Design, Implementation, and Evaluation of a Software-based Real-Time Ethernet Protocol

Chitra Venkatramani *Tzi-cker Chiueh*

Department of Computer Science
State University of New York at Stony Brook

email : {chitra, chiueh}@cs.sunysb.edu

Abstract

Distributed multimedia applications require performance guarantees from the underlying network subsystem. Ethernet has been the dominant local area network architecture in the last decade, and we believe that it will remain popular because of its cost-effectiveness and the availability of higher-bandwidth Ethernets. We present the design, implementation and evaluation of a software-based timed-token protocol called RETHER that provides real-time performance guarantees to multimedia applications without requiring any modifications to existing Ethernet hardware. RETHER features a hybrid mode of operation to reduce the performance impact on non-real-time network traffic, a race-condition-free distributed admission control mechanism, and an efficient token-passing scheme that protects the network against token loss due to node failures or otherwise. To our knowledge, this is the first software implementation of a real-time protocol over existing Ethernet hardware. Performance measurements from experiments on a 10 Mbps Ethernet indicate that up to 60% of the raw bandwidth can be reserved without deteriorating the performance of non-real-time traffic. Additional simulations for high bandwidth networks and faster workstation hardware indicate that the protocol allows reservation of a greater percentage of the available bandwidth.

1 Introduction

With the growing trend towards distributed multimedia applications, it has become essential for the underlying systems to provide resource guarantees. Applications such as LAN-based teleconferencing and video-on-demand services require support for real-time data transport from the underlying network to support real-time video playback. LAN-based real-time transport also plays a critical role in extending WAN-based teleconferencing systems to the end-user's display, which may be connected to the site's WAN interface with a local area network. With the large existing base of Ethernets, it is essential to support such applications on the Ethernet. Our work addresses the problem of providing bandwidth guarantees on an Ethernet-based network.

Ethernet has been the most popular LAN scheme used in the last decade. It is simple, efficient and its bus-based architecture is particularly useful for multimedia applications because it can easily support broadcast and multicast. Despite recent trends towards ATM and FDDI-based high-speed networks, we believe that Ethernet will continue to be popular, especially at the leaves of a hierarchy of inter-networks, for the following reasons. Firstly, newer generations of Ethernets such as Grand Junction Networks' 100Base-X and HP's 100Base-VG provide a bandwidth of 100 Mb/sec, which is comparable to those of other high-speed LANs, but at a lower price. Secondly, the advent of *switch-hubs* has improved the scalability of the Ethernet architecture by allowing the interconnection of Ethernet segments into a local inter-network. Hence, even for large-scale organizations, Ethernet will remain a viable choice. Lastly, for the existing base of 10 Mbps Ethernet users, the alternative of changing to a new technology, and consequently replacing the wiring, hubs, and network interfaces would be very expensive and would almost certainly delay the deployment of distributed real-time and multimedia applications.

Most existing approaches for supporting distributed multimedia applications take a best-effort approach by dynamically adapting applications' behavior to the available network bandwidth. One of the advantages claimed is that no additional support is required from the hardware. The RETHER system described in this paper supports real-time bandwidth *guarantees* without any modification to existing Ethernet hardware.

Providing bandwidth guarantees on an Ethernet is problematic for the following reasons. Firstly, its medium access protocol is contention-based. All nodes requiring the use of the channel, first sense it to determine if it is idle. If so, they compete for the channel and on sensing collision, backoff for a random time interval before attempting retransmission. This protocol performs well for non-real-time traffic under light load, by providing very quick access to the channel. However, it causes nondeterministic access delays to the network, which is usually unacceptable for transport of temporally constrained data. Although the Ethernet protocol specification has a provision for prioritized access arbitration, this mechanism does not in itself offer guaranteed bandwidth to an arbitrary pair of nodes. Besides, most commodity Ethernet controllers do not necessarily implement this feature.

Secondly, the Ethernet protocol is not "fair" [7]. We conducted the following experiment to verify this. In this experiment, a video-conferencing application was run over the Ethernet without any bandwidth guarantees. The results are shown in Table 1. Each stream stands for a connection between a sender and a receiver. Each sender sends up to 3000 video frames at the rate of 30 frames/sec and 6400 bytes per frame (equals MPEG-I bandwidth of 1.5Mbps). The main performance criterion for our evaluation is the number of frames that arrive late at a receiver. Packets are considered late if they arrive more than 2 msec after their expected arrival time. Because the measured delay of our experimental setup is the minimal overhead that the client program has to experience, the choice of 2 msec as the threshold seems reasonable. The results are not an average of several runs, but are representative of a single experiment. This preserves the skewed pattern of the experimental results. Table 1 shows the scenario when there is no asynchronous non-real-time traffic, and data is copied only once at the receiver side, from the network card to user space. With only one or two such streams, no packets are delayed. With three streams, 0.47% of the packets arrive late. With four or five streams, the number of late frames for certain streams increases dramatically. It is clear from this that the packet delay behavior is asymmetric among the nodes. That is, there is a tendency for a subset of nodes to monopolize the channel within a short period of time. This is particularly bad for real-time video data transfer, since the quality of certain video sequences will be significantly worse than others, and there is no way to identify these sequences a priori.

Number of Streams	**1**	**2**	**3**	**4**	**5**
Stream 1	0	0	0.00	16.47	32.47
Stream 2	x	0	0.00	0.13	98.33
Stream 3	x	x	0.47	0.03	28.47
Stream 4	x	x	x	13.00	3.17
Stream 5	x	x	x	x	69.33

Table 1: The percentage of delayed frames vs. the number of concurrent video streams, assuming data is copied once at the receiver. x means non-applicable.

The following points need to be kept in mind while interpreting the above results. First, the experiments are done on a relatively idle network and therefore non-real-time traffic is at a minimum. Second, the maximum number of real-time nodes on the network is relatively small, in this case, five. Therefore, the loss of channel efficiency due to collision does not show up even when the traffic load is as high as 45%. As the number of hosts increases, collisions may become the principal reason for packet delay. Third, the results reported here only apply to a single segment network. When a video connection spans multiple segments, the overall degradation of end-to-end digital video transport would be the product of the degradation due to each segment.

Given the above observations, the central goal of this work is the implementation of a software-based protocol for existing Ethernet hardware to provide real-time guarantees to multimedia applications. The protocol, RETHER (Real-time ETHERnet), adopts a contention-free deadline-driven token-bus protocol to provide deterministic performance. Some important design decisions to ensure that the protocol works were driven by features unique to the Ethernet.

