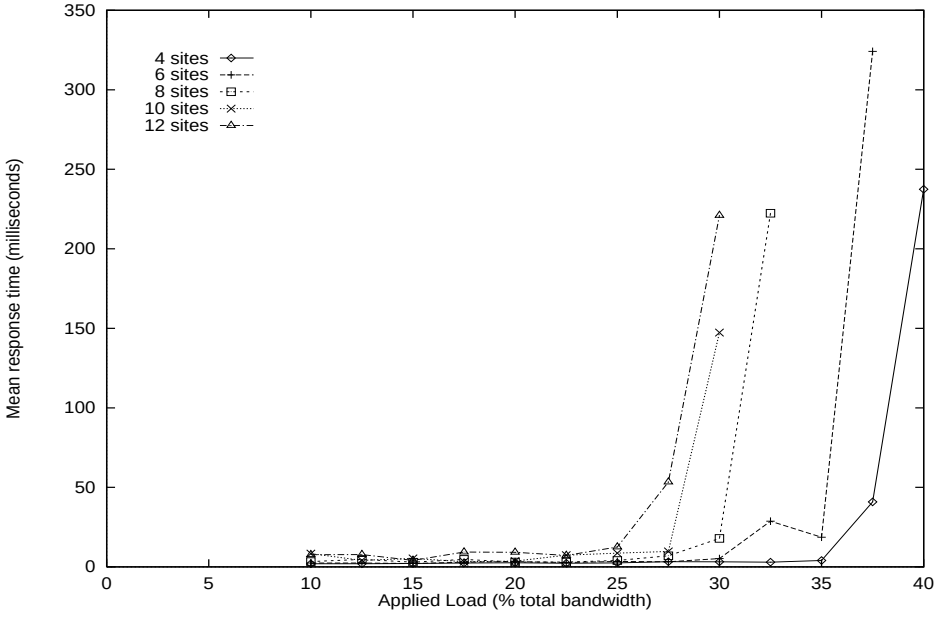

Figure 21: Response Time for High Priority Data for Asynchronous Method (2% Bursty Traffic)

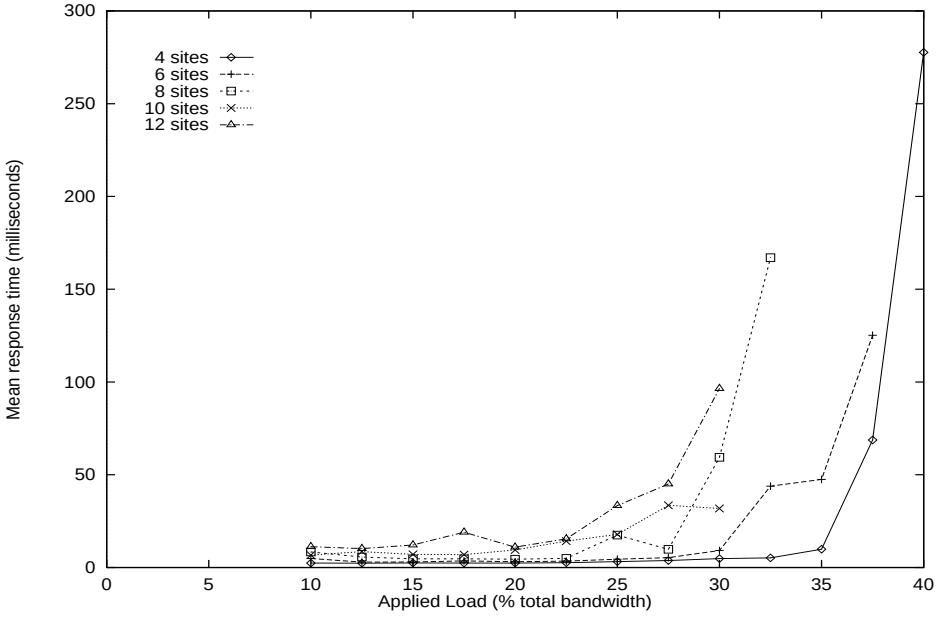

Figure 22: Response Time for Low Priority Traffic for Asynchronous Method (2% Bursty Traffic)

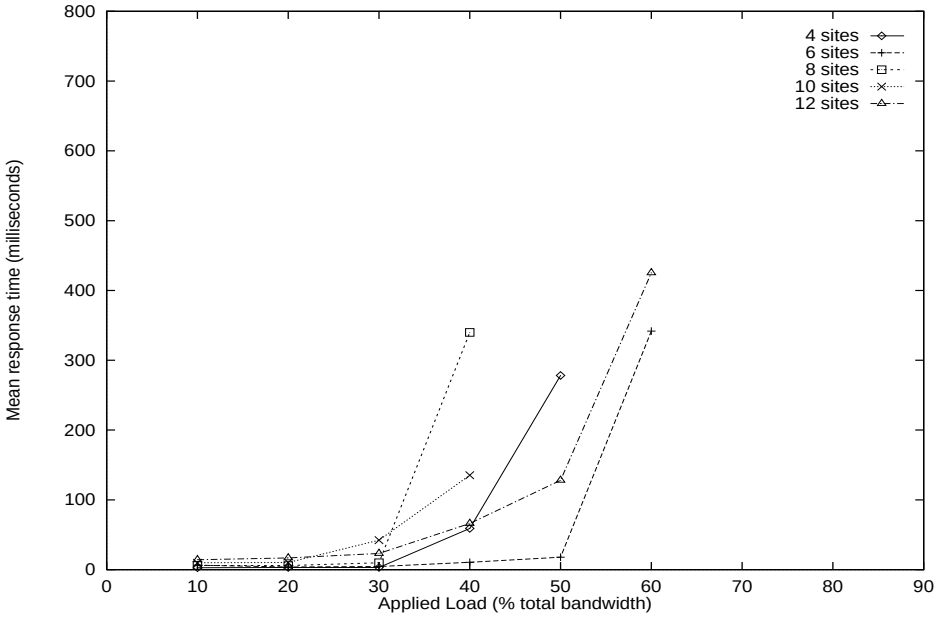

Figure 23: Response Time for Asynchronous Method for High Priority Traffic (20% Bursty Traffic)

