

EtheReal: A Host-Transparent Real-Time Fast Ethernet Switch

Abstract

Distributed multimedia applications require guaranteed quality of service (QoS) from the underlying networks. Most of the QoS research is focused on networks consisting of Layer-3 routers or ATM switches. Relatively little attention is given to the QoS support on existing local area networks, in particular Ethernet, which essentially forms the last mile of the end-to-end network bandwidth guarantee solution. This paper describes the design, implementation, and evaluation of a real-time Fast Ethernet switch called *EtheReal* that provides bandwidth guarantees to real-time applications running on Ethernet *without modification to the hardware and operating system on the host machines*. At the heart of the *EtheReal* switch architecture is a novel real-time connection setup protocol that is completely transparent to the host machine's OS, and thus allows the switch to be deployed in a network of heterogeneous machines running different OS platforms. The only dependency of *EtheReal* on the hosts is their support for TCP/UDP/IP. In addition, the *EtheReal* switch uses an Ethernet address swapping technique for real-time packets, similar to ATM to avoid the complexity of maintaining a global connection ID space. Because of the inherent CRC support built into Ethernet hardware, this technique significantly reduces the total processing overhead compared to address swapping at higher network layers. The current *EtheReal* switch prototype is fully operational, and is built with off-the-shelf PC hardware. It is capable of supporting up to 640 Mbits/sec across 4 ports, and has a per-hop connection establishment overhead of 0.1-0.6 msec and a 10 μ s non-real-time packet latency.

1 Introduction

In the past four years, our group has been developing technologies to support real-time bandwidth guarantees over Ethernet hardware. So far, we have successfully developed and demonstrated through working prototypes the *REETHER* protocol over 10 Mbps and 100 Mbps Ethernet links [1], over single-segment and multi-segment network environments [2, 3], and over wired and wireless Ethernet-like LANs [4]. The most interesting aspect of *REETHER* was its unique capability of supporting real-time bandwidth reservation and guarantee over existing Ethernet infrastructure without any special hardware support or modification. The only modification required was the replacement of the Ethernet driver with a *REETHER* driver on every machine on the network. Existing network protocols and applications would run without any changes.

Despite its apparent attractiveness, several problems arise during practical deployment of the *REETHER* technology. First, it is difficult to expose *REETHER*'s real-time connection abstraction to user-level applications without modifying the hosting kernel. This is a serious hurdle for extending *REETHER* to commercial OS platforms such as Windows and Mac, for which access to the kernel source code is all but impossible. Second, even though *REETHER* allows users to preserve existing Ethernet hardware infrastructure, installation of *REETHER* drivers on a massive scale is an expensive proposition from a system administration standpoint. Finally, with the continuing price decline of 10-Mbps Ethernet adapters and Ethernet switches, the switch-based network architecture is likely to dominate future LAN environments. As a result, the token passing mechanism implemented in *REETHER* is less appealing. Based on the above experiences and projections, the goal of the *EtheReal* project is to build a scalable real-time Ethernet switch that supports bandwidth reservation/guarantee over a switched LAN environment *without any hardware and OS modification*. Moreover, real-time applications can readily exploit the bandwidth guarantee mechanism using de facto socket-like Application Programming Interfaces (APIs). This paper reports the design, implementation and evaluation of the *EtheReal* switch, which is capable of supporting real-time network applications running on a diversity of OS platforms such as Windows/95, FreeBSD, and Linux, as long as they support TCP/UDP/IP and socket interfaces.

The major contribution of this research is an innovative real-time connection setup algorithm that installs per-connection bandwidth guarantee information along the network path from the source host to the destination host without involving the kernels on the host machines. This algorithm at once solves the first and second problems described in the previous paragraph. Compared to other existing solutions, the proposed approach supports bandwidth guarantees and not just packet prioritization, allows more efficient data transfer because real-time network packets are switched at the link layer, and is more general and scalable, in that its dependency on kernel support is minimal. Since *EtheReal* is designed specifically to work with host machines running heterogeneous OSes, we believe this work has the potential to, once and for all, close the gap of Quality of Service (QOS) support between Ethernet and ATM, and thus further vindicate the dominance of Ethernet technology in LAN environments.

The rest of this paper is organized as follows. Section 2 describes the design of the *EtheReal*

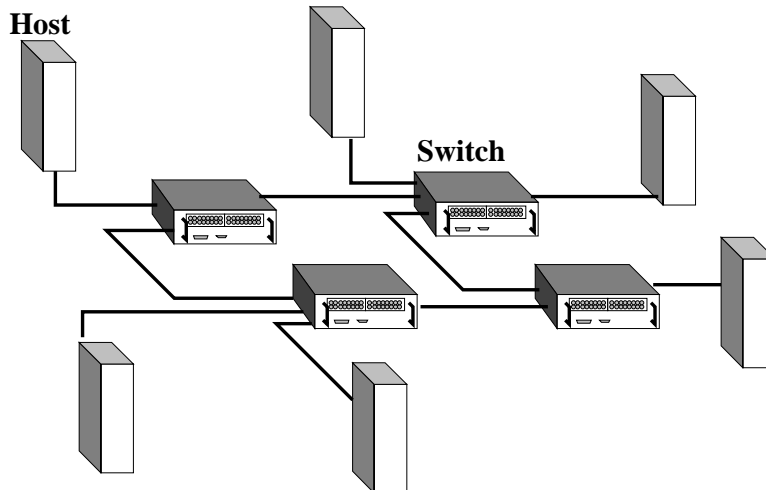


Figure 1: *The EtheReal switch architecture assumes that hosts are connected through point-to-point links using switches. This architecture eliminates to a large extent the non-determinism due to CSMA/CD.*

switch, including the connection establishment algorithm, and the real-time packet scheduling technique. Section 3 presents the implementation details of the first *EtheReal* switch prototype, which is a port switch using off-the-shelf hardware, and the host-end software support. Section 4 is the results and analysis of a detailed performance evaluation study of the prototype switch. Section 5 reviews earlier work on real-time support on Ethernet, and contrasts the proposed approach with these efforts. Section 6 concludes this paper with a summary of the main results and a brief outline of on-going work.

2 EtheReal Switch Design

2.1 Assumptions

From the beginning of the project, we aimed at developing a real-time Ethernet switch that provides guaranteed-bandwidth network services without the cooperation of host OS. All the real-time capabilities must be implemented through a combination of user-level libraries and switch hardware/software. Given this design goal, the design of the *EtheReal* switch is based on the following assumptions:

- The host machines on the network are connected through point-to-point links and Ethernet switches, as shown in Figure 1. These links can be either 10 Mbps or 100 Mbps. We believe this assumption makes economic sense because of the cost-effectiveness of Ethernet hardware. Moreover, it eliminates the major source of non-determinism due to the CSMA/CD medium access protocol, which the *RETHEP* protocol solves. This constraint can be relaxed by allowing a small number of hosts (e.g. fewer than 3) residing on the same Ethernet segment,

which is then connected to a *EtheReal* switch. The potential disadvantage of this approach is that the bandwidth reservation is no longer 100% guaranteed.

- The total network resource requirement from real-time and non-real-time applications on a single host is below its network link bandwidth. Therefore the lack of real-time process scheduling in the host kernel has little effect on bandwidth guarantees. In other words, the major sources of threat to the performance guarantees of real-time applications are assumed to come from applications running on other hosts.
- Two types of network services are supported in *EtheReal*'s service model: real-time constant-bit-rate connections, which require connection setup and bandwidth reservation before data can start to flow through the network, and non-real-time connections, on which packets are transported through the network without the need to establish a connection and thus without any performance guarantee. The key difference between the two is that the performance of the traffic on each real-time connection is protected against and thus independent of the network load, whereas non-real-time packets are serviced in a best-effort fashion. All real-time connections in the *EtheReal* architecture are simplex.
- Host machines are assumed to support the TCP/UDP/IP suite, on which existing network applications can readily run. The TCP/UDP/IP support provides necessary APIs for high-level applications to access the real-time networking capability of *EtheReal*, as well as a simple mechanism to identify packets belonging to a real-time connection, as we will show in the next subsection.