In this paper, we present the design and implementation of RETHER and analyze the measured performance results. We also analyze its performance under faster network hardware–100-Mbps Ethernet–and faster workstations, using a simulator. Finally, we discuss scalability issues of RETHER and the functioning of the protocol as part of the Stony Brook Video Server (SBVS) project [10]. Our current implementation runs on a network of five i486-based PCs running FreeBSD v1.1.5.1 and using off-the-shelf Ethernet cards (SMC Elite), on a 10Mbps Ethernet. Initial performance measurements indicate that it is possible to support real-time traffic utilizing up to 60% of the raw Ethernet bandwidth without disrupting the timings of other operating system functions that use the network.

The remainder of the paper is organized as follows. Section 2 describes the RETHER protocol. Section 3 describes the implementation of the system and the procedural interface provided to the user. Section 4 describes the measurements of our implementation and their performance implications. Section 5 uses simulations to further evaluate the performance of the RETHER protocol under larger networks and faster hardware/links. Section 6 describes the Stony Brook Video Server currently under development. Related work is described in Section 7 and finally, Section 8 presents our conclusions from the experiments and also discusses future directions in our research.

2 The RETHER Protocol

This section explains the protocol design in detail and is organized as follows. Section 2.1 gives an overview of the protocol. Section 2.2 fills in the details. Our decentralized admission control policy is described in Section 2.3, while Sections 2.4 and 2.5 explain the fault-tolerance features and the issues relating to extension of the protocol to multi-segment Ethernets, respectively.

2.1 Overview

The RETHER protocol has the following features —

1. It allows applications to reserve bandwidth and guarantees the reservation throughout the lifetime of the application.
2. It is implemented completely in software over off-the-shelf network hardware.
3. It adopts a hybrid scheme whereby the network operates using a timed token-bus protocol when there are real-time sessions and using the original Ethernet [1] protocol at all other times. This scheme reduces the performance degradation of non-real-time traffic, and presents minimal disruption to existing network applications.

The network operates using the original Ethernet protocol (CSMA/CD) until a real-time request comes along. When this happens, it switches to a token-bus mode. In this mode, real-time data (like audio and video) is assumed to be periodic and time is divided into cycles of one period. For example, for video applications that require to send data at 30 frames per sec, the cycle time would be 33.33 millisec. In each cycle, channel access for *all* traffic – real-time(RT) and non-real-time(NRT) – is regulated by a token. RT traffic is scheduled to be sent out first in each cycle and NRT traffic is allowed to use the channel in the remaining time. Hence, in each cycle, all admitted RT nodes get to send data. But, all nodes may or may not get to send NRT data, depending on the time left in the cycle. When all the time in the cycle is exhausted, the token returns to the first RT

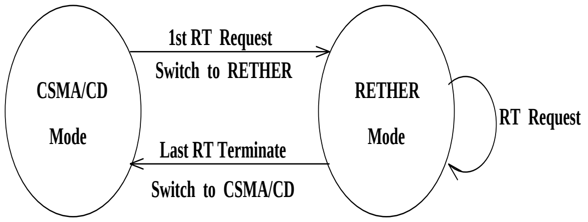

Figure 1: Transitions in the RETHER Protocol.

node and begins a new cycle. The details of how this time is maintained and how the token is circulated are explained in the following sections. When the last RT session terminates, the network switches back to the CSMA mode.

2.2 Protocol Description

Figure 1 shows the different modes and transitions in the RETHER protocol and they are described below.

2.2.1 The CSMA Mode

Nodes compete for the channel using the generic Ethernet protocol. This protocol performs well under light loads and deteriorates only when the network is heavily loaded. Nodes operate in this mode until the arrival of the first RT request which initiates a switch to the RETHER-mode.

2.2.2 Switch to the RETHER Mode

When an application on a node generates an RT request and the node is in the CSMA-mode, it becomes an *Initiator* by broadcasting a *Switch-to-RETHER* message on the Ethernet. Every node that receives this message responds by setting its protocol mode to RETHER mode. It holds off sending any more data and awaits completion of transmission of the packet already in the transmission buffer of its network interface. Then it sends an acknowledgement back to the initiator. As soon as the initiator receives all the acknowledgements, it creates a token and begins circulating it. This completes a successful switch to RETHER-mode.

Acknowledgements are crucial to the success of the protocol for two reasons. Firstly, acknowledgements signify the willingness of the nodes to switch to RETHER-mode. Secondly, the fact that acknowledgements are successfully sent out indicates that the nodes do not have any pending packet in the backoff phase of the CSMA/CD protocol. The latter is particularly important because in typical Ethernet cards, software has no control over the data once it has been transferred to the network interface buffer.

The protocol can robustly handle the following scenarios that may affect the process of switching to RETHER-mode:

Multiple Initiators : This condition occurs when applications running on two or more different nodes generate an RT request simultaneously, and the corresponding nodes initiate a transition to RETHER mode. The race condition occurs when all of these nodes have placed their *Switch-to-RETHER* broadcast message in the network interface buffer and hence have no way of withdrawing it. We resolve this condition by giving the node with the smaller ID, higher priority. All the initiators contend for the channel and eventually one of them succeeds in broadcasting the switch message. Since there are more initiators, the first node receives a switch message from one of the other initiators. The first node responds only if the ID of the new node is smaller than its own. If not, it ignores the switch message. All nodes (initiators and non-initiators) acknowledge a switch message from an initiator with ID smaller than the one they already acknowledged. Hence, only the initiator with the smallest ID wins. It receives acknowledgements from all the other nodes and completes the transition to RETHER-mode.

For example, if Node 1 and Node 3 were initiators and Node 3 succeeds in broadcasting its message first. Then, all other nodes except Node 1 send back acknowledgements. This is because Node 1 knows that it is an initiator that has priority over Node 3 to complete the transition. Hence, Node 3 does not receive all the acknowledgements and cannot complete the transition. Subsequently, when Node 1 succeeds in accessing the channel, all nodes, including Node 3, send back acknowledgements and Node 1 completes the transition.

Other failures: Ideally, the initiator that gains control of the transition will receive $N-1$ acknowledgements, where N is the number of nodes on the network, and complete the transition. But, the following scenarios may occur– i) some of the nodes in the network are non-operational, ii) some of the acknowledgements are lost, iii) some of the nodes do not receive the *Switch-to-RETHER* message. The initiator sets a timer while awaiting acknowledgements. In all of the above cases, the initiator times out because it does not receive all the acknowledgements. After a few retries, it concludes that the nodes that did not acknowledge are dead. It then informs all other "live" nodes about the current state of the network by including the list of "dead" nodes in the token.

2.2.3 The RETHER Mode

In this mode, access to the channel is regulated by a token. The token circulates among two sets of nodes — the RT-set and the NRT-set. Only nodes that have made a bandwidth reservation belong to the RT-set, while *all* nodes belong to the NRT-set.