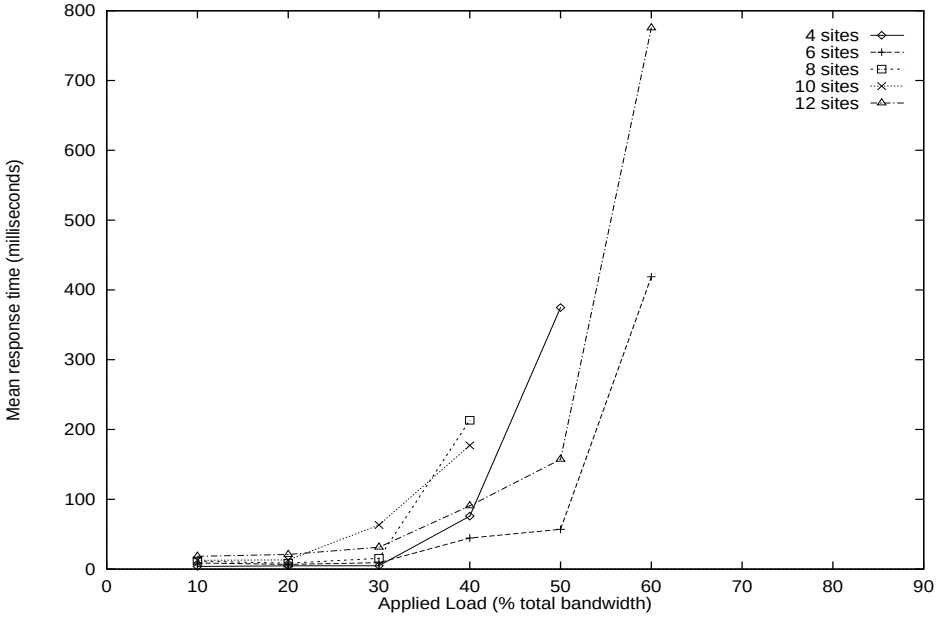

Figure 24: Response Time for Low Priority Traffic for Asynchronous Method (20% Bursty Traffic)

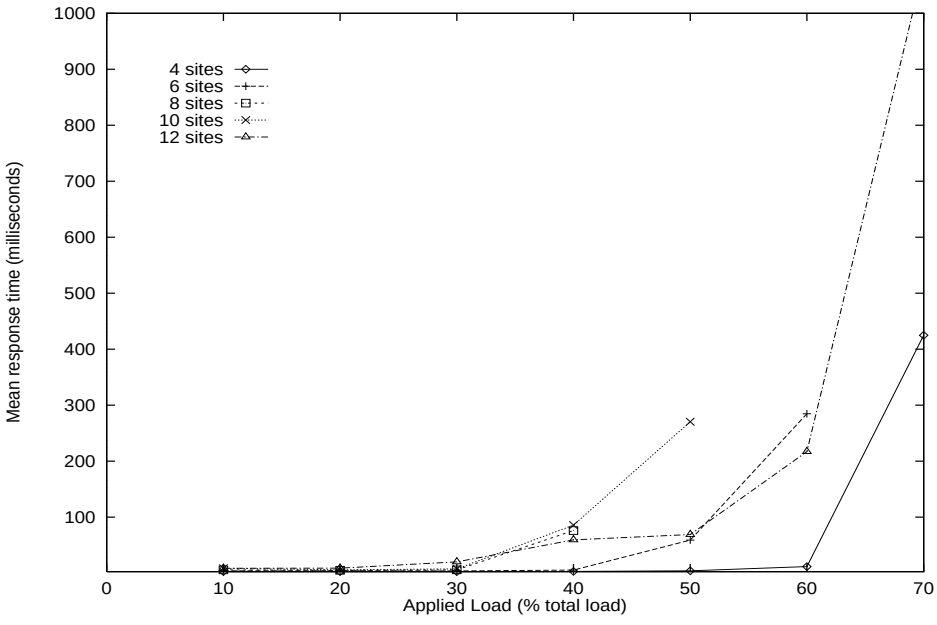

Figure 25: Response Time for High Priority Traffic for Asynchronous Method (50% Bursty Traffic)

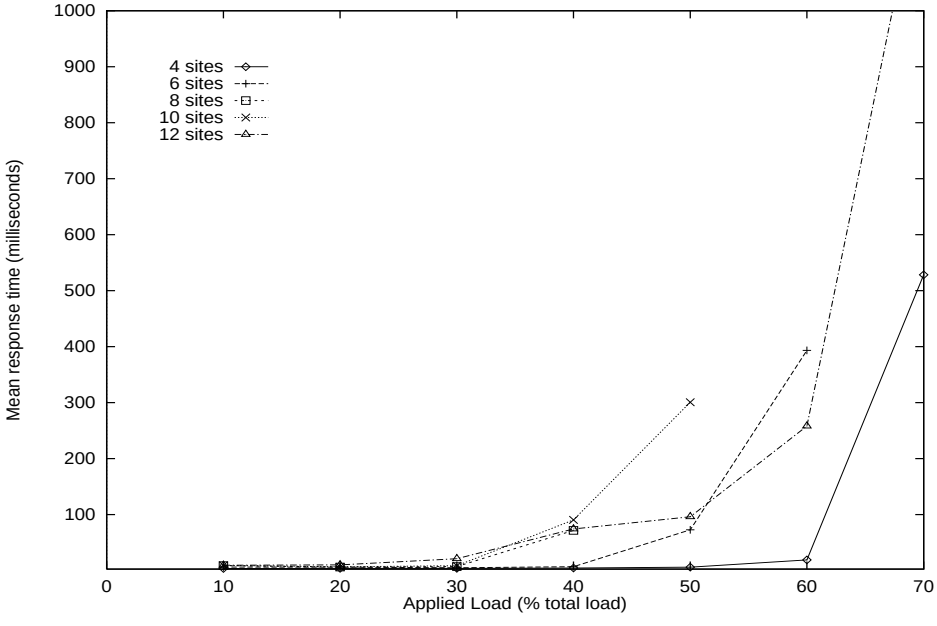

Figure 26: Response Time for Low Priority Traffic for Asynchronous Method (50% Bursty Traffic)

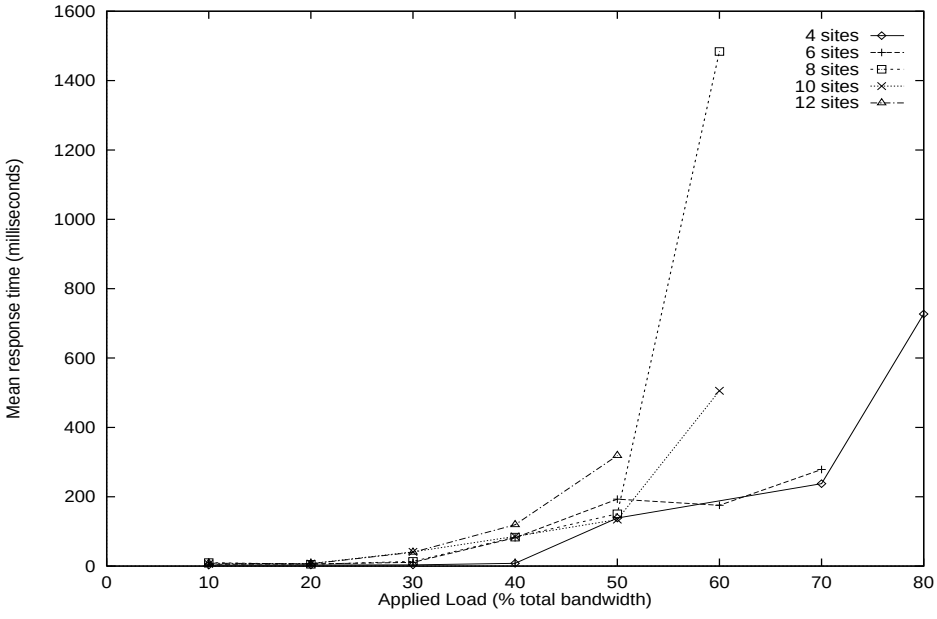

Figure 27: Response Time for High Priority Traffic for Asynchronous Method (20% Bursty Traffic, Non-Uniform Load)