2.2 Real-Time Connection Setup

The most challenging aspect of the *EtheReal* switch architecture is to devise a connection setup mechanism that doesn't require any OS support at the hosts. Any real-time network architecture that provides per-connection performance guarantee requires the switches to maintain per-connection state for resource management. At run time, the switches associate packets with connection states through the connection ID fields in the packets. Since *EtheReal* switches work at the link layer, the connection IDs should be embedded in the link-layer headers, in this case Ethernet headers. So the fundamental challenge in developing *EtheReal*'s connection setup algorithm is to embed connection ID information in the Ethernet headers in a way that preserves normal Ethernet operation semantics and yet does not require kernel intervention at the sender/receiver machines.

The proposed real-time connection setup algorithm assumes that the host OS supports UDP/IP socket interfaces and ARP (Address Resolution Protocol) cache entry deposit and retrieval. When a real-time application attempts to set up a real-time connection, it sends the reservation request to a user-level process called the *Real-Time Communication Daemon* (RTCD) on the same host, which is responsible for real-time connection establishment and tear-down. The request includes the destination node's IP address, the port number, and desired QOS. For example, in Figure 2, an application running on the machine 130.245.21.1 (Host A) reserves a real-time connection to

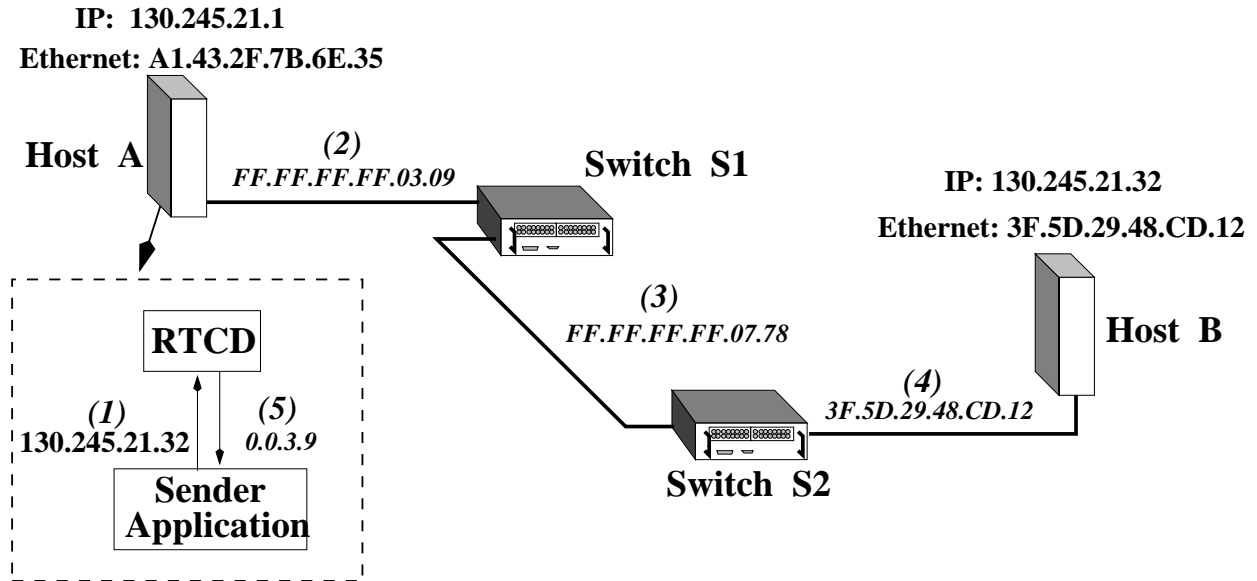


Figure 2: The proposed connection setup procedure in the EtheReal Architecture. Host A initiates a connection to Host B through the Real Time Communication Daemon (RTCD). The daemon at host A creates a Proxy IP address and a Proxy Ethernet address as a connection ID. Connection ID remapping occurs at each switch along the path from A to B. The numbers in the parenthesis indicate the logical ordering of the setup process.

the machine 130.245.21.32 (Host B). RTCD sends the reservation request in a UDP packet to the *EtheReal* switch to which the sending host (in this case A), is directly connected (in this case, it's S1). The packet includes the destination node's IP address, the QOS parameters, and a 2-byte *EtheReal* connection ID¹, in this case 03.09, which is created by RTCD and is unique on the link between A and S1. S1 forwards this request to the next switch, S2, and so on, until it reaches the destination node. S1's reservation packet contains the same information as the one from A except that the 2-byte *EtheReal* connection ID is changed to 07.78, which is unique on the link between S1 and S2. Each switch along the way creates a new connection ID on the output link for the real-time connection except the last switch that is directly connected to the destination host. On this switch, in this case S2, the *EtheReal* connection ID on the output link for the real-time connection is the destination host's Ethernet address, in this case 3F.5D.29.48.CD.12. Moreover, there is no need for S2 to forward the reservation request to the destination machine.

If the real-time connection request is admitted, the RTCD will receive an "success" UDP packet from the peripheral switch, Each switch along the path, creates a routing table entry with the corresponding QOS and connection ID information for the connection. With this indication, RTCD first creates a proxy IP address and a proxy Ethernet address for the real-time connection by prefixing the local connection ID with 1.1 and FF.FF.FF.FF, respectively. Then the RTCD inserts an Address Resolution Protocol (ARP) entry into the sending host's ARP cache that maps the

¹ *EtheReal* connection IDs are represented in hexadecimal format.

4-byte proxy IP address, 1.1.3.9, to the 6-byte proxy Ethernet address, FF.FF.FF.FF.03.09. Finally, RTCD returns the proxy IP address to the sender application, which then opens a normal UDP socket using the proxy IP address, and starts transferring real-time data through that socket.

When the IP layer of the host operating system receives UDP packets from the real-time application, it looks up the ARP cache to map the packet's destination IP address, in this case 1.1.3.9, to its Ethernet address, FF.FF.FF.FF.3.9, and puts it on the wire. Whenever an *EtheReal* switch such as S1 receives an Ethernet packet with an Ethernet destination address of the form FF.FF.FF.FF.X.Y, instead of interpreting it as an Ethernet address, it interprets the address' last two bytes as a real-time connection ID. Therefore S1 looks up the routing table using the 2-byte connection ID, in this case, 03.09, changes the packet's Ethernet destination address field to FF.FF.FF.FF.07.78, and passes it to S2. The packet is switched in the same way through the network until it reaches the switch that is connected to the destination host. This switch then changes the packet's Ethernet destination address field to the Ethernet address of the destination node, and forwards the packet over. Since the connection IDs of real-time packets are swapped from one switch to the next, *EtheReal* doesn't need to deal with globally unique connection IDs.

The switch connected to the sending host, in this case S1, needs to perform one additional task: modifying the destination IP address field of every real-time packet from the proxy IP address, in this case 1.1.3.9, used by the sender application to the receiver node's IP address, 130.245.21.32. Without this step, the destination node's IP layer will simply reject these real-time packets because of destination IP address mismatch. However, this processing also means that the *EtheReal* switch is no longer a pure layer-2 device.

The trick of *EtheReal's* connection setup mechanism is the exploitation of ARP cache to use certain ranges of Ethernet addresses as connection IDs. The portions of the IP and Ethernet address spaces, i.e., 1.1.*.* and FF.FF.FF.FF.**.** respectively, are carefully chosen to avoid conflicting with existing network protocols and applications. Note also that the receiver is not involved in the connection setup process because *EtheReal* is not concerned about resource reservation on the hosts. Finally, as far as the sending application is concerned, the real-time data stream is sent out through a normal UDP connection after the connection is established. That is, any platforms that support BSD-style sockets can support real-time applications on *EtheReal* without any modification.