Real-time data is assumed to be periodic and the interval between successive visits of the token to each node in the RT-set is exactly one period. This period is also known as the Token Rotation Time (TRT) and is a system configuration parameter. The token visits the nodes in either RT or NRT modes, when the node can hold the token for an interval of time called the Token Holding Time (THT). During this time, it can send the corresponding type of message. Each RT process specifies its required transmission bandwidth as the amount of data it needs to send during each TRT. This is translated into its THT using Equation 1, where $S/W\ Overhead$ refers to software overheads and t_{token} referes to the token processing overhead. These are discussed in detail in Section 4.2.

$$THT_{RT} = \frac{Data\ per\ TRT}{Ethernet\ bandwidth} + S/W\ Overhead + t_{token} \quad (1)$$

$$THT_{NRT} = \frac{Message\ Size}{Ethernet\ bandwidth} + S/W\ Overhead + t_{token} \quad (2)$$

Equation 2 is used to determine the THT for NRT nodes. While THT_{RT} is predetermined based on the reservation request, THT_{NRT} is computed each time the token visits an NRT node, based on the size of the message to be sent. By controlling the $Message\ Size$, it is possible to tune THT_{NRT} to get better performance. Each message could

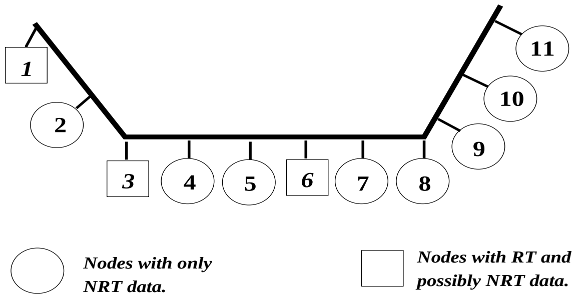

Figure 2: A Sample Network Configuration.

consist of one or more packets, each of which can have a maximum size of an Ethernet Maximum Transmission Unit (MTU). We can adopt different policies for the transmission of NRT data when the token arrives —

- transmit only one packet,
- transmit as many packets as possible, or
- transmit at most a fixed number packets.

The first policy is fair and allows each node a small share of the residual bandwidth. However, since Ethernet traffic is usually bursty in nature, the second policy might be a better choice since the response time will be shorter if as many of the queued packets as possible could be transmitted. The third policy only permits at most a fixed number of packets to be sent. Depending on the characteristics of the traffic on the network, any one of the policies could be adopted to improve performance. For simplicity, we adopt the first policy in our prototype implementation.

In order to maintain the real-time guarantees, the interval between two consecutive visits of the token to an RT node must be one TRT. This time is maintained in the token as a soft clock that holds the value of the residual time in the current cycle. In each cycle, the token starts out with its *ResidualTime* field set to one TRT. It visits all the RT nodes in order. Each of these nodes sends one unit of RT data and decrements the *ResidualTime* by its THT_{RT}. Then, in the remaining time, the token visits the NRT nodes in a round-robin fashion, starting with the last unvisited NRT node in the previous cycle. Each NRT node determines if there is sufficient time to transmit data. If so, it sends out data using one of the policies outlined earlier, decrements the *ResidualTime* field on the token by its THT_{NRT} and passes the token onto its neighbor. If not, it sends the token back to the RT-set flagging itself as the NRT node to be the first to receive the token in the next cycle. The first RT node then resets the *ResidualTime* to be equal to one TRT, thereby beginning a new cycle. Hence, the token need not necessarily visit all the nodes in the NRT-set in each cycle. It may visit nodes in the RT-set multiple times before visiting a node in the NRT-set. In this way, the nodes in the RT-set are given priority over those in the NRT-set.

It is also possible to have multiple senders or receivers per node. For instance, if a node has two senders, the token visits the node in the RT mode twice in each token cycle.

Figure 2 shows a sample configuration of a network. Here, the italicized nodes 1, 3 and 6 belong to the RT-set. During each TRT, the token visits the nodes 1, 3 and 6 and then the NRT nodes in the remaining time. A possible sequence in two token rotation cycles would be–

$$\mathit{1} - \mathit{3} - \mathit{6} - 1 - 2 - 3 - 4 - 5 - 6 - 7 -$$

$$\mathit{1} - \mathit{3} - \mathit{6} - 7 - 8 - 9 - 10 - 11 - 1 - 2 \ldots$$

Besides the *ResidualTime*, the token carries the following information vital to the functioning of the protocol:

- *RT-set information*: This is an up-to-date list of all the currently active RT sessions and their bandwidth reservations. This information is required by the admission control algorithm.
- *Network state information*: This is an up-to-date list of all the nodes in the network that are currently non-operational. This data structure determines which nodes the token should visit as it circulates around the network. Section 2.4 describes how this list is maintained.

2.2.4 Switch to the CSMA Mode

The last node to terminate its RT session destroys the token and sends out a *Switch-to-CSMA* broadcast message. All the nodes switch back to CSMA-mode in response to this message.

2.3 Admission Control

We adopt a simple decentralized admission control policy to admit new RT sessions. Each node determines whether or not to admit its own application's reservation request. A new session with a reservation THT_{RTnew} can be admitted if and only if

$$\left(\sum_{i \in RT\_set} THT_{RTi} \right) + THT_{RTnew} + T_{NRT} \leq TRT \quad (3)$$

where T_{NRT} is the time in each cycle set aside for NRT traffic. It basically sets an upper bound on the time between consecutive visits of the token to an NRT node and hence determines the starvation characteristics of NRT traffic. The minimum value for T_{NRT} should be such that the worst case NRT network access latency is less than the time-out values of higher layer protocols. As explained in Section 2.2.3, the token returns to an NRT node when it has visited all the other nodes in NRT-set, in a round-robin fashion. This happens when the sum of the residual times in X cycles is greater than or equal to to the time to visit all the NRT nodes once and transmit NRT data from them. This is indicated in Equation 4 where THT_{NRTavg} is the average time that an NRT node holds the token, over X cycles, and N is the number of nodes in the network. Hence, the worst case network access latency is X cycles, as indicated in Equation 5.

$$X * T_{NRT} \geq N * THT_{NRTavg} \quad (4)$$

$$Worst\ Case\ Net.\ Access\ Latency = X * TRT \quad (5)$$

If 5% NRT traffic needs to be supported on the network, then $N * (THT_{NRTavg} - t_{token}) = 0.05 * X * TRT$. Substituting this back in Equation 4, we get

$$T_{NRT} \geq \frac{N * t_{token}}{X} + 0.05 * TRT \quad (6)$$

X can be determined from Equation 5, given a worst case network access latency and the minimum value of T_{NRT} can be determined from Equation 6.

If a request is admitted, it is added to the RT-set information on the token and hence, it can start sending RT data from the next cycle, when the token arrives in the RT mode. If not, the request returns to the user process with an error. The user may keep trying until the request is admitted.