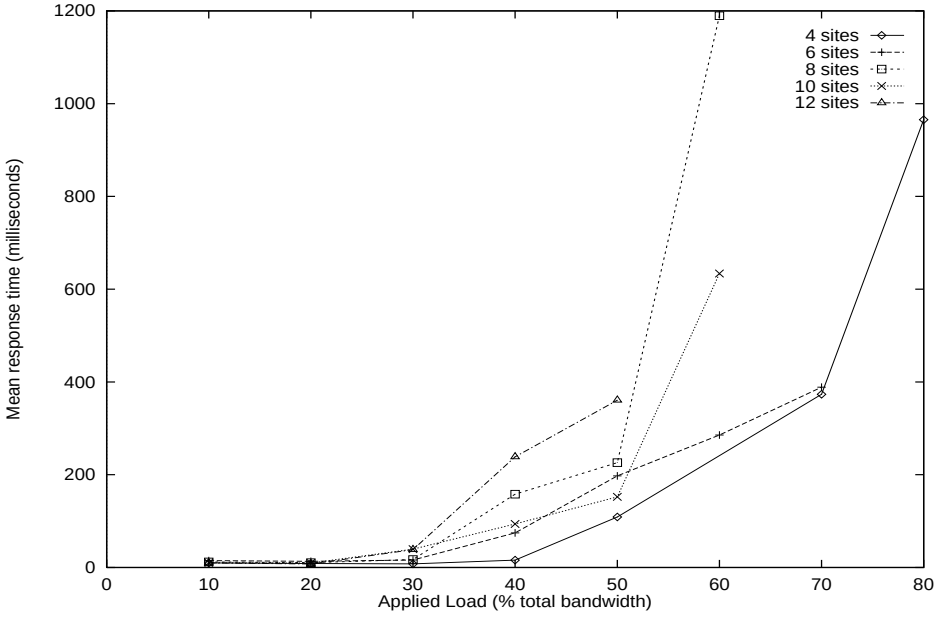

Figure 28: Response Time for Low Priority Traffic for Asynchronous Method (20% Bursty Traffic, Non-Uniform Load)

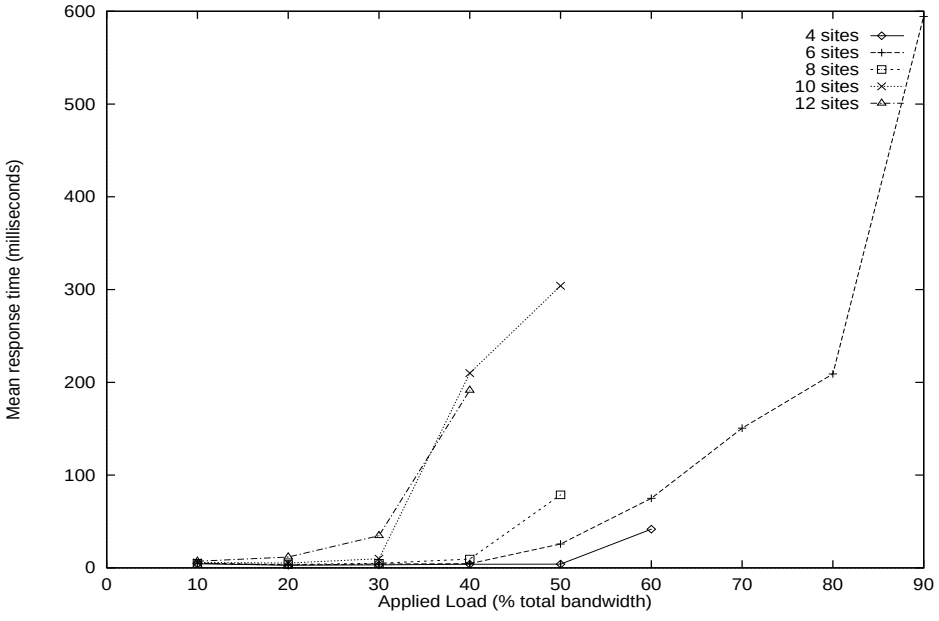

Figure 29: Response Time for High Priority Data for Asynchronous Method (50% Bursty Traffic, Non-Uniform Load)

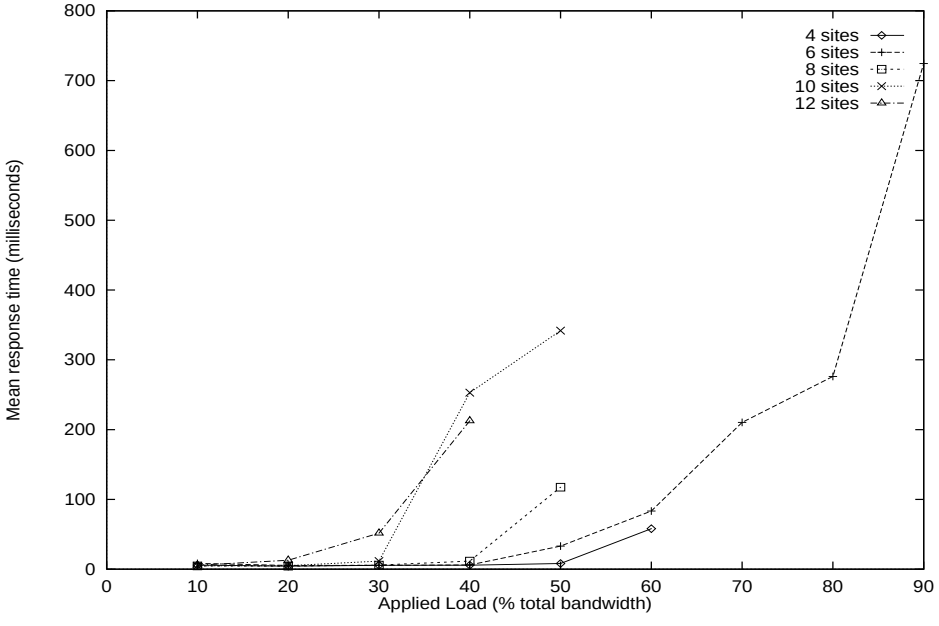

Figure 30: Response Time for Low Priority Data for Asynchronous Method (50% Bursty Traffic, Non-Uniform Load)

2.7 Summary

Traditional protocols for local area networks, e.g., an Ethernet, were designed to handle data traffics for which variable delay is acceptable. These protocols are not suitable for multimedia traffic which requires mostly a constant delay in packet deliveries as well as fast (or real-time) delivery of packets.