2.3 Admission Control and Resource Reservation

To guarantee the quality of service of real-time connections, resources on *EtheReal* switches need to be reserved and dedicated to the service of these connections at run time. To avoid over-commitment, an admission control check is performed at each switch during the connection setup phase to ensure that the new connection not break the guarantees already made to previously admitted real-time connections. If the resources required by a new connection are available, the connection is admitted, resources are reserved, and the request is forwarded to the next hop, otherwise it is rejected and a reservation failure message is sent back to the source node, freeing up the reserved resources along the way.

The *EtheReal* architecture only provide bandwidth guarantees to network applications. Consequently, each real-time connection requires the reservations of three types of resources inside the switches for target QOS:

- The bandwidth on the switch’s backplane and the associated output link.
- The CPU cycles for scheduling the network connections to meet their performance objectives.
- The data buffer to absorb possible mismatches between data producing and consuming rates on the switches for each real-time connection.

These three types of resources are largely independent of one another and need to be reserved separately. Only when the reservations for all three resources succeed will the connection setup request on the switch be admitted. Reservation on each type of resource (bandwidth, CPU cycles, and buffer size) simply means marking the amount of resource required as ”used” and subtracting it from what is currently available. After the subtraction, if the amount of available resource is negative, the reservation fails; otherwise it succeeds. Given the hardware configuration, an important implementation issue of the *EtheReal* switch is to identify the total amounts of resources of these three types which are actually available after taking into account all hardware/software overheads.

2.4 Traffic Shaping

Once a real-time connection is set up, the switches service the connection’s traffic through the network with the target QOS as long as the traffic source observes its specified characteristics during connection establishment. To ensure the traffic source is well behaved, *EtheReal* requires that all the traffic from real-time applications be passed through a user-level library, which uses a leaky bucket algorithm to shape the traffic’s transmission rate to the reserved bandwidth. One problem with this approach is that although the data rate from the user-level library to the kernel is regulated, the fact that the kernel buffers data implicitly creates burstiness in the traffic sent out on the physical wire, and thus throughout the network. This traffic burstiness is smoothed by the switches by their per connection shaping and policing mechanisms. This isolates the performance impact of one connection’s burstiness from other connections and protects the switch in case of non-conformant traffic.

2.5 Limitations

The *EtheReal* architecture described so far does not address the collision problem due to simultaneous access attempts from both ends of a point-to-point link. Fortunately, almost all Fast Ethernet links support the full-duplex mode when used in a point-point topology. The full-duplex mode allows both ends of a link to transmit data simultaneously, thus eliminating the collision problem. The other weakness of the *EtheReal* architecture is that the QOS impact on a real-time connection due to other real-time and non-real-time traffic from the same host is not well controlled. For

example, if a file transfer session is active when a real time connection is started, the file transfer connection would fill up the IP buffers in the kernel, thus significantly delaying the real time connection. The fundamental issue here is how to prioritize packets from different connections on the same machine without accessing the kernel.

For a given real-time connection, the interference due to other real-time connections from the same host can be safely ignored because their corresponding traffic is assumed to be properly shaped, and thus would not over-consume the kernel's IP buffer resources. Among non-real-time traffic, TCP streams are easier to handle than UDP streams. Fortunately, TCP traffic accounts for the majority of non-real-time traffic, with NFS being the major source of UDP traffic.

The *EtheReal* switch controls the sending rates of TCP connections from a host by dropping packets belonging to these connections whenever the observed throughputs exceed their allocated bandwidths. The allocated bandwidth for TCP/UDP traffic from a host is calculated by subtracting the total amount of bandwidth reserved for real-time connections on that host from the link bandwidth. Dropping packets of TCP connections invokes TCP's congestion control mechanism, in particular, the slow start and congestion avoidance scheme, which reduces the TCP connection's transmission rate to a small value, exponentially grows the rate, and finally slowly increases it until it exceeds the allocated bandwidth, at which point the switch would again drop its packets. Therefore, each TCP connection's transmission rate is always kept under its allocated bandwidth. This ensures that real time connections originating from the same host get their share of the link bandwidth.

Unfortunately the packet dropping strategy does not work for UDP traffic because UDP does not support congestion control. For all practical purposes, NFS traffic is the only kind of UDP traffic of interest. The client-sent NFS traffic is small in volume and therefore would not cause serious interference. The server-sent NFS traffic, on the other hand, can lead to long bursts, and therefore may impact the performance guarantees of real-time traffic from NFS servers. We believe that the number of NFS servers is sufficiently small that it is feasible to modify them to adequately support both real-time traffic and normal file traffic from NFS servers. As a final note, even NFS itself is moving towards a TCP implementation, which could eliminate the interference problem from UDP traffic altogether.

3 Implementation

3.1 EtheReal Prototype Overview

The current *EtheReal* switch prototype is implemented on an Intel PentiumPro 200-MHz machine with 32 MBytes of main memory running Linux 2.0.30. Four Intel Express Fast Ethernet cards are used to connect to hosts and other switches through a point-to-point link topology. Therefore the current prototype implements a four-port Fast Ethernet switch using the 32-bit PCI bus as the backplane. Each packet traverses through the PCI bus twice, once from the input port to the main memory and the other from main memory to the output port. All data transfers on PCI are

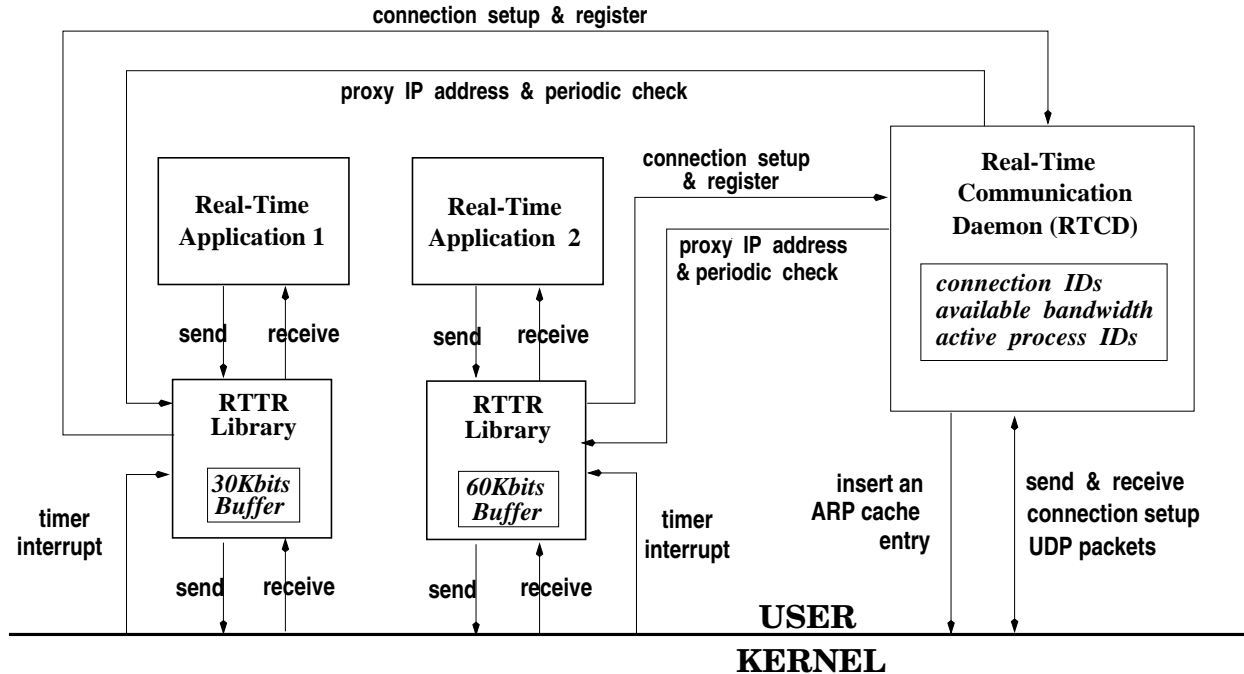


Figure 3: The interactions between *EtheReal*'s host software modules, *RTCD* and *RTTR* library, real-time applications and the underlying OS. In this case, we assume that the timer value is 10 msec, and Application 1 and 2 reserve 1,500 Kbits/sec and 3,000 Kbits/sec respectively. Since the *RTTR* library uses double buffering, the *RTTR* libraries for Application 1 and 2 allocate 30 Kbits and 60 Kbits, respectively, for leaky bucket buffering.

done through DMA transactions. Since each link is operating in full-duplex mode, the maximum cross-section bandwidth that the switch hardware can support is 800 Mbits/sec, i.e., 4 100-Mbps inputs and 4 100-Mbps outputs. All cards are put in the promiscuous mode so that the switch can take in all packets on the input links.

