

Declaration of Rachel J. Watters on Authentication of Publication

I, Rachel J. Watters, am a librarian, and the Head of Resource Sharing for the General Library System, Memorial Library, located at 728 State Street, Madison, Wisconsin, 53706. Part of my job responsibilities include oversight of Wisconsin TechSearch (“WTS”), an interlibrary loan department at the University of Wisconsin-Madison. I have worked as a librarian at the University of Wisconsin library system since 1998, starting as a graduate student employee in the Kurt F. Wendt Engineering Library and WTS, then as a librarian in Interlibrary Loan at Memorial Library. I began professional employment at WTS in 2002 and became WTS Director in 2011. In 2019, I became Head of Resource Sharing for UW-Madison’s General Library System. I have a master’s degree in Library and Information Studies from the University of Wisconsin-Madison. Through the course of my studies and employment, I have become well informed about the operations of the University of Wisconsin library system, which follows standard library practices.

This Declaration relates to the dates of receipt and availability of the following:

Varadarajan, S. and Tzi-cker, C. (1999). Automatic fault detection and recovery in real time switched *Ethernet* networks. Proceedings, IEEE INFOCOM '99: The Conference on Computer Communications, Volume 1, Eighteenth Joint Conference of the IEEE Computer and Communications Societies, The Future is Now, 21-25 March 1999, Hotel Sheraton, New York, NY, USA, p. 161-169.

Standard operating procedures for materials at the University of Wisconsin-

Madison Libraries. When an volume was received by the Library, it would be checked

Declaration of Rachel J. Watters on Authentication of Publication

in, stamped with the date of receipt, added to library holdings records, and made available to readers as soon after its arrival as possible. The procedure normally took a few days or at most 2 to 3 weeks.

Exhibit A to this Declaration is a true and accurate copy of the inside cover page with library date stamp of *Proceedings, IEEE INFOCOM '99: The Conference on Computer Communications, Volume 1, Eighteenth Joint Conference of the IEEE Computer and Communications Societies, The Future is Now, 21-25 March 1999, Hotel Sheraton, New York, NY, USA* (1999), from the University of Wisconsin-Madison Library collection. Exhibit A also includes an excerpt of pages 161 to 169 of that volume, showing the article entitled *Automatic fault detection and recovery in real time switched Ethernet networks* (1999). Based on this information, the date stamp on the inside cover page indicates *Automatic fault detection and recovery in real time switched Ethernet networks* (1999) was received by the Kurt F. Wendt Library, University of Wisconsin-Madison Libraries on April 20, 1999.

Based on the information in Exhibit A, it is clear that the volume was received by the library on or before April 20, 1999, catalogued and available to library patrons within a few days or at most 2 to 3 weeks after April 20, 1999.

Members of the interested public could locate the *Proceedings, IEEE INFOCOM '99: The Conference on Computer Communications, Volume 1, Eighteenth Joint Conference of the IEEE Computer and Communications Societies, The Future is*

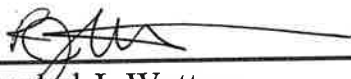
Declaration of Rachel J. Watters on Authentication of Publication

Now, 21-25 March 1999, Hotel Sheraton, New York, NY, USA (1999) publication after it was cataloged by searching the public library catalog or requesting a search through WTS. The search could be done by title and/or subject key words. Members of the interested public could access the publication by locating it on the library's shelves or requesting it from WTS.

I declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Date: April 15, 2024

Memorial Library
728 State Street
Madison, Wisconsin 53706



Rachel J. Watters
Head of Resource Sharing

1999 IEEE INFOCOM

Volume 1 of 3



89062718234



B89062718234A

1999 IEEE INFOCOM

VOLUME 1 of 3



99

TK
5105.5
I48
.v.1999
1

0-7803-5418-4

1999 IEEE INFOCOM

Volume 1 of 3



**KURT F. WENDT LIBRARY
COLLEGE OF ENGINEERING**

APR 20 1989

UW-MADISON, WI 53706

Proceedings

IEEE INFOCOM '99

The Conference on Computer Communications

Volume 1

**Eighteenth Annual Joint Conference
of the IEEE Computer and Communications Societies**

The Future Is Now

21 - 25 March 1999
Hotel Sheraton, New York, NY, USA

Sponsored by
IEEE Computer Society
IEEE Communications Society



IEEE COMMUNICATIONS SOCIETY



Kurt F. Wendt Library
University of Wisconsin - Madison
215 N. Randall Avenue
Madison, WI 53706-1688