In our implementation, admission decision at a node is postponed until that node receives the token. In other words, a process' request to initiate an RT connection does not return until the associated node receives the token. This is because the token contains the most up-to-date information about the RT-set and the bandwidth reservations. We make this decision because the only other alternative is to maintain this information at each node. However, this may lead to incorrect admission decisions when two or more nodes receive RT requests simultaneously and each node admits its request without the knowledge of admission decisions made at other nodes. Given the number of nodes on a 10Mbps Ethernet and the number of RT sessions that can be supported on it (around five MPEG-I streams), including the RT-set information in the token seems reasonable.

When an RT session terminates, the node merely removes it from the RT-set information on the token, when it receives the token.

2.4 Failure and Addition of Nodes

Since nodes in a network of workstations environment are likely to crash or get rebooted without any warning to other nodes, it is essential to have a mechanism that can detect this, reconfigure the token bus and regenerate the token if it is lost. The RETHER protocol is designed to be robust enough to handle these events transparently. Hardware-based token passing schemes such as FDDI, however, have built-in hardware to detect node shut-down and act accordingly.

When the network operates in the RETHER-mode, each node monitors the state of its logical successor. Each node, after sending the token to its successor, sets an *Acknowledgement Timer*. If the successor is alive, it holds the token for the duration of its THT and then sends the token to its successor and also an acknowledgement to its predecessor. On receiving this, the monitoring predecessor cancels its timer. If, on the other hand, the successor is dead, the monitoring node times out and assumes that its successor is dead. It then updates the list of "dead" nodes on the token, modifies the RT-set information if the failed node had any reservation and sends the token to the next "live" successor.

The value of the *Acknowledgement Timer* is chosen to be equal to the residual time on the token. This is because, the successor should definitely transmit the token within this interval. Also, in order to reduce the performance penalty due to acknowledgements, we piggyback the acknowledgement, with the token passed to the successor. This is possible because Ethernet hardware supports multicast based on group destination addresses. This scheme detects failed nodes and maintains the current state of the network in the token.

When a node boots up, it broadcasts a message identifying itself. If the network is in RETHER-mode, the node with the token removes this new node from the list of "dead" nodes on the token. The new node then waits for the token to arrive. The quantity T_{NRT} in Equation 3 guarantees that the token will reach the new node within a certain interval. If the token does not arrive within this interval, the new assumes that the network is in the CSMA mode. The broadcast message may introduce a slip in the protocol timings because it may collide with the node attempting to access the network. However, the slack bandwidth T_{NRT}, absorbs this slippage.

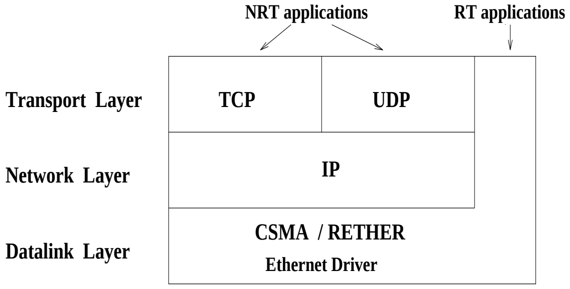

Figure 3: Interface Provided by RETHER in the UNIX Network Subsystem.

2.5 Extension to Multi-Segment Ethernet

One popular method to boost the performance of Ethernet is to partition a segment into multiple segments and connect them through bridges or routers. The RETHER protocol, as described above, guarantees real-time bandwidth to network connections within a single Ethernet segment. To extend this protocol to multi-segment Ethernet environments, a real-time connection among nodes on different segments is decomposed into multiple real-time sub-connections, each of which corresponds to one of the segments that physically connect the two communicating nodes. Due to this, the fact that Ethernet does not support broadcast across multiple segments does not affect the protocol. Hence, the RETHER protocol can easily support bandwidth reservations in multi-segment Ethernets as long as all the network nodes run the RETHER system, and the intermediate routers observe the real-time schedules associated with each of the segments to which they are connected. However, unlike single-segment Ethernets, there are two issues related specifically to multi-segment Ethernets. The first is that latency guarantees are harder because the real-time schedule on each segment is determined completely independently of the others. As a result, extra buffering must be provided on intermediate routers to accommodate the discrepancy of these schedules. These buffering delays create interactivity problems for real-time teleconferencing applications, which according to ergonomical studies can have a maximum delay of 100-150 milliseconds. This means that the number of hops that such a real-time connection can cross is bounded. The second is the issue of connection setup time. In the worst case, each of the participating segments needs to switch from the CSMA to the RETHER mode. Therefore, the connection setup time in this case is roughly the maximum of the mode switch time associated with each segment since multiple segments can switch to RETHER mode in parallel.

3 Implementation

Figure 3 shows the organization of a UNIX network subsystem that includes the RETHER software. RETHER functions directly above the hardware interface layer and all protocol processing is performed by special network interrupt service routines. Real-time applications can directly access the services of the RETHER subsystem. For conventional TCP/IP applications, the token-bus scheme in RETHER is completely transparent.

The implementation of the protocol is modular and can be partitioned into two parts–the first part deals with the procedural interface provided to the user along with initial setup required by the protocol, while the second is the actual low-level implementation of the token passing protocol.

3.1 Procedural Interface

RETHER services can be accessed by applications either directly or through a user-level library.

- *retherOpenRTsession(bufAddressVector, bufLength, numBufs, sessionId, flag, receiverId)* : Maps all the user buffers of length *bufLength* specified in *bufAddressVector* to kernel address space and locks them in physical memory. The parameter *flag* determines if the caller is a sender or a receiver. If it is a sender process, the system call takes the receiver node Id and returns a unique *sessionId*. If the caller is a receiver process, the call takes a *sessionId*, which is the unique Id of the corresponding sender. This call allows the user to specify multiple buffers, to allow users to use schemes like double-buffering.

- *retherCloseRTsession(sessionId)*: Awaits the RT token and when it arrives, terminates this RT session by removing it from the RT-set information maintained on the token. It also unlocks the buffers from physical memory and unmaps them from kernel address space.

- *retherAdmit(sessionId)*: Performs the admission control test for the session when the NRT token arrives. If this session is the first RT session, then this call also initiates a switch to the RETHER mode.

- *retherSend(sessionId, blockFlag)*: This can be called in the blocking or non-blocking mode depending upon the value of *blockFlag*. In the blocking mode, this call blocks until one of the buffers is emptied by the kernel, when the RT token arrives. This may be called when all send buffers are full. When the kernel receives an RT token, it sends the contents of the next buffer with valid data.

- *retherRecv(sessionId, blockFlag)*: This can be called in the blocking or non-blocking mode, depending upon the value of *blockFlag*. In the blocking mode, this blocks until one of the buffers is filled by the kernel. This system call may be made when all the receive buffers are empty. On receiving a frame, the kernel copies it into the next buffer, overwriting its contents if it still has valid data, thereby giving priority to newer data.