3.2 Host Support

Because *EtheReal* is targeted to work with multiple host platforms, the entire design has been carefully architected to ensure that the user-level software support on the host end could be successfully built on various OS platforms. There are two components to the host software support: Real-Time Communication Daemon (*RTCD*), which implements connection establishment and tear-down, and real-time data transmission/reception (*RTTR*) library, which implements traffic shaping. Figure 3 depicts the host software architecture for *EtheReal*.

The data structure manipulated by *RTCD*, local connection IDs and available bandwidth, is shared among real-time connections on a host, and thus should be centralized in one module which all connection establishments and teardowns have to go through. The only unusual OS support that *RTCD* needs is the ability to insert and retrieve ARP entries that map proxy IP addresses to proxy Ethernet addresses. Windows 95, Windows NT, and all UNIXes such as SUNOS, Linux, and

FreeBSD support this feature. An independent RTCD also greatly simplifies "orphan connection reclamation." In the current *EtheReal* implementation, all real-time applications have to register with the RTCD by sending their process IDs. The RTCD then periodically, every minute in the current implementation, checks if the registered real-time applications are still alive. If not, the RTCD assumes that the application is dead and the associated real-time connections are torn down.

The main purpose of the RTTR library is to shape the traffic from the application according to the reservation using a leaky bucket algorithm. Although it is possible to integrate this functionality into RTCD, a library implementation significantly reduces the data copying overhead due to inter-process communications. To implement traffic shaping, the library allocates N times the reserved bandwidth, where N is dependent on the maximum burst size from the application, and registers a timer value, T , and a handler with the kernel. The handler is a call-back routine that will be invoked by the kernel every T second. Every time the timer handler is called, it sends $T * B$ bytes of data from the buffer to the kernel, where B is the reserved bandwidth. Note that different platforms support different timer resolutions, for example, 10 msec for UNIX and 60 msec for Windows 95. Even on the same machine, the timer interrupts are not necessarily delivered at multiples of the timer values exactly. Therefore the RTTR library actually sends out $T_m * B$ bytes of data on each timer interrupt, where T_m is the measured interval between the previous and current timer interrupts. Our experience shows that the deviations in the timer values are typically sufficiently small that significant data bursts are unlikely.

The RTTR library also hides all the interactions with the RTCD during the connection setup time. From a real-time application's standpoint, it issues a connection setup call, receives a UDP socket descriptor if successful, and starts transferring data through the socket from that point on. There are two differences between *EtheReal's* UDP sockets and normal UDP sockets. First, *EtheReal's* UDP sockets are half-duplex, i.e., either for sending or for receiving, but not both, whereas normal UDP sockets are full duplex. Second, *EtheReal's* sending UDP sockets require explicit connection setup while normal UDP sockets and *EtheReal's* receiving UDP sockets don't.

Currently the host-end RTCD and RTTR library have been successfully developed on the following platforms: Linux, FreeBSD, and Windows 95. Since the code is POSIX compliant, and has minimal dependency on the OS, we believe porting RTCD and the RTTR library to other UNIXes and Windows NT should not pose any problems.

3.3 Real-Time Packet Switching

EtheReal switches distinguish among the following five types of Ethernet destination addresses. For packets with a *broadcast* destination address, it simply forwards the packets to every other port except the one on which it came. For packets destined to the switch, it only forwards ARP packets upwards and discards the others. This is to done to keep the switch transparent to the network. The switch also reserves the set of addresses between For packets with the destination address FF.FF.FF.FF.00.00 to FF.FF.FF.FF.00.0A, for various switch related activities like connection setup and inter switch messaging. For a packet whose destination address falls within the range of

FF.FF.FF.FF.**.** but is not a reserved address packet, the switch extracts the 2-byte connection ID, looks up the routing table based on the connection ID, and routes it to its per-connection scheduler queue at the corresponding output port. Such packets belong to real-time connections with previous bandwidth reservation. Finally, packets with a destination address that does not fall into any of the above four categories are non-real-time Ethernet packets and will be routed to the associated output ports based on the entire destination address. For the packets destined to the switch, and connection setup packets, *EtheReal* needs to look beyond the Ethernet header into the IP header field. Fortunately, these packets appear relatively infrequently and thus should not incur serious performance overhead. For real time data, only the peripheral switch connected to the sending host needs to manipulate the IP header, in particular, changing the destination IP address field from the proxy IP address to the receiver's IP address. All other switches in the network forward these packets strictly based on Ethernet headers only. For the broadcast and non real time packets, no fields beyond Ethernet headers need to be examined.

The real-time connection routing entries in the *EtheReal* switch contain the following information: *Output Queue*, *Output Connection ID*, *Receiver IP Address*, *QOS parameters*, and a *Swap* flag. For real-time packets (i.e., the fourth type), the switch uses the last two bytes of their destination address as the connection ID, indexes into the routing table, changes the two-byte connection ID field in the packets to the corresponding routing table entry's *Output Connection ID*, and if the *Swap* flag is on, modifies the packet's destination IP address field to the corresponding routing table entry's *Receiver IP Address*. The *Swap* flag is turned on only for the peripheral switch connected to the sending host at connection setup time. Finally the packet is buffered at the tail of the associated *Output Queue*, waiting for the scheduler to put it on the output link. Each real-time connection has its own per-connection queue allocated at connection setup time. For non-real-time packets, the routing table entries only contain the *Output Queue* field, and routing table lookup is based on the 6-byte destination Ethernet address. There is only one non-real-time output buffer at each output port, which is shared among all non-real-time packets going through the given output link.

The general architecture of the current *EtheReal* switch prototype is output queuing, since the PCI bus provides sufficient raw bandwidth for 4-port wire-speed switching. On each output port, there are a certain number of real-time queues, one for each active *EtheReal* connection, and a non-real-time queue for all non-real-time traffic. To guarantee each real-time connection's bandwidth reservation, *EtheReal* uses a cyclic round-robin scheduling algorithm, which visits each real-time connection queue at every output port within a switch during a pre-determined cycle time. In the current implementation, the cycle time is chosen to be 10 msec. On each visit, the switch's scheduler arranges a proportional amount of data from the real-time connection queue to be sent out. For example, a real-time connection with a 2 Mbits/sec bandwidth reservation should be allowed to send out 20 Kbits per 10-msec cycle. If the amount of data in the real-time connection queue is smaller than its allocated amount at the time of visit, the scheduler simply empties out the queue.

In each cycle, the scheduler first visits all real-time connection queues on all output ports, and

then visits the non-real-time queues. In addition, once the scheduler starts to service non-real-time packets, it never turns back to real-time traffic until the current cycle ends. In other words, the scheduler loops among the non-real-time queues for the remaining of the cycle time. This is similar to the original *RETHEER* system [1] in that they are both *non-work-conserving*, and thus minimize the possibility of data burst accumulation inside the network. The scheduling cycle boundaries are signaled by a timer interrupt. To avoid starvation of non-real-time packets and higher-layer protocol timeouts due to excessively long packets delays, the sum of all real-time bandwidth reservations is restricted to a certain percentage (in the current implementation, 60%) of the raw link bandwidth.