Proceedings IEEE INFOCOM'99 The Conference on Computer Communications

Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limit of U.S. copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923. For other copying, reprint, or republication permission, write to the IEEE Copyrights Manager, IEEE Operations Center, 445 Hoes Lane, Piscataway, NJ 08855-1331. All rights reserved. Copyright © 1999 by The Institute of Electrical and Electronics Engineers, Inc.

IEEE Catalog Number: 99CH36320 (softbound)
99CB36320 (casebound)
99CH36320C (CD-ROM)

ISBN Softbound: 0-7803-5417-6
Casebound: 0-7803-5418-4
Microfiche: 0-7803-5419-2
CD-ROM: 0-7803-5420-6

ISSN: 0743-166X

Additional copies of this publication are available from

IEEE Operations Center
P. O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331 USA

1-800-678-IEEE
1-732-981-1393
1-732-981-9667 (FAX)
833-233 (Telex)
email: customer.services@ieee.org

Automatic Fault Detection and Recovery in Real Time Switched *Ethernet* Networks

Srinidhi Varadarajan Tzi-cker Chiueh

Department of Computer Science
State University of New York,
Stony Brook NY 11794-4400
(srinidhi, chiueh)@cs.sunysb.edu

ABSTRACT

EtheReal is a real-time Fast *Ethernet* switch architecture that provides bandwidth guarantees to distributed multimedia applications without OS or hardware modifications on the host machines. It implements true link-layer multicast, and offers a natural match to support network-layer QoS protocols such as RSVP. Because real-time performance guarantees fundamentally require state to be installed inside the network, link/switch failures could lead to significant disruption to the QoS promised to the user applications. This paper describes the fault detection and recovery mechanism supported by the *EtheReal* architecture, and reports on the performance measurements of the initial prototype implementation. The heart of *EtheReal*'s fault detection and recovery mechanism is a fast spanning tree reconfiguration algorithm to reduce the total fault recovery time, and a delayed link inactivation scheme that allows real-time connections which are not affected by the failed links/switches to continue to exist, even though some of the links are marked as "blocked" in the new spanning tree topology. Measurements on the prototype show that the fault detection and recovery time on a network whose diameter is 10 hops are 220ms and 31ms, respectively. This combined delay corresponds to a minor jitter in real-time audio/video communication, and is a significant improvement over the standard IEEE 802.1d implementation, which takes on the order of 30 sec.

1 INTRODUCTION

With the availability of powerful desktop computers, distributed multimedia applications are becoming commonplace. In contrast to best effort traffic, multimedia traffic places strict resource requirements in terms of bandwidth, end to end delays and delay variance. A large body of the research effort in networks is concentrated on providing Quality of Service (QoS) guarantees over wide area networks at the internetwork layer, assuming that the underlying medium access layer is capable of delivering such guarantees. *Ethernet*, which forms the last mile of the network is incapable of guaranteeing any performance bound. QoS guarantee mechanisms over *Ethernet* are needed for an end-to-end resource guarantees.

A connection-oriented mechanism is needed to deliver hard resource guarantees, since it ensures that traffic flows through a fixed path on which resources have been reserved to deliver the guarantee. Our *EtheReal*[11] architecture provides a transparent connection-oriented mechanism for delivering

bandwidth guarantees over switched *Ethernet* networks without any changes to the operating system/device drivers or network hardware on the hosts connected to the network. The functionality for establishing real time connections, maintaining resource reservations and transferring best effort traffic is implemented in *EtheReal* switches.