- *Utility functions*: Other utility functions for initialization, getting and setting RETHER state information and collecting statistics are also provided.

We have built a library over the system call interface to do the buffer management. The user only specifies the number of buffers required and the library allocates them, makes the appropriate RETHER system calls to map the buffers and keeps track of full/empty buffers. Since the data buffers are mapped in user and kernel address spaces, buffer status can be checked and updated by both.

3.2 Protocol Implementation

Our implementation of the RETHER protocol is in the kernel and bypasses all upper-layer protocols like TCP/IP. By mapping the user buffers to kernel address space, we are able to copy data directly from user buffer to the Network Interface Card (NIC) at the sender side and from the receive buffer on the NIC to the user buffer at the receiver side. This address remapping is a performance optimization that minimizes data copy overhead. This part of the implementation depends upon certain functions and interfaces provided by the virtual memory module of the kernel.

Individual Ethernet packets belonging to one RT session carry sequence numbers to aid reassembly at the receiver end. If the receiver receives an incomplete frame, the user process is informed accordingly. We do not retransmit lost data since it is not important for data like video and audio and would lead to violation of the performance guarantees.

The implementation of the token-bus protocol itself is independent of any other part of the kernel code. This module processes the various events in the protocol and maintains the state of the token bus. It involves modifications to the low-level data structures pertaining to the Ethernet driver software and the hardware interface layer of the UNIX network software. Because of its modularity, eventually we plan to distribute our implementation as a network driver with well-defined interfaces so that it can be easily integrated with other UNIX derivatives.

4 Performance Measurements and Analysis

In this section we describe the experimental setup, the performance measurements we made on the token passing overhead, the delay as seen by NRT packets, and the bandwidth guarantees available to RT sessions.

4.1 Experimental Setup

We have the implementation running on a network of five i486 PCs running the FreeBSD v1.1.5.1 kernel. The CPU runs at a clock rate of 66 MHz and the Ethernet card is on an ISA I/O bus. Data copy from the NIC memory to physical memory is done using programmed-I/O. The limit of five nodes is due to resource constraints on the number of PC's.

We first conducted various latency measurements to account for software overhead and token passing overhead. These are reported in Section 4.2. We then conducted experiments to measure the effect of bandwidth reservation on NRT traffic under different amounts of bandwidth reservation and different number of sessions. The average time between NRT token visits to a node was used as the metric to measure the impact of the protocol on NRT traffic. This metric reflects the average network access latency of a node. The NRT load on the network was artificially introduced by having two arbitrary nodes exchange packets of different sizes continuously. This workload simulates a heavily-loaded Ethernet. So the NRT packet delay measurement reported here is on the conservative side because typical Ethernet loads are below 10-15%. The total real-time bandwidth reservation ranged between 5% and 75% of the raw Ethernet bandwidth, which is 10 Mbits/sec. The results are reported in Section 4.3.

We have also developed a prototype video-conferencing application that is directly built on top of RETHER. Since our applications required to send data at the rate of 30 frames per second, the token-bus was configured to operate with an TRT of 33.33 ms.

4.2 Latency Measurements

Since the RETHER protocol is implemented in software, it is important to reduce the protocol overhead to the minimum. Our current implementation achieves this goal by embedding the protocol processing directly within the interrupt handlers, and thus avoiding unnecessary buffering and scheduling overheads. Table 2 shows the components of the time spent by a token on a node when there is nothing to send on that node. It is broken up into sender and receiver sides. The total time for a token to visit a node is approximately 247μsec.

No.	Operation	Time Taken (μsec)
1	Receiver preprocessing	27
2	Copy token from NIC to memory	31
3	Receiver postprocessing	12
4	RETHER Protocol processing	15
5	Sender preprocessing	30
6	Copy token from memory to NIC	29
7	Sender postprocessing	15
8	Transmission delay + Receive Interrupt + Context switch + Schedule interrupt	88
	Total	247

Table 2: Breakdown of the Token Passing Overhead

Software has no control over Item 8. With faster machines and faster links, this overhead will reduce. The other major components are the time for data copies and the time for interrupt handling in the Ethernet driver. In our current implementation, all RETHER packets go through a thin layer of driver software before reaching the RETHER code. We are in the process of moving the RETHER protocol processing into the Ethernet driver, whereby, the interrupt handling overhead will be reduced to a minimum. Other than these, the RETHER protocol processing takes only 15μsec, or around 6% of the total token overhead.

The values for *THT*s are determined using Equations 1 and 2. The *S/W Overhead* in these equations was determined using empirical measurements on an idle network. The time to send a real-time packet to the receiver includes two main parts. One is the time to copy the data from main memory to the Ethernet card and the other is the time to put the packet on the wire. With the PC's and the network that we used, the time to transmit the packet over the wire is larger than the time to copy the packet to the Ethernet card. Since each RT data unit is composed of a number of Ethernet MTU packets, the overall transmission time per RT data unit was determined predominantly by the time to transmit the data, since data copy and data transmission overlap. Our empirical measurements yielded the following formula –

$$Transmission\ Time = \frac{Packet\ Size}{Ethernet\ Bandwidth} + a*n + b \quad (7)$$

where a is a constant processing overhead per packet. It includes processing of the transmission completion interrupt and initiation of the transmission of the next packet. This

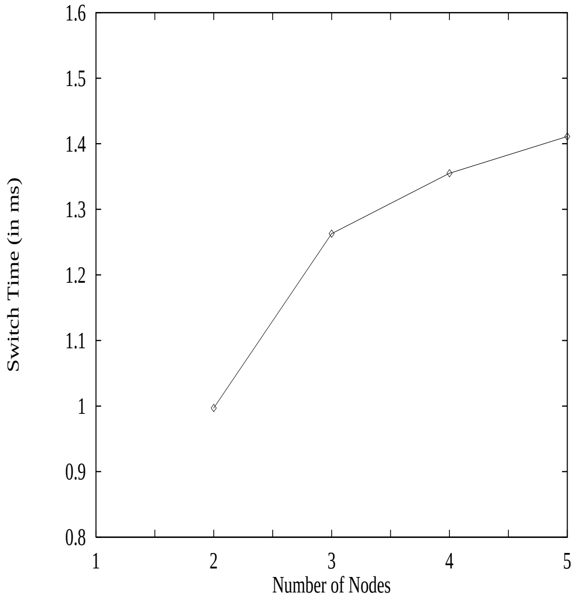

Figure 4: Effect of Network Size on Switch Time.

was determined to be 140μsec. n is the number of Ethernet packets and b is the processing and data copy overhead for the first packet, measured to be 650μsec for an MTU packet. The data copy overhead is high because the machines we use do not support DMA to/from network cards. All the data copy operations between memory and the NIC use programmed-I/O. The quantity $(a * n + b)$ in Equation 7 corresponds to the *S/W Overhead.*

For heterogeneous networks, these constants and the token overhead would have to be measured for each architecture separately.