In this chapter, we presented three protocols for priority transmission of data on Ethernets: (i) the Synchronous Method, which virtually eliminates packet collisions by having a well-defined order for data transmission that is established during the priority information exchange phase in which nodes have a fixed *a priori* order to transmit their priority data, (ii) the Modified Synchronous Method, which relaxes the ordering constraint during the priority exchange phase to improve efficiency under heterogeneous traffic, and (iii) the Asynchronous Method, which eliminates the priority information exchange phase to reduce the time high-priority messages wait until they can be transmitted. By giving multimedia data priority over regular data, these priority protocols can be used to deliver multimedia data over the Ethernet in a timely manner with low variance in delay. These protocols will allow a higher throughput for multimedia data with fewer packets dropped (due to missed deadlines).

Since multimedia is becoming important in a large number of business, government, and academic organizations and since an Ethernet is a widely used local area network in most of these organizations, priority protocols for Ethernets will enable these organizations to

integrate multimedia capabilities into their existing systems with little change. Other alternatives (such as ATM or other high-speed networks) available to these organizations to integrate multimedia capabilities into their systems will cause an expensive hardware overhaul. Besides multimedia applications, these protocols can be used in applications (e.g., control systems, industrial process control, automated manufacturing, air-traffic control, program trading, etc.) where certain tasks must meet deadlines. Critical tasks can be assigned higher priority to ensure better service to them. In addition, these protocols can be used for priority scheduling in a general computing system to give priority to a user because of project deadlines, importance of the user, or because a user pays higher fees for better service, etc.

We identified the functions of new (intelligent) network interfaces that will perform most of the priority transmission related chores in hardware. We presented a qualitative analysis comparing the three methods of disseminating priority information based on several different criteria. And finally, we presented the results of a simulation study of the proposed protocols.

The simulation results show that the proposed methods are sensitive to the type of traffic that is applied. The synchronous method has the best performance due to the lack of collisions. It consistently had a lower response time for both high and low priority data, and was able to sustain a higher maximum throughput than an Ethernet only with CSMA/CD. The modified synchronous method had a good response time and maximum sustained throughput, although the number of collisions was greater than synchronous

method. The asynchronous method's performance varied, depending on the load and number of sites. In some cases, it had better performance than a basic Ethernet, and in other cases it did not. For non-uniform traffic, the modified synchronous and asynchronous methods should have a better response time and maximum sustained throughput than the synchronous method.

3. Chapter 3

CHAPTER III

Real-Time Causal Message Ordering in Multimedia Systems

3.1 Introduction

Multimedia systems are characterized by the use of several different methods of presenting data, such as voice, still and motion video, music, and text, all of which are real-time in nature. The applications send streams of data that must be used within a time interval because the data is useless after that interval [17]. Missed data can be discarded and only represents a degradation in the quality of the playback or presentation that should be minimized.

When sites send messages to each other through a communication network, all sites that receive the messages should see them in a consistent order. A *causal order* relation between two messages, say *m1* and *m2,* exist if either a) both messages were sent from the same site and *m1* was sent before *m2* or, b) *m2* was sent by a site after that site received *m1*. Causal ordering of messages implies that there is the same causal relation

between message send events and the corresponding message receive events. When a site sends messages to other sites, they might not be received in the same order by all sites. By imposing causal ordering on messages, all messages are ordered in a consistent way at all sites.

As an example, consider a multimedia conference call that is taking place at several different sites where each input is multicast to other sites. For correctness the transmitted data (i.e., speech or video) should satisfy causal order and should be seen (and heard) in the same order by all sites, especially if one data stream is in the response to another. In addition, messages have real-time constraints; if a message can not be delivered, i.e., played, within a certain time, it must be discarded. In a multimedia conference call, where each "speaker" is heard by multiple sites, the messages need to have their cause and effect relations preserved. A causal order algorithm is required to preserve the causal order of the messages.

Birman-Schiper-Stephenson [7] and Schiper-Eggli-Sandoz [38] present algorithms to implement causal ordering of messages. These algorithms use the concept of logical clocks and vector timestamps [29] to provide information as to which messages have been sent before a message. Logical clocks are a good tool to capture casual ordering of messages at a logical level. However, logical clocks increment only when an event happens, such as the transmission or reception of a message and have little correlation to physical time. Real-time systems require the notion of physical time because of the constraints imposed by the deadlines on the data. Therefore, while logical clocks are

appropriate for causal ordering in non-real-time systems, they are not appropriate for real-time systems.

While there is no physical global clock in a distributed system, it is possible to keep all of the local clocks loosely synchronized [19, 31], which means that a bound can be placed on the clock drift. In addition, the clock drift can be kept small enough so that local clocks at each site can be used for global physical time.

Multimedia data has a limited lifetime after which the data can not be used. A message that arrives out of order should be held until all messages that were sent before it have expired or are received. However, a message should not be held beyond its lifetime while waiting for messages that were sent before it, because it will be discarded anyway. We can assume that all data of the same type (e.g., video or audio) have the same life-span. So, if message *m2* is being held until message *m1* arrives, *m1* must expire before *m2* expires. After *m1* expires, there is no longer any reason to hold *m2*, because even if *m1* arrives, it will be discarded.

So, because data is only useful during a certain interval, the dependency on previous messages applies only for a limited interval. This time-constrained dependency is referred to as "delta-causality" [49] and is defined formally in the next section.

In addition to limiting the interval of the dependency, the number of messages in a dependency chain is limited. For every message in a dependency chain, the dependency information must pass through another site other than the source and destination. Every

message induces a delay due to processing at the source and destination sites as well as propagation delay. After passing through a few sites, the delays can be comparable to the lifetime of a message. By the time the message at the end of a chain arrives, the messages at the beginning would have been processed or discarded. Since chains of length three or longer are typically longer than the lifetime of a message, and do not typically occur in practice, we concentrate on chains of length two. This case is called the "triangle inequality," and can be efficiently detected.