To ensure fairness to non real time queues at each output port, the scheduler sends out one packet from each queue, moves on to the next port and so on visits in a round robin manner till the end of the cycle. The scheduler continues visiting the non-real-time queues until the end of the cycle even when these queues are empty. This visiting strategy helps to reduce the average non-real-time packet latency.

While the scheduler is visiting the queues in a round-robin fashion, receiver interrupts from the Ethernet cards invoke routing table lookup and queuing for incoming packets. All data transfers on the PCI bus, between main memory and network interface cards, are performed through DMA transactions, which are physically set up by network card interrupt handling routines. Therefore, the CPU in the current *EtheReal* switch prototype needs to perform administrative functions such as connection setup, route discovery and network management, in addition to packet scheduling, DMA setup, and packet routing.

Each real-time connection queue is allocated a pre-determined buffer size which is set to twice the connection's maximum burst size. Packets from a real-time connection will be dropped in a switch if the corresponding queue is full, as they should since it means that the connection is sending data faster than it should. The size of the non-real-time connection queue at each output port is fixed at twice the product of the output link bandwidth and the cycle time. Non-real-time packets will be dropped if their target output port's non-real-time queue is full.

To reduce the performance impact of non-real-time traffic on real-time traffic on the same host, a preemptive packet discarding strategy is implemented to invoke the backoff mechanism in TCP. This technique is similar in spirit to Early Packet Discarding, although for different purposes. The *EtheReal* switch continues to estimate the transmission rates of real-time and non-real-time packets flowing through each peripheral input link² by counting the bytes from each category in the recent past. As soon as the non-real-time transmission rate exceeds the link's residual bandwidth, which is the result of subtracting the sum of real-time bandwidth reservations on that link from the total raw link bandwidth, the scheduler starts to drop non-real-time packets preemptively. The data structure uses a sliding-window count of non-real-time bytes transferred. It is updated on packet arrival and consulted during packet routing and forwarding.

²A peripheral input link is one that is connected to a host rather than a switch.

3.4 Routing

The current *EtheReal* switch supports a generic *backward learning* algorithm which inspects the source address of every incoming packet, and creates a routing table entry corresponding to the source address to indicate its associated output port is the input port through which the packet comes in, if such an entry doesn't exist. If the entry already exists, the update is ignored. If routing table look-up for an incoming packet fails, i.e., the switch does not know how to forward the packet, the switch sends out an ARP query, via Ethernet broadcast, with the packet's destination IP address as the argument. When the packet's destination host eventually responds to this ARP query, and the response packet reaches the querying switch, the switch will then be able to make a routing table entry using the same backward learning algorithm. In addition, *EtheReal* maintains an internal ARP cache of all such ARP responses. The use of ARP queries, in this case, is to force the unknown host to respond, rather than to perform IP/Ethernet address mapping. This ARP packet's source IP address is set to 0.0.0.0 and the ethernet address is set to the Ethernet hardware address of the link on which the ARP is sent. The reason that 0.0.0.0 is used as the source IP address is to prevent the responding host from putting the ARP query packet's source Ethernet address and source IP address pair into its ARP cache.

For real time connections, the connection setup message contains the IP address of the destination host. To route the connection setup message we use the ARP cache, typically populated by routing queries for non real time data. If the ARP cache does not contain an entry, we issue an ARP request to determine the destination Ethernet address and a route to the destination and cache the response. Further real time and non real time packets to this destination can use the cached entry, reducing their routing cost.

Currently the *EtheReal* switch prototype does not perform any distributed spanning tree discovery algorithm at initialization time to prevent infinite loops when broadcasting Ethernet packets. This feature will be incorporated in the next prototype.

4 Performance Evaluation

4.1 Experiment Setup

Four hosts are connected to the 4-port *EtheReal* switch prototype through point-to-point 100-Mbps Fast Ethernet links, each of which is operating in the full-duplex mode and therefore can support up to 100 Mbits/sec in both directions. Both the switch prototype and the hosts are Pentium-Pro 200-MHz machines with 32 Mbytes of main memory, running the Linux operating system. The RTCD and the RTTR library tune the hosts. This performance evaluation study aims to quantify the real-time QOS, specifically bandwidth guarantees, supported by *EtheReal*, and the implementation efficiency of *EtheReal*, in terms of the percentage of raw hardware capability that *EtheReal* can actually exploit.

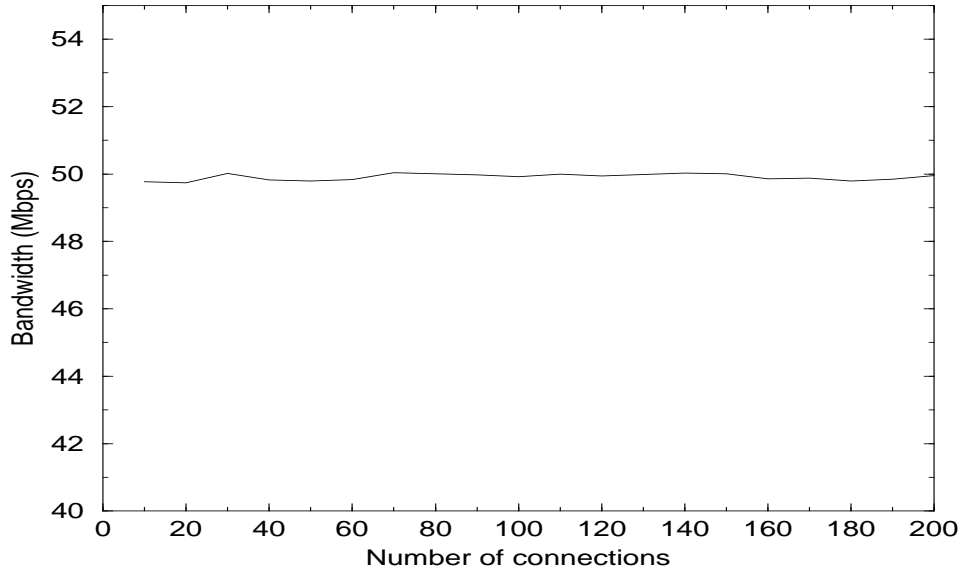


Figure 4: Average bandwidth measured at the destination host of a test real-time connection in the presence of a varying number of real-time connections between hosts distinct from those in the test connection. The test connection makes a reservation of 50Mbps and the interfering realtime connections have a total reservation of 100Mbps.

4.2 Real-Time Bandwidth Guarantees

To validate the bandwidth guarantees provided by the *EtheReal* switch prototype, two sets of experiments are performed. In the first set, we test the switch’s ability to guarantee a test real-time connection’s bandwidth reservation in the presence of external real time traffic. A test real-time connection makes a bandwidth reservation of 50 Mbits/sec through the switch. Then a varying number of real-time connections whose bandwidth reservations total 100 Mbits/sec run through the switch, between a pair of hosts distinct from those associated with the test connection. These real-time connections serve as interfering traffic. Figure 4 shows the average bandwidth observed at the destination of the test real-time connection for a varying number of interfering connections. Each data point is the average bandwidth observed over a 250 MB data transfer. As expected, the actual bandwidth available to the test real-time connection is rather close to 50 Mbits/sec, with about 0.6% deviation. The fact that the deviation does not have any correlation with the number of interfering connections suggests that the real-time packet scheduler in the switch functions correctly. Two factors cause the deviations in bandwidth guarantees. First, because there is no kernel support at the host, the application and the RTTR library may not be scheduled at exactly the right intervals. Second, the switch’s packet scheduling mechanism is implemented in software and therefore not necessarily exact, because of interrupts, which may preempt the scheduler, and timer inaccuracy, which causes scheduling cycles to be shifted.