Because the goal of the RETHER protocol is to provide real-time bandwidth guarantee, we measured the temporal distance between consecutive arrivals of the token to an RT node, under various RT/NRT workload conditions. By choosing the protocol parameters as described above, we observed that during our experiments, the bandwidth reservations were indeed guaranteed in all cases. Therefore, for the rest of this section, our focus is mainly on minimizing the performance impact of the protocol on non-real-time traffic.

4.3 Results and Analysis

Because the RETHER protocol features a hybrid-mode operation, it is important that the overhead of switching between these two modes be reduced to the minimum. We first measured the time taken to switch to the RETHER mode from the CSMA mode. Figure 4 depicts the effect of the number of physical nodes in the network on the switch time from the CSMA to the RETHER mode. Network nodes will operate using the RETHER protocol when there is at least one RT session active between any two nodes. As the number of nodes on the network increases, an *Initiator* needs to collect more acknowledgements before completing the switch to the RETHER mode, thus potentially increasing the switch time. Also, it takes additional time to empty the packets already on the network card when the RT session request arrives. The worst case scenario would be when all the nodes have full-length buffered packets at the interface and acknowledgements need to be sent. But, since a typical

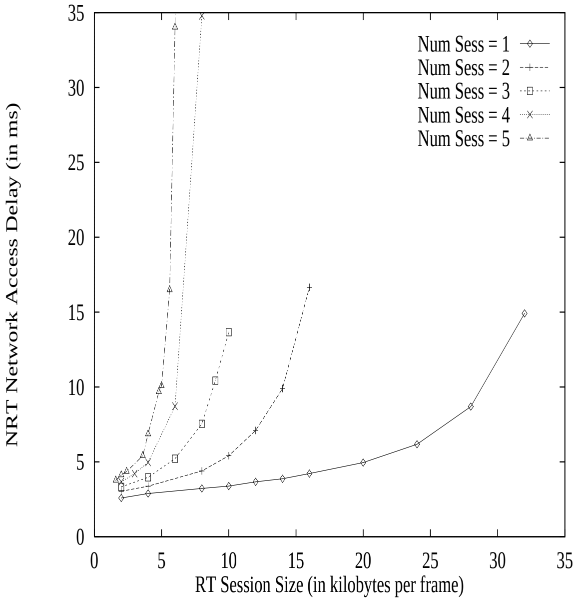

Figure 5: Non-Real-Time Packet Delay vs. Session Size, Maintaining a Constant Number of Sessions.

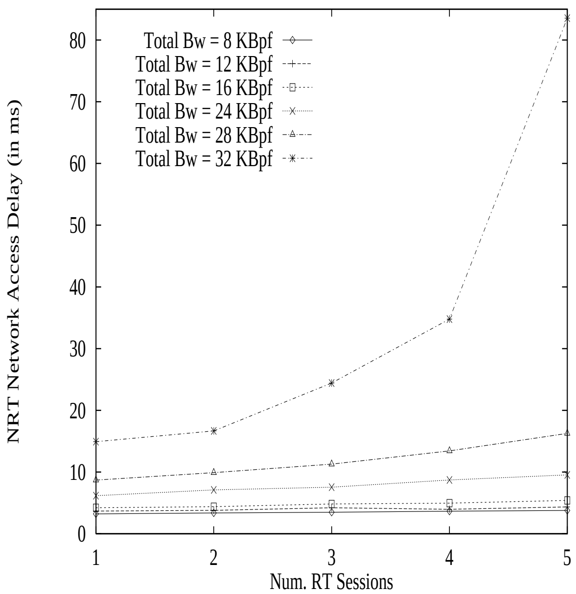

Figure 6: Non-Real-Time Packet Delay vs. Number of Sessions, Keeping the Total Reserved Bandwidth Constant.

Ethernet is idle most of the time, this delay should be quite small. As expected, Figure 4 shows a positive slope indicating that the switch-time increases with the number of nodes on the network. This is mainly because of the interrupt and processing overheads due to handling the various acknowledgements. Part of the reason for this could also be the inevitable collisions that occur when all the nodes simultaneously try to send acknowledgements back to the initiator. To avoid this, we can use a strategy where each node waits for a time interval proportional to its ID before acknowledging the *Switch-to-RETHER* message. The results in Figure 4, however, do not reflect this optimization.

Next we examine the effect of different numbers of RT sessions with different session sizes on the NRT access delay, in a heavily-loaded network. In each of the plots, RT session size stands for the amount of data to be sent per TRT. For instance, an RT session with a bandwidth reservation of 1.5 Mbps (MPEG-I stream) would send 6.25 kilobytes of data per TRT of 33.33 ms.

Figure 5 displays the effect of the amount of bandwidth reserved per RT session on the NRT packet delay. It shows that given a tolerable access delay of 10ms, we can reserve up to 28KBytes per frame (KBpf) (69% of raw Ethernet bandwidth) for one session, and up to a total of 26KBpf (64% of Ethernet bandwidth) for four sessions. The session sizes chosen are for transmitting compressed digital video data at MPEG-I or MPEG-II rates. As expected, the total percentage of bandwidth that can be reserved decreases slightly with the number of sessions, because the protocol overhead due to token passing increases. If greater delays in NRT traffic can be tolerated, then more bandwidth can be reserved. Note that the design goal of the RETHER protocol is to support real-time connections without causing unnecessary time-outs in higher level applications like NFS, that use the network. Therefore, it is acceptable to have longer NRT packet delays as long as it is within the bounds of time-out values.

Figure 6 shows the variation in NRT packet delays as the number of sessions is increased, keeping the total reserved

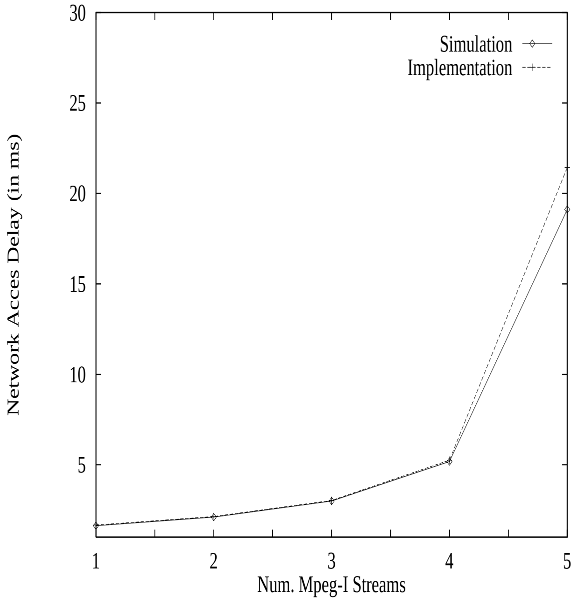

Figure 7: Comparison of Simulation and Implementation Results.

bandwidth constant. This experiment effectively measures the protocol overhead due to the token passing scheme. As is apparent from the graph, the effect of the number of sessions is noticeable only when the total bandwidth reserved is very high — 32 KBpf or around 75%. This implies that the protocol overhead is not very significant in the normal range of operation (under 50%-60% of Ethernet bandwidth reserved).