This chapter presents an algorithm that solves the problem of delivering messages containing multimedia data with real-time constraints in causal order. It attempts to deliver messages in the same order at all sites while trying to minimize the number of messages past their deadlines that must be discarded. This chapter focuses on handling "triangle inequality" causal order violations and argues that handling this case is sufficient for this algorithm.

The chapter is organized as follows. Section 2 presents formal definitions of terms and the problem statement. Section 3 presents the algorithm to ensure that messages are delivered in order and a proof of correctness. Section 4 discusses performance, loosely synchronized clocks, message size and complexity, and argues that handling longer paths is expensive. Finally, Section 5 summarizes this chapter.

3.2 Preliminaries

This section formally defines causal order and delta causal order and concludes with the problem statement.

3.2.1 Definitions

The terms "delivered" and "received" are used throughout this dissertation. It is important to note the difference between them. A message is said to be received when it reaches the destination site from the communication network. It is said to be delivered when the communication level software passes the message on to the application software. While a message has been received at a site, it may not necessarily be deliverable to the application due to dependencies. A received message is held in a queue until all missing messages that it depends upon have arrived or expired.

The next two definitions provide a framework for relations between messages.

Definition 1: "$\longrightarrow$" ("causal dependency") is defined in [25] such that $m1 \longrightarrow m2$ if and only if:

1. *m1* and *m2* are messages that were sent from the same site and *m1* was sent before *m2*, or
2. *m2* was sent by site *i* after *m1* was received by site *i*.

Note that in Figure 31, $m0 \longrightarrow m1$ and $m2 \longrightarrow m3$.

Definition 2: Causal ordering, denoted by "$\xrightarrow{*}$", is defined as the transitive closure of the "$\longrightarrow$" relation for the reception of messages at a site [38].

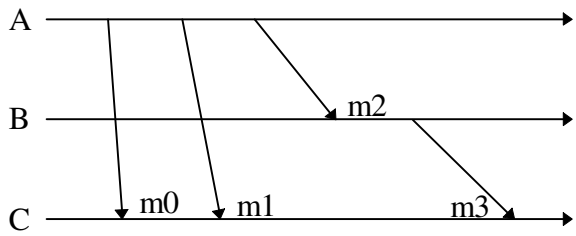

Figure 31: Causal Ordering

Causal ordering implies that there is the same causal relation between message send events and the corresponding message receive events. In Figure 31, site C receives messages in causal order, because it receives *m0* before *m1*, and *m1* before *m3*. Message *m2* "bridges the gap" of the dependency information from *m1* to *m3*. By the definition of causal ordering, there can be an arbitrary number of messages involved in a chain of dependencies, since there is no notion of a deadline or expiration time to the data. This issue is addressed subsequently.

Disparity in the speed of communication links as well as network congestion can contribute to causal order violations. The simplest form of a violation of causal ordering is due to the "triangle inequality." This occurs when a message that passes through a site takes less time to travel than one that is sent directly to the same destination. For example, in Figure 32, let X, Y, and, Z represent the time required to send a message

from A to B, B to C, and A to C, respectively. If Z is greater than the sum of X and Y, then messages arrive at site C out of causal order. Figure 33 shows how the triangle inequality could cause a violation of causal ordering. A simple example of the effects of causal order violations is an audio conference in which a question is asked by one participant and is answered by another, but they are heard in the opposite order by a third participant.

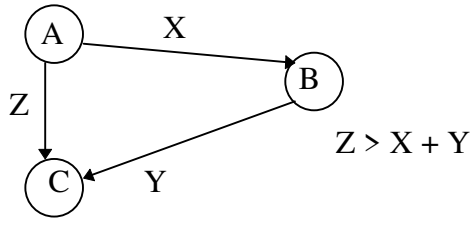

Figure 32: Triangle Inequality

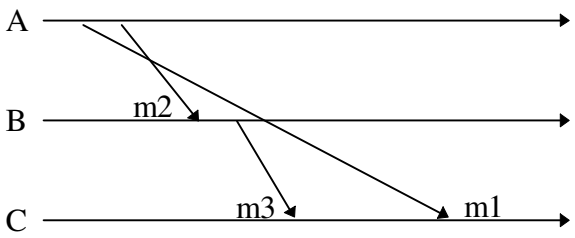

Figure 33: Causal Ordering Violated

Definition 3: "$\xrightarrow{\Delta}$" ("delta causal order") is defined in [49] such that $m1 \xrightarrow{\Delta} m2$ if and only if:

1. $m1 \xrightarrow{*} m2$

   AND

2. *m1* is sent at most Δ time units before *m2*.

Figure 34 shows how delta-causality can be observed even if causal ordering is not observed. The deadline of a message is defined as delta time units after the message is sent. So, messages expire delta time units after they are sent. Because *m1* is sent more than Δ time units before *m3* is sent, there is no delta causality relation between *m1* and *m3*, so *m3* is not queued when it arrives. If *m3* is held until *m1* arrives, *m3* as well as *m1* are discarded since the data in both messages has expired. Thus, in this case the data in message *m3* is discarded along with *m1*, whereas the information in *m3* could have been used.

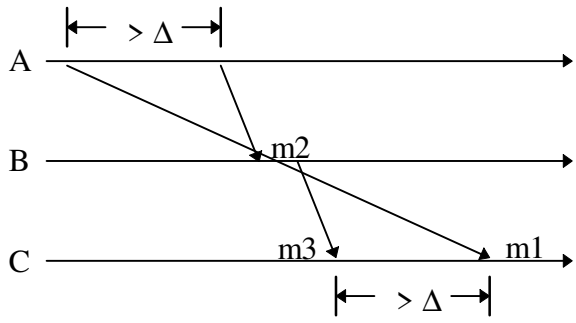

Figure 34: Delta Causal Ordering Observed (m1 Discarded)

Delta-causality is reasonable to use for multimedia data because of the limited lifespan of the data. There is no reason to maintain a causal order relation with data that has exceeded its lifespan. Out of date messages can be detected with the use of physical timestamps. In the case of a long pause by a participant spent "thinking" about a message before a reply is sent, the communication medium handles message ordering; if causal order were violated, the original message will have long since expired anyway. The causal ordering relations due to messages that have passed through many sites are not as significant, because violations in that ordering require the late message to take longer to go from its source to its destination than it takes for several messages to travel through several sites. By limiting the size of the dependency chains to two, the message overhead associated of the algorithm presented in this chapter can be drastically reduced. This is why the algorithm presented in this chapter is designed to handle causal order violations due to only the triangle inequality. In general, it should be sufficient for multimedia applications to handle only this case.

Figure 35 shows how higher order causality violations, such as a transitive relation spanning four sites, do not violate delta causality. Causal ordering forces *m4* to be held until *m1* is received, even if it is known that *m1* is past its deadline and is useless. Delta-causality does not impose that constraint however, so *m4* is delivered before its deadline, even though *m1* has not yet arrived. When *m1* eventually arrives, it is discarded.