In the second set of experiments, the interfering traffic runs between the same host pair as the test real-time connection. The goal of these experiments is to determine the effectiveness of the

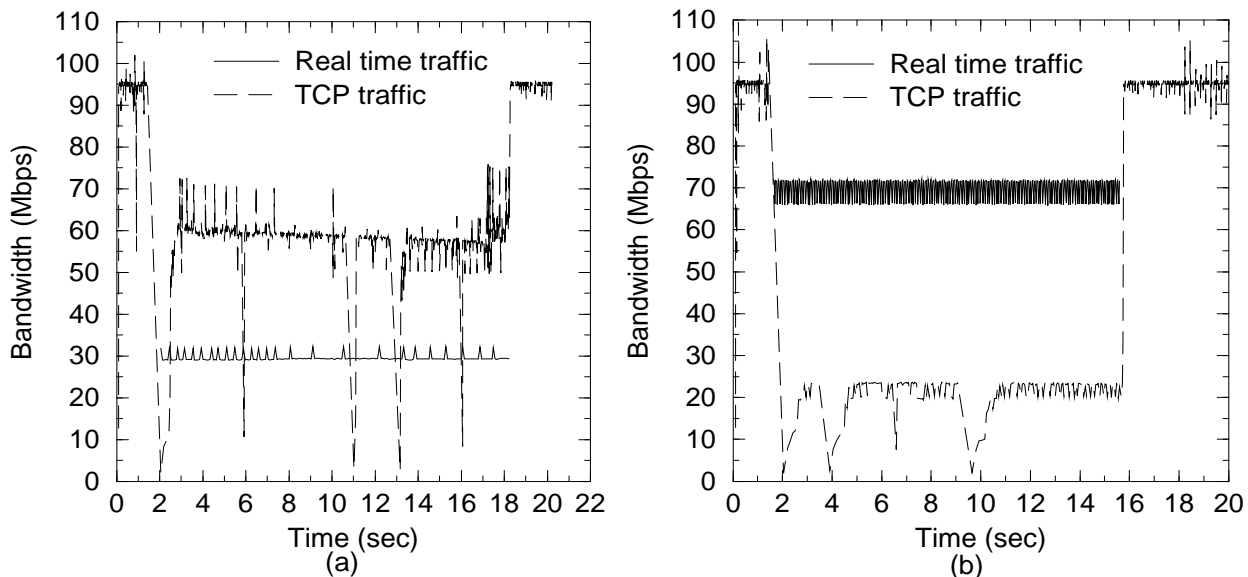


Figure 5: (a) 80-msec sliding window averages of bandwidth measurements to a TCP connection and a real-time connection when the real-time connection’s bandwidth reservation is 30 Mbps. (b) 80-msec sliding window averages of bandwidth measurements to a TCP connection and a real-time connection when the real-time connection’s bandwidth reservation is 70 Mbps.

preemptive non-real-time packet discarding technique and user-level traffic shaping and policing, in supporting bandwidth guarantees. Figure 5 shows the a 80-msec sliding window avergae of bandwidth measurements, to a TCP connection and the test real-time connection over the life time of the latter, as measured at the destination node when the bandwidth reservation for real time traffic is 30 Mbits/sec and 70 Mbits/sec, respectively. The TCP source attempts to use up the entire available capacity of the link. In both cases, as soon as the real-time connection is setup, the bandwidth available to the real time connections instantly jumps to the reserved amount, and the TCP’s share of the link bandwidth falls to the residual bandwidth. Similarly, immediately after the test real-time connection is closed, the link bandwidth is used entirely by the TCP connection. This figure demonstrates the effectiveness of the preemptive packet discarding technique in limiting non-real-time traffic to the residual link bandwidth.

One interesting observation from Figure 5 is that the TCP connection manages to stay at the residual link bandwidth most of the time, with the occasional dips when TCP’s congestion control mechanism is invoked. It turns out that there is a subtle interaction between the timer delivery accuracy in the RTTR library and the TCP connection’s average throughput. When the timer interrupts are delivered in time and therefore the real-time connection *consistently* uses up all the reserved bandwidth, the IP layer would not allow TCP to increase its congestion window beyond the residual link bandwidth, and therefore the TCP connection stays at the maximum window size. Occasionally the real-time connection sends data at a slower rate than the reserved bandwidth, and the TCP connection grabs the opportunity to increase the congestion window size, only to

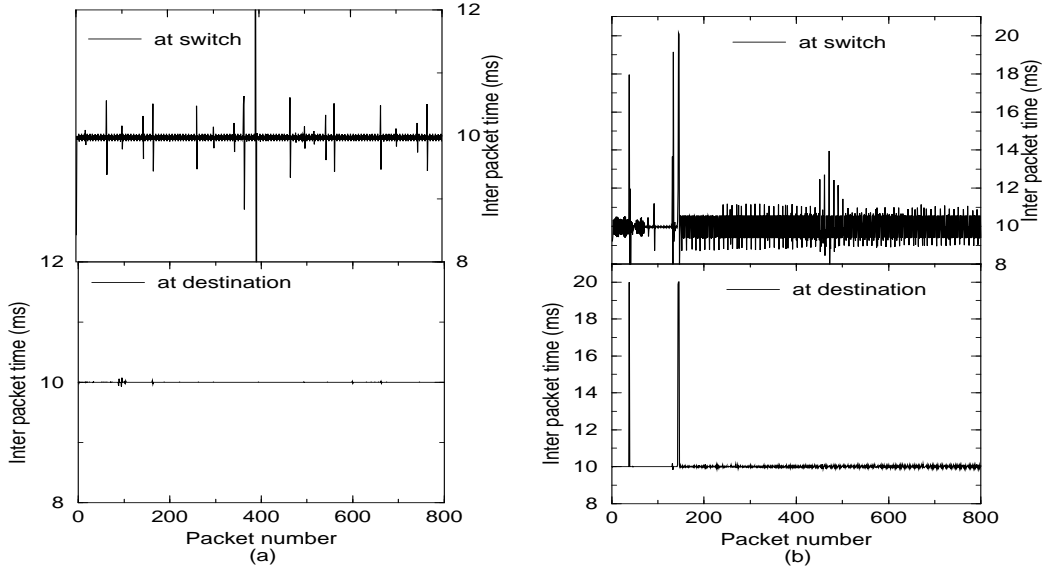


Figure 6: (a) Inter-packet arrival times as observed at the input of the switch and the destination host for a real-time connection with a bandwidth reservation of 50 Mbps and no internal interfering traffic; (b) Inter-packet arrival times as observed at the input of the switch and the destination host for a real-time connection with a bandwidth reservation of 50 Mbps and in the presence of internal interfering TCP traffic.

invoke the congestion control mechanism and take a serious hit in the throughput. These are the bandwidth dips of TCP traffic in Figure 5. The counter-intuitive lesson is that to keep the TCP connection's throughput at its maximum, the real-time connection should use up all its reserved bandwidth, so that TCP never has the chance to increase its window size and eventually is forced to back off.

From Figure 4 and Figure 5, it is clear that the *EtheReal* prototype indeed delivers the bandwidth guarantees regardless of the amount of internal and external traffic. In both cases, the aggregate bandwidth delivered to the real-time connection at the destination is within 0.6% deviation of its reservation. These figures also demonstrate the effectiveness of the packet scheduling and preemptive packet discarding algorithms at the switch, and the traffic shaping mechanism at the host.

Jitters, the differences between inter-packet delays, should be as small as possible for real-time connections to reduce the buffering requirement. Figure 6 shows the jitters as seen at the inputs of the switch and at the destination host when the data rate is 50 Mbps with no internal interfering traffic and with a 100 Mbps TCP source between same host destination pair. The figures show that while, internal interfering traffic increases the jitter seen by the switch, the destination always sees a smooth delivery of packets. This demonstrates the effectiveness of the switch's traffic shaping capability.