A connection-oriented network maintains distributed state associating connection identifiers with their resource reservations. In the case of network failures, automatic fault detection and recovery mechanisms are needed to recover the distributed state and reestablish broken connections. However, *real time* connection oriented networks place additional constraints on their fault detection and recovery mechanisms. First, to minimize the effect of a fault, the fault detection mechanisms have to be responsive. Long fault detection times may void any resource guarantees offered by the network. Secondly, the fault recovery mechanism should constrain the effect of the fault to the affected link(s)/switch(es) and the connections through them, while allowing normal operation in the rest of the network. Finally, the fault recovery mechanism has to be fast, so that the affected connections can be reestablished with minimal interruption in service.

Unlike the case for layer 3 devices, routing loops in transparent switched *Ethernet* networks pose serious problems. Because transparent layer 2 devices cannot maintain a hop count field to avoid loops, they superimpose a logical spanning tree upon the physical topology and transfer data only on the links belonging to the spanning tree. When new links are detected or links in the spanning tree fail, the spanning tree has to be rebuilt. Since routing has to wait till the spanning tree has been built, spanning tree rebuild constitutes the major part of the time spent in fault recovery.

Current transparent switched *Ethernet* networks use the IEEE 802.1d distributed spanning tree algorithm to discover a loop free network topology and recreate it, in the event of changes. IEEE 802.1d was written with best effort traffic in mind and has several drawbacks in a real time connection oriented environment. First, it uses global timers to detect the completion of the spanning tree build phase. Because the size

of the network is not known a priori, these timers are conservatively large, resulting in a large fault recovery time. Secondly, the spanning tree can change after a reconfiguration. Because data can only be transmitted through links belonging to the new spanning tree, all preexisting connections through the network have to be torn down.

While there has been previous work on fault tolerance in transparent switched *Ethernet* networks, to the author's knowledge, there has been no work on fault detection and recovery mechanisms for real time connection oriented switched *Ethernet* networks. In this paper we present our software based fault detection and recovery mechanism that delivers fast spanning tree reconfiguration and the ability to send data through links that do not belong to the spanning tree, thus localizing the effect of network faults. Our prototype module is implemented over *EtheReal* and has been operational for over 4 months.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the *EtheReal* architecture for delivering bandwidth guarantees over switched *Ethernet*. In section 3, we present our approach to fault detection and recovery in real time switched *Ethernet* networks. Section 4 presents an implementation overview and performance evaluation of our approach. In section 5, we present related work in the area. Finally, we conclude the paper in section 6 with the main results from our work.

2 ETHEREAL ARCHITECTURE

2.1 ASSUMPTIONS

The primary research goal of *EtheReal* is to deliver bandwidth guarantees transparently to hosts connected over a switched *Ethernet* network. The real time capabilities of *EtheReal* are implemented as user level libraries running on the hosts and *EtheReal* switches, which establish connections and reserve resources to deliver the guarantees. Given the above constraints, *EtheReal* makes the following assumptions:

1. The host machines are connected to *EtheReal* switches by point to point links as shown in . These links run at either 10 or 100 Mbps. This assumption makes sense due to the cost effectiveness of *Ethernet* hardware and removes the main source of non-determinism; congestion. This constraint may be relaxed by running a small number (2-4) of hosts on the same segment, but "hard" guarantees can no longer be offered.
2. Sufficient compute resources are available on the hosts connected to the network to satisfy their bandwidth guarantees. It is assumed that the lack of real time process scheduling on the connected hosts does not affect the delivery of the guarantee.
3. Host machines run the TCP/UDP/IP suite. This provides the necessary APIs for applications to specify and use the

real time bandwidth guarantee capabilities of *EtheReal* as well provides mechanisms to distinguish between packets belonging to different real time connections and best effort packets.