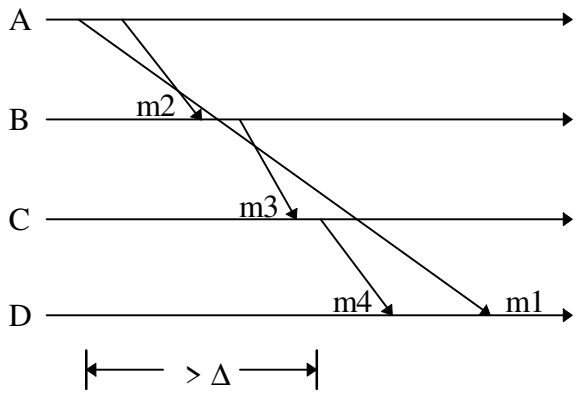

Figure 35: No Delta Causal Relation Between m1 and m4

3.2.2 Problem Statement and Objective

In a multimedia system with real-time data, messages must be ordered such that all messages are delivered to the sites in the same causal order. Congestion and delays in the network can cause messages to be delivered to the destination sites out of causal order. The problem is to design an algorithm that preserves causal message ordering to

the application. Since the data is real-time in nature, it is sufficient to ensure delta causal order, since the data and their causal order relations have a limited lifespan. As a progress requirement, if a site receives a message before its deadline, the message should be delivered to the application layer before its deadline.

This chapter presents an algorithm that ensures that multimedia messages with real-time data are delivered in delta-causal order and argues that there cannot exist dependencies of length greater than two in this algorithm. The algorithm handles triangle inequality violations and ignores higher order violations. In addition, it is designed to minimize the number of messages that are discarded. Once a message is received by a site, it will be delivered rather than discarded, provided it arrives before its deadline. Messages with dependencies on other messages that have not arrived are held until either the missing messages arrive or have passed their deadlines. In either case, the held message is delivered.

The algorithm initially assumes that all sites have perfectly synchronized clocks. This assumption is relaxed later, such that all sites have loosely synchronized clocks with a clock drift bound by some known value.

3.3 The Delta Causal Order Algorithm

In this section we present the algorithm. The relevant data structures and variables are introduced and a proof of correctness is presented.

3.3.1 Data Structures and Notations

- Each site *i* has an NxN dependency matrix denoted Dep_i, where element (*x*, *y*) of the matrix represents the timestamp of the last message site *x* sent to site *y*, as far as site *i* knows.

- Each site has a physical clock that is constantly updated by an underlying clock synchronization algorithm and is denoted by the variable *current_time*.

- Each site maintains a priority queue of messages that have arrived but have not yet been delivered, sorted by expiration time.

- The variable *min_wait* denotes the amount of time necessary to hold a message to compensate for the network transit time. It is used to force causal dependencies of length 3 or more to be out of date. A typical value is 30-35 milliseconds for LANs.

- Δ represents the time the data in a message is useful and is defined to be < 3 * min_wait. An upper bound for Δ for audio is 100 milliseconds, after which the sound quality degrades [49].

3.3.2 Overview

The algorithm adds two vectors, denoted *vect1* and *vect2*, to messages. When site *i* sends a message to site *j*, the first vector, *vect1*, is the i^{th} row of *i*'s dependency matrix and denotes the timestamp of the last message site *i* sent to other sites (*i*'s vector timestamp). The second vector, *vect2*, is the j^{th} column of *i*'s dependency matrix and

denotes the timestamp of the last message sent to *j* by other sites. Inclusion of the two vectors allows causal order violations due to the triangle inequality to be detected because out of order messages will have a timestamp greater than the corresponding timestamp in the destination site's matrix. Messages that have no causal relation between them are defined as being concurrent. By holding all messages longer than the network latency, almost all concurrent messages are identically ordered at all sites. This ensures that a message sent earlier than another but delayed due to network delays is still delivered in the proper order. The timestamps are used to determine the delivery order of concurrent messages whose arrivals are separated by at most min_wait. Since every site that receives a message gets the same timestamp value and uses the same ordering algorithm, all sites order concurrent messages in the same way.

3.3.3 The Algorithm

Sending A Message From Site *i* To *j*

```
send_message ( msg, j) {
        Dep_i[i,j] := current_time
        /* row i of dependency matrix */
         ∀x, (msg.vect1[x] := Dep_i[i,x])
        /* column j of dependency matrix */
         ∀x, (msg.vect2[x] := Dep_i[x,j])
        Send message to site j.
}       /* end function */
```

Receiving A Message From Site *i* At Site *j*

```
receive_message (msg, j) {
        IF (msg.vect1[j] + Δ > current_time) THEN
                /* discard if it is past its deadline */
                Discard message.
                RETURN
        ENDIF
        /* insure the message is not delivered until at
         least min_wait time units after being sent */
        wait (MAX(0, (msg.vect1[j] +  min_wait - current_time)))   /* Rule 1 */
        check_and_deliver (msg, j)
}       /* end function */


check_and_deliver (msg , j) {
        /* compare the message with the Dep matrix
                of the destination site */
        IF (dependency_exists(msg, Dep_j)) THEN                    /* Rule 2 */
                Put message msg in queue.
                RETURN
        ENDIF
        /* deliver message with no dependencies */
        deliver_msg(msg)
        /* check other messages in queue and deliver any that
           do not have dependencies, since other messages may
            have been waiting for this message. */
        FOREACH x in (message-queue)
```

```
                IF (¬(DependencyExists(x, Dep_j)) THEN
                        DeliverMessage(x)
                ENDIF
        ENDFOREACH
}       /* end function */


deliver_msg ( msg ) {
        /* deliver message and merge sending sites
         dependency vector into local matrix */
        Remove message msg from queue.
        FOR x := 1 to N DO
                IF (msg.vect1[x] > Dep_j[i, x]) THEN                    /* Rule 3 */
                        Dep_j[i, x] := msg.vect1[x]
                ENDIF
        ENDFOR
}       /* end function */

Boolean dependency_exists (msg, matrix) {
        /* a dependency is represented by a timestamp in
         the message that is later than the corresponding one
         in the dependency matrix, provided that that
         timestamp is no older than by delta (i.e., missing
         message has not expired) */
        FOR x := 1 to N DO
                IF ((msg.vect2[x] > matrix[x, j]) AND
                        ((current_time - msg.vect2[x]) < Δ)
                THEN
```

```
                    RETURN true
                ENDIF
            ENDFOR
            RETURN false
}       /* end function */
```

When a message is put in the queue, an asynchronous timer is set at MAX (msg.vect2[x] + Δ) for all x. When the timer expires, the system checks if the message has any dependencies by calling the function dependency_exists(). The message is discarded if so and delivered otherwise. All other messages in the queue are checked for dependencies because undelivered messages may be delivered at that point, if the queued message depends upon the discarded or delivered message. Once the message is delivered (or discarded), the other messages on its dependency chain can be delivered.