Total processes	Std. Dev. (msec)	Max. (msec)	Mean. (msec)
1	0.12	11.25	9.99
2	0.54	18.46	10.00
3	0.64	19.78	9.99
4	0.51	20.44	10.00
5	1.14	27.54	10.03

Table 1: *Timer inaccuracies in the presence of increasing CPU load. The ideal timer value is 10 msec.*

At the host end, accurate delivery of timers from the OS is essential to *EtheReal's* bandwidth guarantee mechanism. Timer variations cause real-time sources to burst data instead of sending a smooth flow. Since timer signals in UNIXes are only delivered when the application is executing, high CPU loads can cause large timer variations. For Windows 95, this is not a problem.

To test the accuracy of the timer in the presence of varying CPU load, we ran multiple copies of a timer-based application that sets a timer interrupt handler and blocks. This attempts to mimic the typical behaviour of a real-time application which spends most of the time waiting for data. Table 1 shows the mean timer intervals, the standard deviation and the maximum timer interval obtained when the number of such applications is increased from 1 to 5. The maximum CPU load reached was 0.5 on a scale 0 - 1.0. Table 1 shows that the timer inaccuracies are quite small under such low to medium load. However, disk I/O due to page faults, swapping or program startup can significantly increase the timer inaccuracy on UNIX-based systems since the kernel is blocked. Due to the real-time semantics of the timers, Windows and Windows 95 do not have this problem.

4.3 Prototype Implementation Efficiency

To scale up the *EtheReal* switch prototype to a 16-port system, we need to identify the bottlenecks of the current implementation. The real-time connection setup overhead is composed of two parts: (a) per-hop connection setup time and (b) time taken to get the connection setup success/failure message back to the originating host. The per hop connection setup time is 0.6 msec when we need to issue an ARP request (ARP cache miss) to determine the destination and 0.1 msec when we have an ARP cache hit. The average time taken to get the connection setup success/failure message back to originating host is 8 msec. This is due to the fact that the RTCD is a low CPU utilization process and consequentially, is allocated a smaller share of the CPU by the OS scheduler at the end hosts. In a typical LAN environment, the end-to-end real-time connection delay is within 10 msec.

For non-real-time packets, the end-to-end delay in the presence of real time traffic should be smaller than time-out values used by higher-layer network protocols or applications so that unnecessary timeouts are avoided. Figure 7 shows the relationship between the non-real-time packet latency through a switch and the amount of real-time bandwidth reservations on the switch. The increasing non-real-time packet latency is due to the increasing queuing time when the switch's

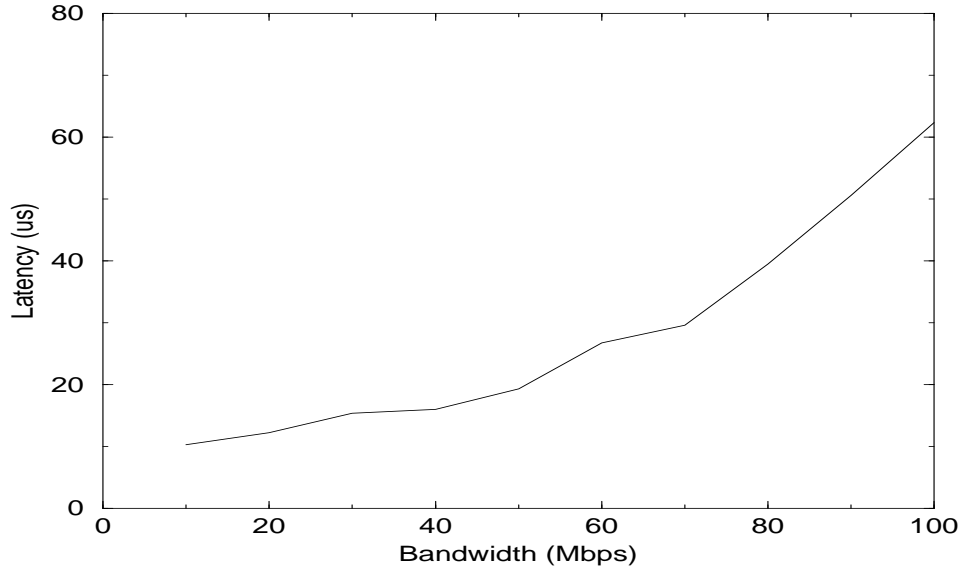


Figure 7: *Non real time packet latency in the presence of real time connections with increasing bandwidth. The non real time traffic rate is 50 Mbps.*

scheduler has more real-time packets to send out first. Under very light load, the non-real-time packet latency is about 10 μ sec per switch.

For admission control, the switch needs to determine the maximum limit imposed by each of its components. On the 32-bit/33 MHz PCI bus, the measured sustained bandwidth, including packet scheduling and data transmit receive overhead is 80.5 MBytes/sec. That means the total sum of real-time bandwidth reservations through a switch must be smaller than 40.25 MBytes/sec or 322 Mbits/sec, since each real-time packet goes through the PCI bus twice. The upcoming 64 bit/66 MHz PCI bus could potentially quadruple this bandwidth. As for the CPU, during each schedule cycle, the scheduler has to visit each real-time connection exactly once. If a real-time connection's buffer queue is not empty, the switch sets up a DMA transaction to transfer the 10-msec equivalent of its bandwidth reservation. The DMA transaction set up overhead is 1.79 μ sec. Also, each incoming packet invokes a receive interrupt handler, which performs routing table lookup and packet queuing whose total cost is 2 μ sec. Similarly, whenever a packet is put on the wire, a transmit interrupt occurs, which frees up a transmit buffer entry, and costs 2.2 μ sec. Assume that each real-time connection takes one scheduler visit, one transmit and receive interrupt, the CPU overhead required per real-time connection is thus 5.99 μ sec per schedule cycle. Therefore the CPU can only support $\frac{10msec}{5.99\mu sec} = 1669$ real-time connections at the maximum. The current switch prototype's CPU is a 200-MHz PentiumPro chip. But we believe the overall CPU overhead has more to do with the system architecture, in particular the access latency to network devices' registers, than with the processor architecture, and therefore should remain largely unaffected even when newer processors are used.

It is possible to increase the maximum number of real-time connections that can be supported

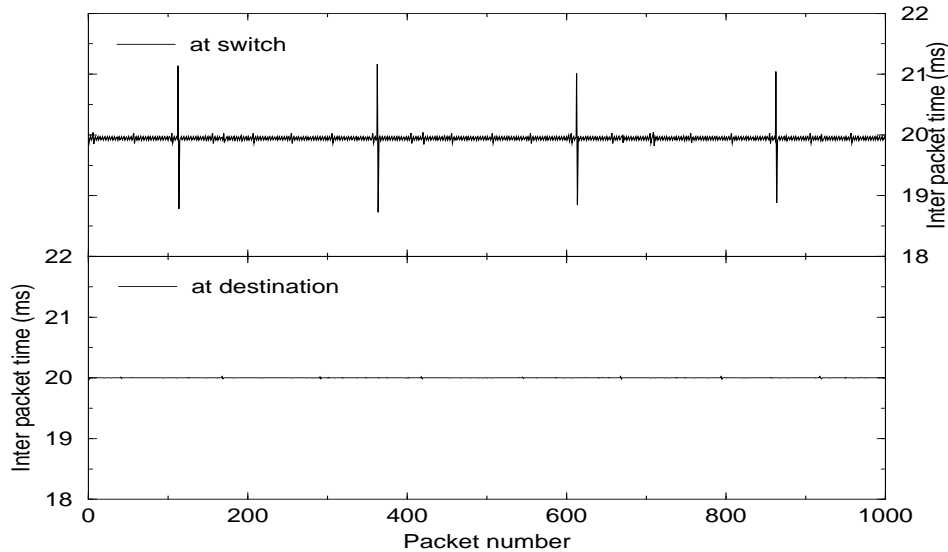


Figure 8: *Inter-packet arrival times at the input of the switch and the destination host for a real-time connection with a bandwidth reservation of 50 Mbps. The scheduler cycle time in this case is 20 ms.*

with a single-CPU switch by increasing the scheduler cycle time, because the same processing overhead is amortized over a larger transmission granularity. The disadvantage of this approach is that the end-to-end real-time packet delays increase accordingly. Moreover, the delay jitters may also increase. Figure 8 shows the inter-packet delays for a 50-Mbps real-time connection as seen at the input of the switch and the destination when the scheduler cycle time is 20 msec. Although the inter-packet delays do increase because of a larger cycle time, the delay jitters seem to remain unchanged, compared to Figure 6, which uses a 10-msec cycle time.