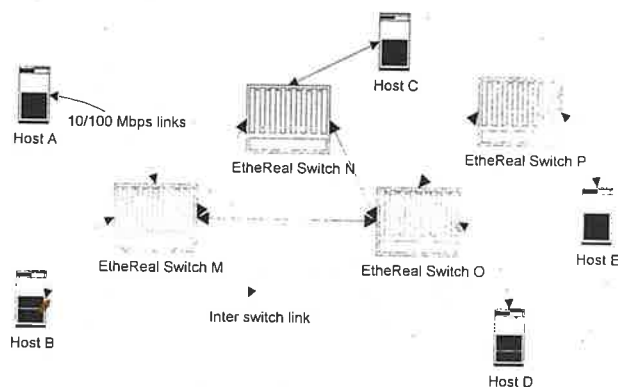


Figure 1: *EtheReal* Network Architecture

EtheReal is designed to support real time traffic along with existing best effort delivery traffic, delivering independent guarantees to both of them. *EtheReal* offers two service classes; a variable bit rate (RT-VBR) class specified by its average traffic rate and its maximum burst size and a best effort (BE) class intended for existing non real time applications. *EtheReal* connections are simplex in nature.

2.2 REAL TIME CONNECTION SETUP

Connection oriented network architectures need to reserve resources such as link bandwidth and buffers at the network switches to deliver their guarantee. At run time, the switches associate packets, with their QoS using connection ids embedded within the packets. Since *EtheReal* operates at the link layer, the embedded connection id. should be in the link layer i.e. the *Ethernet* header. The fundamental problem here is to change the *Ethernet* header without changing the semantics of regular *Ethernet* operation and to do this without changing the host OS.

Our real time connection setup protocol assumes the presence of the Address Resolution Protocol (ARP) as a part of the TCP/UDP/IP suite. We assume that we can insert and delete entries from the ARP cache maintained by the OS. When a real-time application attempts to set up a connection, it sends the reservation request to a user-level process called the Real Time Communication Daemon (RTCD) on the same host. The RTCD is responsible for connection setup and tear down. The connection request contains the destination node's IP address, the destination IP port number and the desired QoS. Figure 2 shows an example where a sender on host A wants to open a real time connection to a receiver on host B. When the RTCD on host A receives a connection request, it contacts its neighboring switch, and sends it the destination IP address, the QoS parameters and an *EtheReal* connection id, in this case 0309. The connection id is generated by the RTCD and

is unique on the link between A and M. Switch M then forwards this request to switch N and so on till the destination host is reached. When switch M forwards the connection setup packet, it specifies a new connection id, one that is unique on the link between M and N. This connection remapping, is similar to the mechanism used in ATM and eliminates the need for a centralized connection id allocator. Each switch performs connection remapping, except the periphery switch, which is directly connected to the destination host. On this switch, the output connection id is the *Ethernet* address of the destination machine, in this case A1:78:32:73:89:CD. Since, *EtheReal* connections are simplex in nature, there is no need to forward the request to the destination host.

If the connection setup is successful, the periphery switch sends a success packet back to the RTCD on the sender. Each switch in the connection path also sets up a routing entry with the connection id, and the QoS parameters. The RTCD then creates a proxy *Ethernet* address of the type FF:FF:FF:FF:XX:YY and a proxy IP address of the type 1.1.XX.YY where XX.YY is a 16 bit connection id. The RTCD uses the ARP cache deposit mechanism to bind the proxy *Ethernet* address to the proxy IP address. It then returns the proxy IP address along with a “connection setup successful” flag to the application, which requested the connection. The application then opens a normal UDP connection using the proxy IP address and starts transferring real time data.

When the IP layer on the host receives packets going to the proxy IP address, it uses the ARP cache to lookup the corresponding destination *Ethernet* address. Since the RTCD has inserted a binding to the proxy *Ethernet* address, IP uses this address. In the example above, the application uses the proxy IP address of 1.1.3.9. When data is sent using this address, IP uses the proxy *Ethernet* address FF:FF:FF:FF:03:09 as the destination address. When the *EtheReal* switch receives a packet of the form FF:FF:FF:FF:XX:YY, it interprets the last two bytes as a connection id¹ and looks up the corresponding routing table entry. In this example, when packets with the connection id 03:09 are received, switch M looks up the routing table, changes the connection id to 01:03 and sends the data to switch N. The packet is switched through the network till it reaches the periphery switch connected directly to the receiving host. This switch, in this case N, changes the *Ethernet* address to the address of the destination node and forwards the packet to it.