3.3.4 Correctness

Definition 4: "$\xrightarrow{2HOP}$" ("two-hop delta causal dependency") is defined such that $m1 \xrightarrow{2HOP} m2$ if and only if:

1. $m1 \longrightarrow m2$ or
2. $\exists\ m3$ such that $m1 \longrightarrow m2 \wedge m3 \longrightarrow m2$ and $m1$ and $m2$ are sent to the same destination

   and

3. $| t_{m1} - t_{m2} | <= \Delta$, where t_{m1} and t_{m2} are the times *m1* and *m2* were sent, respectively.

In Figure 31, $m0 \xrightarrow{2HOP} m1$ and $m1 \xrightarrow{2HOP} m3$. Either the messages originates at the same site or there exists a single message between those two sites. In addition, they must be sent within Δ time units. If *m1* and *m2* are sent greater than Δ time units apart, then there is no delta-causal relation between them. If *m1* arrives after *m2*, which would be more than Δ time units after *m1* was sent, then the information in *m1* is useless and *m1* would be discarded.

Lemma 1: The algorithm guarantees that 2Hop order is preserved in message delivery.

Proof: Assume $m1 \xrightarrow{2HOP} m2$. By the definition of 2Hop there are two cases, the same site case and the one hop case. In the same site case, *m1* and *m2* are sent from the same site and *m1* is sent before *m2*. In the one hop case, there is one message that is one "hop" between *m1* and *m2*, such that it has *m1*'s source as its source and *m2*'s source as its destination. We now show message order is preserved for both cases.

Same Site Case:

Assume *m1* and *m2* are sent from *i* to *j*. If they were sent consecutively, then the timestamp of *m1* is stored at position (*i*, *j*) of site *i*'s dependency matrix and this value is sent to site *j* in *vect1* and *vect2* of *m2*. If *m2* arrives before *m1*, by Rule 2 in

check_and_deliver(), *m2* is held in a queue until either *m1* arrives or *m1* is past its deadline, implying that there is no longer a dependence on *m1*. Any message sent between *m1* and *m2* to a site other than *j* has no 2Hop relation to *m1* or *m2* and can be ignored. If site *i* sends messages to site *j* between *m1* and *m2*, then it can be shown by induction that each pair of consecutive messages are properly ordered and therefore the entire sequence is properly ordered.

One Hop Case:

Assume *m1* and *m2* are sent to site *k* from two different sites, *i* and *j* respectively. Then by the definition of 2Hop, there must exist a message, *m3*, such that *m3* was sent from site *i* after *m1* was sent, and received by site *j* before *m2* was sent. The j^{th} entry of *vect1* (i.e., vect1[j]) of *m3* contains the timestamp for *m1*. This is incorporated into the dependency matrix at site *j* and is passed to site *k* as vect2[j] of *m2*, by Rule 3 of deliver_msg(). If *m2* arrives before *m1*, site *k* detects the dependency on *m1* from *vect2* of *m2* and *m2* is held until either *m1* arrives or *m1* is past its deadline, in which case there no longer is a dependency between *m2* and *m1*. Therefore, *m1* and *m2* are delivered in the proper order and the algorithm delivers messages in 2Hop order. □

Lemma 2: There can be no delta-causal dependencies of length greater than two.

Proof: Assume a transitive dependency of length three exists between site *i* and site *j*. This implies that there has been a delay of > 3 * *min_wait*, since Rule 1 of the algorithm insures that a delay of *min_wait* is incurred for each hop. Δ is defined to be < 3 *

min_wait, therefore no delta-causal dependency exists. Since the delays increase with the length of the dependency chain and no delta-causal dependency of length three can exist, there can be no delta-causal dependency of length greater than two. □

Lemma 3: Messages are delivered in delta-causal order.

Proof: If the transitive dependencies among messages are of length two, then from Lemma 1 the messages are delivered in delta-causal order since 2Hop is equivalent to delta causality restricted to transitive dependencies of length two.

If the transitive dependency is of length greater than two, then by Lemma 2, no delta-causal relation exists. Therefore, the algorithm delivers messages in delta-causal order. □

If a message arrives out of order, the algorithm holds the out of order message until the missing message either arrives or is past its deadline. Since all sites are running the same algorithm, all messages arriving at a site before their deadlines are delivered and the algorithm guarantees that delta-causal ordering holds by Lemma 1 and Lemma 3. For homogeneous data, any message that arrives at its destination before its deadline is delivered since it will have no dependencies by the time it reaches its deadline. All messages it depended on will either have arrived or exceeded their deadlines. Messages with no delta causal order relation, i.e., concurrent messages, are consistently ordered if each arrives within *min_wait* of the time the other message was sent. Rule 1 of the algorithm forces each message to be held for *min_wait* with respect to the time of transmission, after which, they are delivered if they have no dependencies. While

uniform ordering of concurrent messages at all sites can not be guaranteed, it is unlikely that concurrent messages will be delayed in such a fashion.

3.4 Performance

This section discusses performance issues of the algorithm. The synchronized clock constraint is relaxed to allow loosely synchronized clocks. The message size and complexity is discussed and a proof is presented that shows that handling longer dependency chains is expensive.

3.4.1 Loosely Synchronized Clocks

For simplicity, we have assumed that the physical clocks local to each site were all perfectly synchronized. If that constraint is relaxed and the clocks are no more than ε time units out of synchronization with each other, then the only change to the algorithm is the definition of delta, which only increases the "safety margin." Algorithms exist to keep clocks synchronized within 5-10 milliseconds [19,31,49]. If ε is non-zero, then delta is defined to be the time data in a message is useful *plus* the maximum clock drift (i.e., $\Delta_{new} = \Delta_{old} + \varepsilon$). The functionality and validity of the algorithm remains, because Δ only defines the lifetime of a message. By increasing Δ, messages are not discarded as soon, but the relation between messages does not change. Even with loosely synchronized clocks, the triangle inequality problem does not change. The transitive information is still sent and the ordering is still preserved. Lemmas 1 and 2, in Section 3.3.5, are still valid with a non-zero ε.

The use of physical time clocks requires that there be an external program that periodically synchronizes clocks at the sites. The overhead of such a program is small; however, as the level of synchronization increases, the overhead increases [19,31]. A higher level of synchronization is required to decrease the clock drift.