5 Scalability Simulation Study

Our initial simulations results reported in [9] were done with transmission times determined using the Packet Internet Groper (ping) program. Values obtained from later low-level measurements reported in Section 4.2 were lower than those obtained using ping and were used in the simulator to validate it. A comparison of the results obtained from the simulator and those obtained from the implementation is shown in Figure 7. In order to be consistent with the experimental setup used to obtain the implementation results, the simulator was run with a network of 5 nodes. All results are averaged over 100 runs. Each real-time session reserves bandwidth enough to transmit an MPEG-I stream of 1.5 Mbit per second and lasts for five minutes. For comparison purposes, both implementation and simulation results were obtained for an idle network. Figure 7 indicates that the results from the simulator are very close to those obtained from the implementation.

5.1 Overhead due to Token Passing

Because of the hardware constraints in our lab, our performance results are limited to small-scale networks. To study the scalability behavior of the RETHER protocol, we used a realistic simulator that took values from real measurements, as configuration parameters. Figure 8 plots the NRT network access delay in a lightly loaded network against the size of the network, assuming that there are two MPEG-I streams. The protocol overhead increases linearly with the size of the network. For LANs supporting up to 20 nodes on a 10 Mbps Ethernet, a network access latency of 8ms seems to be reasonable for production operation.

5.2 Faster Hardware/Links

Here, we investigate the impact of faster network adaptor hardware and faster links on the performance of the RETHER protocol. Simulations were done for a network using 100 Mbps Ethernet and PCs with local buses (with 132 Mbytes/sec peak data rate). To simulate the 100Mbps Ethernet, transmission times were reduced by a factor of 10 from those used for the 10 Mbps Ethernet. For the local-bus architecture, in which the network card is on the CPU bus, the data copy overhead was reduced by a factor of 7, which is calculated by taking the ratio of the time for memory-to-network-card copy and that for memory-to-network-card copy.

Size (bytes)	100	500	1000	1500
Mem. to Mem. Copy(μs)	12	34	61	89
Mem. to NIC Copy(μs)	51	210	402	604

Table 3: Time in μsec to copy data over the CPU bus vs. over the I/O bus.

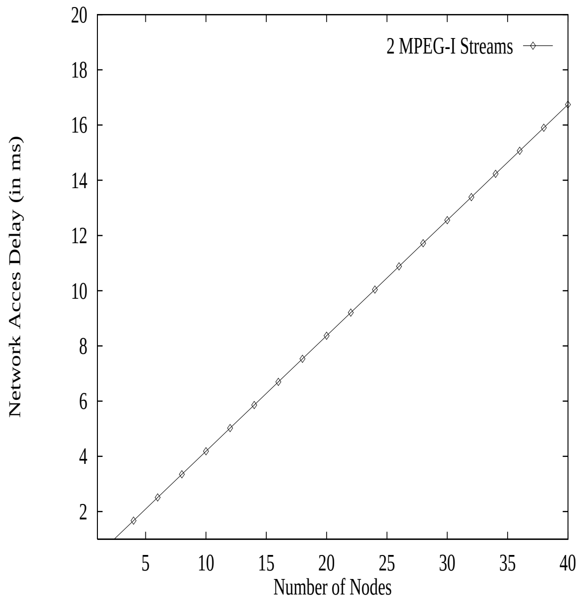

Figure 8: Scalability of RETHER with the Network Size.

Table 3 shows the speed difference between these two types of copy operations. Figure 9 shows the performance of the RETHER protocol under different combinations of fast/slow hardware/network-links. The figure demonstrates that faster link is more important than faster bus in terms of improving the overall performance. This is because the data transmission time dominates and the data copy time can often be masked through pipelining.

Considering the transport of compressed MPEG-I video data with an average bit-rate of 1.5 Mbits/sec, we found from simulations, that this configuration can support a maximum of 40 MPEG-I streams.

6 Stony Brook Video Server (SBVS)

SBVS is a videoserver currently under development. It uses a client-server architecture where a dedicated video server provides real-time delivery of video data to nodes on a LAN, from the storage subsystem. The kernel running on the server has two major parts – one that fetches data from the storage subsystem into the buffers in memory at the server and the other that empties data from the buffers into the network, in real-time. The latter part uses RETHER to get deterministic access to the network. All the nodes on the network, including the clients and the server run the RETHER kernel. The server reserves adequate bandwidth for each RT request that it has to service. In each cycle, the token visits the server once for every RT stream. At this time, the server delivers the data to the appropriate client.

7 Related Work

The idea of token passing was used in both IEEE 802.4 and IEEE 802.5 standards. In particular, the IEEE 802.4 [3] standard describes a token bus protocol and the use of a token as a means to resolve collisions has also been adopted by the IEEE 802.5 token ring [2] and the FDDI [4] protocols. These protocols, however, require specialized hardware and cannot be implemented over commodity Ethernet cards

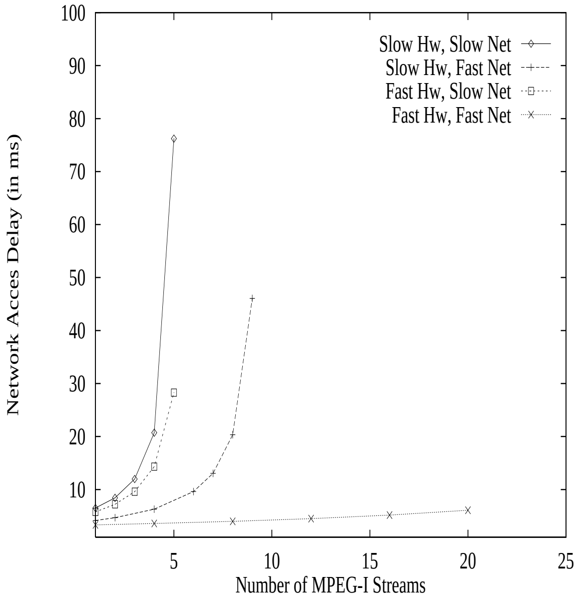

Figure 9: Performance of RETHER under Faster/Slower Hardware/Links. Network Size = 20 Nodes.

using the CSMA/CD protocol. These schemes have fixed token rotation cycles in which the token visits all the nodes in a specific order. Our protocol is an enhancement of the token bus protocol in that it adopts a mechanism that allows the token to visit a real-time node before its deadline expires, even if it has not yet visited all the other non-real-time nodes. The IEEE 802.4 standard does provide for assigning priorities (levels 0, 2, 4 or 6 with 6 being the highest) to different streams of traffic at each node. However the token visits all nodes in each cycle and the token holding times are maintained by timers in hardware. The most significant difference between RETHER and previous token-based protocols is the provision of distributed admission control, which extends the scope of the RETHER protocol from medium access to end-to-end bandwidth guarantee.