Since the application on the sender opened a connection to the proxy IP address (1.1.3.9), the IP layer header contains

¹ An exception is made when the *Ethernet* address is FF:FF:FF:FF:FF:FF. This is the *Ethernet* broadcast address and the data is broadcast to all enabled ports.

the proxy IP address as the destination address. If the packets were forwarded without change, they would be rejected at the destination node due to IP address mismatch. To avoid this, the periphery switch connected to the sender switches the IP address in the packet to the IP address of the destination, in this case 130.245.20.2. However, this means that the peripheral *EtheReal* switches are no longer purely layer-2 devices.

The novel part in *EtheReal*'s connection setup procedure is the exploitation of the ARP cache to cache ranges of addresses of the form FF:FF:FF:FF:XX:YY to 1.1.XX.YY for use as connection ids. This forms the basis of our low cost mechanism for embedding connection ids into packets. As far as the sending application is concerned, it sends its real time

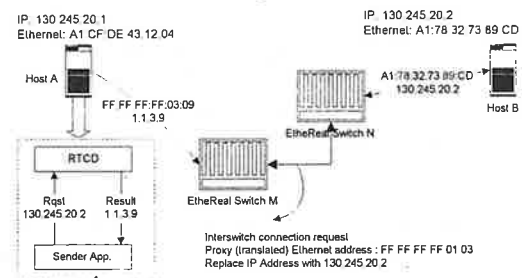


Figure 2: Connection Setup Procedure in the *EtheReal* Architecture. Host A initiates a connection to Host B through the Real Time Communication daemon. The RTCD at host A creates a Proxy IP address and a Proxy Ethernet address as a connection ID. Connection ID remapping occurs at each switch along the route.

data through a normal UDP connection. Thus, *EtheReal* can run on any platform that supports BSD style sockets.

2.3 REAL TIME LINK LAYER MULTICAST

Since *EtheReal* operates at the link layer, we chose a lightweight set of multicast primitives upon which more comprehensive multicast architectures may be built. The basic primitives supported by *EtheReal* are based upon the following assumptions we make about the common case in local area networks:

1. Since our target environment is switched *Ethernet* based LANs, we assume that the network is homogenous in terms of capacity and service offerings and that sufficient capacity exists to offer the same resource reservation to all receivers of the multicast group. Thus *EtheReal* multicast groups are homogenous with respect to receiver resource reservation.
2. The majority of multicast applications on local area networks are either single sender server side push applications or can be built upon a single sender architecture. Hence, *EtheReal* multicast groups are based on a single sender multiple receiver model.

- Receiver membership is dynamic. The multicast architecture should support dynamic receiver membership change without reestablishing the multicast group.

The basic multicast architecture of *EtheReal* supports a single sender homogenous receiver model. In the *EtheReal* multicast model, multiple senders can be supported by opening multiple single sender multicast groups to the same receiver pool. In a similar vein, receiver heterogeneity can be supported by creating multiple homeogenous multicast groups, each of which corresponds to a different resource reservation requested by a receiver or a group of receivers.

3 FAULT HANDLING

At layer 3, IP networks allow multiple routes to exist between a source and a destination host. The shortest path is then chosen by the OSPF algorithm from the list of available routes. Broadcast loops are prevented by using a time to live field in the packet header which is decremented at every IP hop. When the field reaches zero, the packet is discarded.

Unlike, the case for IP, transparent switched *Ethernet* networks are not allowed to maintain their own header with a hop count field; by definition their presence is transparent at the link layer network. Broadcast loops are avoided by superimposing a logical spanning tree upon the physical topology to enforce a single route between any pair of hosts. Links that do not belong to the spanning tree are used to create a new spanning tree in the case of failure of links in the current spanning tree.

Routing in a transparent switched *Ethernet* network relies upon the existence of a spanning tree connecting the switches in the network. In this context, fault detection involves (i) detection of failures on the links/switches belonging to the spanning tree and (ii) detection of broadcast loops caused by the addition of new switches to the network or changes to the existing physical topology. Fault recovery involves rebuilding the spanning tree to ensure connectivity to the entire network. Since packets are not routed during the spanning tree reconfiguration process to avoid introducing data loops in an unstable network, spanning tree reconfiguration constitutes the major of the time spent in fault recovery.