3.4.2 Discussion

Transitive dependencies with a length greater than two (i.e., greater than the triangle inequality) are likely to have no delta-causal relation. Delta < 3 * *min_wait*, by the definition of delta. So a message with a transitive dependency of length three or more that arrives before a message sent directly from the source to the destination implies that the out of order message has spent at least 3 times *min_wait* in transit (a delay of *min_wait* at each site). Since this exceeds delta, the life-span of the message, the data in that message is no longer of use and the message is discarded when it finally arrives. There will be no delta causal order between it and the message at the end of the transitive chain. In Figure 35, there is no delta causal relation between *m1* and *m4*, since they are separated by more than delta time units.

Rule 1 in receive_msg() forces messages to wait at least *min_wait* time units after they were sent before they are delivered. But messages will not be delayed more than *min_wait*, even if they must wait in a queue to be received at the destination site. Once the message has spent *min_wait* time units in the queue, all messages that are ahead of it in the queue must have waited *min_wait* as well, so they will be processed before this message. Rule 1 causes the message to wait only if *min_wait* time units have not elapsed

since the message was sent. The delay due to processing the message by the rest of the algorithm should be small. So at most, a message can spend *min_wait* time waiting to be delivered, plus whatever time it takes to process the messages ahead of it, which should be small in comparison to *min_wait*.

3.4.3 Message Size and Complexity

The message size overhead imposed by this algorithm is O(n), since every message contains two vectors of n elements, where n is the number of sites. This handles transitive dependencies of size 2. To handle larger transitive dependencies, a worst-case message overhead of O(n^2) is required. Refer to Lemma 4 for a proof.

This algorithm uses physical time instead of logical time. Expired data is easily detectable since its timestamp will not be within Δ time units of the current time. Such data can then be ignored, whereas with logical clocks, this information can not be obtained. In addition, the use of physical time allows concurrent events to be consistently ordered.

Simulation can be used to analyze the effectiveness of the algorithm. The percentage of discarded packets (with respect to the total number of packets) that occur both with and without the algorithm can be compared. In addition, the number of causal ordering violations and the overhead imposed by the algorithm, both in terms of processing time and messages can be analyzed.

In Figure 36, there is a transitive dependency involving three messages. To preserve the dependency information, all sites must send their dependency information to every site they communicate with, since that receiving site may send a message to a site with a dependency in the dependency vector. For example, in the figure, site A sends its dependency vector to B and site B sends that vector to site C, so site C will have that information when it sends a message to site D. There is no way to know what the potential destinations are, so all dependency vectors must be included, i.e., the entire dependency matrix. The next lemma shows that a much greater overhead is required to maintain transitive dependencies of lengths greater than length two.

Lemma 4: Transitive dependencies greater than length two require *N* vectors of information.

Proof:

Necessary Condition:

The first part shows that it is necessary to have *N* vectors of information for transitive dependencies of length three or more.

Proof by contradiction:

Assume sites send *X* dependency vectors, where $X < N$. Then current dependency information can be maintained for at most *N-1* out of the *N* sites, because the dependency vectors for at least one site can not be sent. Let that site be denoted as site *i*. If site *i* contains required transitive dependency information, the exclusion of site *i*'s

dependency vector causes that transitive dependency to be undetected. Without that information, subsequent messages can be delivered out of causal order. We now show no matter how the vectors are selected, it is possible for site *i* to contain such transitive dependency information.

Assume each site randomly selects a vector to not send. Let that site be site 1 in Figure 6. Then sites 2 and 3 have no record of message *m1* that was sent from site 1 to 4. Consequently, site 4 does not hold message *m4* until after *m1* arrives and the messages are not delivered in causal order.

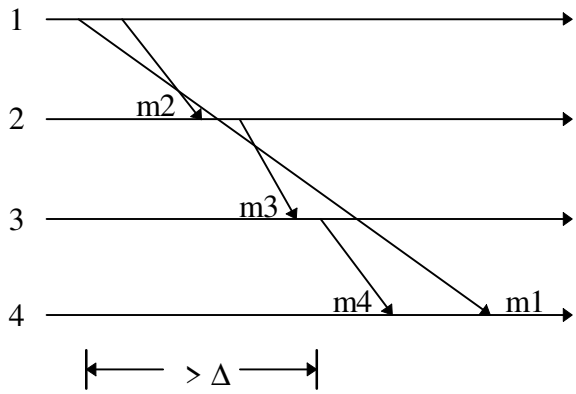

Figure 36: Transitive Dependency Information List If Site 1's Dependency Vector Is Not Sent

No matter what method is used to select which vector is to be excluded, there are always situations in which the dependency vector of the selected site contains information necessary to ensure causal ordering.

Therefore, even by sending *N-1* dependency vectors, all transitive dependencies of length three can not be captured. This can be generalized to show that *N-1* dependency vectors are insufficient for transitive dependencies of lengths greater than three. The dependency information for a chain of length four would be lost, since by the previous argument, the transitive information can not be preserved across four sites (for chains of length three), and would not be regained by adding another site at the end of the chain.

Sufficient Condition:

We now show that *N* dependency vectors are sufficient to capture transitive dependencies of length three or more.

Causal message order delivery can be insured by using the algorithm from [38] which provides causal order delivery of messages by sending *N* vectors with every message. The only difference is that this algorithm does not discard messages. Our algorithm discards messages that contain out-of-date dependency information which has already been subsumed by subsequent messages. If Δ is equal to infinity, then our algorithm does not discard messages.

Therefore, N vectors is both necessary and sufficient to deal with transitive dependencies of lengths greater than two. □

As a space saving technique sparse matrices can be used wherein only cells in the dependency matrix that have non-zero entries can be sent. Initially, this will require a

much lower message overhead. However, the number of these pairs that are sent will grow until they reach the size of the full dependency matrix (with added overhead to identify which cell they represent) [7,38]. Singhal and Kshemkalyani [42] present a technique for compressing vector timestamps that could be useful for reducing the size of the vectors that are sent with the messages.

3.5 Summary

We have presented an algorithm that ensures that real-time messages arrive in delta-causal order while limiting the message overhead to 2 vectors of length *N* per message. A proof was presented to show that the algorithm ensures that messages are delivered in the proper order unless they are past their deadlines in which case they are not be delivered. Transitive dependencies of lengths greater than two are ignored since the algorithm ensures that two messages separated by more than two dependencies do no have a delta causal relation between them. The algorithm minimizes discarding messages that arrive at a site by delivering them when the messages they depend upon expire. This way any message that arrives at a site before its deadline is used.

This algorithm can be used to ensure that multimedia data is delivered in delta-causal order. Multimedia data needs to be delivered in causal order in order to maintain the meaning of the data presentation. Because the algorithm supports data with real-time deadlines, data that has exceeded its deadline is discarded. However, the algorithm attempts to minimize the amount of data that must be discarded, and will utilize

unexpired received messages, even if there are missing messages they depend upon. This algorithm is well suited for multimedia data, which has real-time deadlines.