Earlier work in the area of supporting synchronous or real-time traffic on multi-access LANs can be found in [14], [18], [19]. These are window protocols which assume a slotted time axis. Implementation of this would again require hardware support, in particular global synchronized clocks. A lot of work has been done in the area of timed-token protocols since it was first proposed by Grow [13] in 1982. Protocols for guaranteeing synchronous deadlines based on timed-token protocols can be found in [5], [8], [16], [17]. These protocols assume an underlying token bus MAC protocol. Our work, however, is the software implementation of such a protocol over commodity network hardware, with enhancements to cater to applications with periodic channel access requirements. Also, RETHER addresses reliability issues and adopts solutions, the choices of which have been driven by the characteristics of an Ethernet-based network. Other implementation efforts are described in [11], [12], [15]. They also need specialized hardware in order to provide deterministic channel access.

In summary, although there have been numerous works on real-time local area network protocols, to our knowledge, our implementation is the first complete prototype that successfully demonstrates guaranteed bandwidth reservation on existing Ethernet hardware. The fact that there is a large installed base of Ethernet means that our completely software-based protocol provides an attractive and instant solution to support real-time applications on existing Ethernet-based LAN environments.

8 Conclusion

Most distributed multimedia applications such as videoservers and teleconferencing require resource guarantees from the underlying network. In this paper, we address this problem in the context of Ethernet, because it is the most prevalent LAN architecture. The Ethernet architecture is inherently incapable of providing deterministic network access because of its contention-based medium access protocol. In contrast, our protocol, RETHER, uses a timed token-bus approach to provide real-time performance guarantees to applications that require it. From the measurements of our implementation on a 10Mbps Ethernet, we found that it is possible to provide bandwidth guarantees to applications without significantly affecting the performance of non-real-time traffic, when up to 60% of the raw Ethernet bandwidth is reserved. In addition, the timeout values of higher-layer protocols in the operating system are preserved when no more than 60% of the bandwidth was reserved for real-time traffic. The RETHER protocol therefore can provide real-time performance guarantees required for higher-layer protocols such as the one described in [6]. It also permits distributed multimedia applications to run without any changes to existing hardware. Compared to other *network-adaptive applications*, our approach hides user-level programmers from the details of underlying network dynamics without specialized hardware support. The main contributions of this work are thus the design, implementation, and detailed evaluation of a distributed real-time bandwidth reservation protocol specifically for Ethernet.

In the future, we plan to extend this work in the following directions. First, we are currently integrating the RETHER subsystem with the Stony Brook Video Server (SBVS) project. Second, we will extend the RETHER protocol to run across bridges and routers so that real-time local-area inter-networking is possible. Finally, we are implementing the token monitoring and regeneration mechanism to handle node failures in a more robust fashion.

References

[1] Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. *ISO/IEC 8802-3, (4th edition)*, (1993-07-08).

[2] IEEE/ANSI Standard 802.5, Token ring access method and physical layer specifications. *IEEE Inc., New York.*

[3] IEEE/ANSI Standard 802.4 - 1985, Token passing bus access method and physical layer specifications. *IEEE Inc., New York*, 1985.

[4] Draft Proposed ANSI Standard, FDDI Token Ring Media Access Control. 1986.

[5] Agrawal, G., Chen, B., Zhao, W., and Davari, S. Guaranteeing synchronous message deadlines with timed token protocol. *Proc. of IEEE Intl. Conf. on Distributed Computing Sys.*, June 1992.

[6] Banerjea, A. and Mah, B. A. The real-time channel administration protocol. *Network and Operating Systems*

Support for Digital Audio and Video, pages 160–170, Nov. 1991.

[7] Boggs, D.R. et al. Measured capacity of an ethernet: myths and reality. *SIGCOMM '88 Symposium: Communications Architectures and Protocols., CA, USA. Computer Communication Review*, 18(4):222–34, Aug. 1988.

[8] Chen, B., Agrawal, G., and Zhao, W. Optimal synchronous capacity allocation for hard real-time communications with timed token protocol. *Proc. IEEE Real-time Systems Symp.*, Dec. 1992.

[9] Chiueh, Tzi-cker and Venkatramani, Chitra. Supporting real-time traffic on ethernet. *Proceedings of IEEE Real-time Systems Symp.*, Dec. 1994.

[10] Chiueh, Tzi-cker, Venkatramani, Chitra, and Vernick, Michael. Design and implementation of the stony brook video server. Technical report, SUNY at Stony Brook, 1995. Submitted to ACM Multimedia.

[11] Court, R. Real-time ethernet. *Computer Communications*, pages 198–201, Apr. 1992.

[12] Gopal, P.M. and Wong, J.W. Analysis of hybrid token-csma/cd protocol for bus networks. *Computer Networks & ISDN Syst.*, Sept. 1985.

[13] Grow, R.M. A timed token protocol for local area networks. *Proc. Electro/82, Token Access Protocols*, May 1982.

[14] Kurose, J.F., Schwartz, M., and Yemini, T. Controlling window protocols for time-constrained communication in multiple access environment. *Proc. IEEE Intl. Data Comm. Symposium*, 1983.

[15] Shieh, M, Sheu, J, and Chen, W. Decentralized token-csma/cd protocol for integrated voice/data lans. *Computer Communications*, pages 223–230, May 1991.

[16] Valenzano, A., Montuschi, P., and Ciminiera, L. Some properties of timed token medium access protocols. *IEEE Trans. on Software Engineering*, 16(8), Aug. 1990.

[17] Zhao, W. and Ramamritham, K. Virtual time csma protocols for hard real-time communications. *IEEE Trans. on Software Engineering*, 13(8), Aug. 1987.

[18] Zhao, W, Stankovic, J., and Ramamritham, K. A multi-access window protocol for transmission of time constrained messages. *Proc. of IEEE Intl. Conf. on Distributed Computing Sys.*, June 1988.

[19] Zhao, W, Stankovic, J., and Ramamritham, K. A window protocol for transmission of time constrained messages. *IEEE Transactions on computers*, 39(9), September 1990.