5 Related Work

Earlier works on providing bandwidth guarantees included Kopetz’s MARS system [8], which used a TDMA protocol to provide real-time guarantees on Ethernet, but is more restricted in its target domain, i.e., process control; and a variant of CSMA/CD [6] for real-time scheduling in distributed multiaccess broadcast communication channels. More recently, Hewlett-Packard developed a *Demand Priority Access* protocol [12] which is now the IEEE 802.12 standard. It is called 100VG-AnyLAN [5] and is designed to provide users with guaranteed bandwidth and low latency. It supports both the Ethernet 802.3 packet format as well as the token ring format. A 100VG network consists of hubs and nodes. Each node connects to a port on a hub in a star topology and hubs connect to each other in a tree topology. In the 100VG technology, the media access control is implemented in the hub. That is, when a node needs to send a data packet, it sends a transmit request to the hub. The hub then decides when the node can transmit the data packet. Two types of transmit requests can be made to the hub – normal priority and high priority. The hub cycles

through each of the requesting nodes in port order, permitting all the high priority requests to go through before any low priority requests.

Another proposal is National Semiconductor’s IsoEnet [9, 10]. This is a broadband network which includes a 10Mbps P-channel for normal Ethernet traffic, 96 64-Kbps B channels for real-time traffic, one 64-Kbps D-channel for signaling, and one 64-Kbps channel for control and maintenance traffic. The 96 B-channels can provide bandwidth guarantees to network applications because they are completely isolated from the CSMA/CD traffic. The IsoEnet is a slight variant of the Isochronous Ethernet IEEE 802.9 standard.

3Com’s Priority Access Control Enabled (PACE) [7] technology enhances multimedia applications by improving network bandwidth utilization, providing bounded latency and jitter, and supporting multiple traffic priority levels. PACE technology is designed to run on existing Ethernet LANs. It uses star-wired switching configurations where all the nodes are connected to the switch using their own dedicated 10Mbps Ethernet link. The PACE technology is contained in the switch which controls traffic for an Ethernet workgroup in a way transparent to each end-system adapter. Because the real-time priority mechanism is provided by the switch, there is no need to change the network hardware on the desktop machines. However, the network drivers need to be modified.

IEEE 802.1p [11] standard describes a priority-based approach to support different traffic classes with different performance requirements, from best effort, to controlled load and guaranteed services. The standard does not include any "hard guarantee" mechanism, with the rationale that it is probably not necessary in a LAN environment. The networking software on the client machines need to be modified to exploit the priority mechanism.

Compared to previous efforts, *EtheReal* is unique in two aspects. First, it provides true bandwidth guarantees rather than packet prioritization. As such, *EtheReal* offers better protection for real-time network connections from traffic load fluctuations on the network. Secondly, *EtheReal* is designed to be truly "plug-and-play" in the sense that it is completely transparent to the host OS and network interface hardware. Consequently, it provides a smooth migration path for practical deployment of this technology.

6 Conclusion

This paper describes the design, implementation and evaluation of a real-time Fast Ethernet switch called *EtheReal*, which supports bandwidth guarantees over a switched LAN environment *without OS and network hardware modification at the host end*. An important feature of the *EtheReal* architecture is that real-time applications can exploit the guaranteed QOS using existing UDP socket interfaces. Therefore we expect existing distributed multimedia applications that require bandwidth guarantees can be ported to the *EtheReal* architecture with only minor modifications. Existing non-real-time applications and network protocols can still run on the *EtheReal* architecture as they are.

The most innovative aspect of the *EtheReal* design is that it confines the QOS support in two

modules: the switches and the user-level libraries and daemons at the hosts. This feature renders the *EtheReal* architecture completely host platform-independent, and thus significantly simplifies the deployment and management cost of this new technology. The key enabling technique in the *EtheReal* architecture is a novel host-transparent connection establishment mechanism that exploits ARP caching to map connection IDs to a selected range of Ethernet addresses. At run time, *EtheReal* switches route real-time packets according to their connection IDs and schedule them for transmission based on their bandwidth reservations.

We have built a 4-port *EtheReal* switch prototype using off-the-shelf PC and network hardware. The prototype implementation is completely software based and is fully operational now. The performance study we did on the prototype shows that the switch can successfully provide bandwidth guarantees to real-time connections regardless of other traffic from the same sending host or other hosts that go through the same switch. The packet latency and backplane bandwidth measurements indicate that the current implementation is highly efficient, and further performance improvements can only come from better hardware.

Currently we are pursuing the following directions related to this work. First, we are extending the current 4-port prototype to a 16-port prototype by using four 4-channel Ethernet cards from Matrox. The 4-channel Ethernet card has a local backplane and CPU/memory, and thus forms a mini-switch on its own. Managing the local memory on the cards and the main memory on the motherboard, scheduling the data transfer transactions on local backplanes and the main PCI bus, and coordinating Ethernet routing table updates among the cards, all present interesting technical challenges. Second, in this work we do not address the fault tolerance and security issues raised by the *EtheReal* architecture, which need to be explored in more depth. In addition, the support for guaranteed multicast and broadcast flows need to be incorporated in the *EtheReal* architecture. Finally, the integration of *EtheReal* with real-time IP protocols such as RSVP is mandatory to qualify *EtheReal* as a true "last-mile" solution for end-to-end network QOS guarantees.

References

- [1] This entry is kept anonymous for blind review.
- [2] This entry is kept anonymous for blind review.
- [3] This entry is kept anonymous for blind review.
- [4] This entry is kept anonymous for blind review.
- [5] A.R. Albrecht and P.A. Thaler. Introduction to 100VG-AnyLAN and the IEEE 802.12 local area network standard. *Hewlett-Packard Journal*, 46(4), Aug. 1995.
- [6] Hermant, J.-F., Le Lann, G., and Rivierre, N. A general approach to real-time message scheduling over distributed broadcast channels. *Proceedings 1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation, ETFA '95, Paris, France*, pages 191–204, Oct. 1995.

- [7] The 3COM Technical Journal. 3Com's New PACE Technology. <http://www.3com.com/files/mktg/pubs/3tech/195pace.html>, 1996.
- [8] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. Distributed fault-tolerant real-time systems: the Mars approach. *IEEE Micro*, pages 25–40, Feb. 1989.
- [9] D. Minoli. Isochronous Ethernet : Poised for launch. *NETWORK COMPUTING*, page 156:162, Aug. 1993.
- [10] Xiaonong Ran and W.R. Friedrich. Isochronous LAN based full-motion video and image server-client system with constant distortion adaptive DCT coding. *Proceedings of the SPIE - The International Society for Optical*, 2094:1030:41, 1993.
- [11] Seaman, M., Smith, A., and Crawley, E. Integrated Services over IEEE 802.1D/802.1p Networks. *Internet Draft*, <http://ds.internic.net/internet-drafts/draft-ietf-issll-802-01.txt>, June 1997.
- [12] Greg Watson and et al. The demand priority mac protocol. *IEEE Network*, page 28:34, Jan. 1995.