3.1 IEEE 802.1D DISTRIBUTED SPANNING TREE ALGORITHM

Current transparent switched *Ethernet* networks use the IEEE 802.1d distributed spanning tree discovery algorithm to detect loops and logically block redundant paths, so that only one route exists between any pair of end hosts. In this section we present a summary of the IEEE algorithm and its limitations in a a connection oriented real time transparent switched *Ethernet* framework. For a more detailed treatment of IEEE 802.1d refer to [7, 8].

In a switched *Ethernet* network, each switch has a unique identifier. The IEEE 802.1d algorithm uses this identifier to identify a switch in the spanning tree. When a spanning tree is first started on a network, each switch thinks that it is the root switch and transmits a message on all its ports which contains (a) Root switch id (in this case its own id), (b) Cost to root (initially 0) and (c) Transmitting switch id (in this case its own id). Each switch receives these messages on all its ports from other switches and saves the best message on each port. The best (lowest) message is determined by comparing the information in each message to the message the bridge has currently stored for that port. When a switch comes across a better message than it has stored at a particular port, it replaces the stored message with the new one. When two messages are compared, the root switch id is the most important determining factor. If the root id fields are equal, then the cost is compared. The last determining factor is the transmitting switch id.

From the messages received, each switch determines the root switch as the one with the lowest switch identifier. After the root is chosen, each switch determines its distance to the root and selects the port with the shortest path to the root as the root port. Now that the switch knows the contents of its own message, it compares it with the messages received on all its ports. If the switch's own message is better than all the received messages, it assumes that it is the designated switch for the attached LAN segment. Only the designated switch on a segment is allowed to forward data between the segment and the path to the root. The designated switch is the one with the lowest cost path to the root switch; in case of a tie the switch with the lower identifier wins. If a switch receives a better message on a port than it would transmit, it no longer transmits messages on that LAN. When the algorithm stabilizes, only the designated switch transmits messages on a LAN.

While the IEEE 802.1d protocol works well for best effort traffic on transparent switched *Ethernet* networks, it has several shortcomings in a real time connection oriented environment. First, the election to decide the root switch is based on the smallest switch identifier in the entire network. Since the size of the network is unknown, conservative global timers are used, at the expiry of which it is assumed that all the switches have converged to the same election result. The recommended timer values result in a spanning tree discovery phase of 30 seconds, during which the entire network is unusable. Secondly, the spanning tree can change after a reconfiguration. Because data can only be transmitted through links belonging to the new spanning tree, all preexisting connections using links in the old spanning tree have to be torn down. Thus, faults in any part of the network invalidate all the connections through the network. Coupled with a large spanning tree reconfiguration (fault recovery) time, network faults can result in large interruptions in service in IEEE 802.1d based networks.

3.2 OUR APPROACH: THE PROPAGATION ORDER SPANNING TREE

As stated earlier, a real time connection oriented network places three major constraints on the fault detection and recovery mechanisms used on it:

1. Fault detection has to be responsive in order to minimize the effect of a fault. Long fault detection times may void any resource guarantees offered by the network.
2. The fault recovery mechanism should constrain the effect of the fault to the affected link(s)/switch(es) and the connections through them, while allowing normal operation in the rest of the network.
3. The fault recovery mechanism has to be fast, so that the affected connections can be reestablished with minimal interruption in service.

In the context of a transparent switched *Ethernet* network, constraint 3 translates to a need for a fast spanning tree building algorithm. Constraint 2 states that fault recovery by spanning tree reconfiguration should localize the effect of the fault to connections directly passing through the fault, with the rest of the network delivering uninterrupted service to real time connections. In this section we present our approach, which is based upon the propagation order spanning tree algorithm used in the source routed AUTONET[1] network.

The propagation order spanning tree algorithm works in three phases:

1. A startup phase (SP), which goes from the root of the tree to the leaves.
2. A startup completion phase (SCP), which goes from the leaves of the tree to the root.
3. A topology discovery complete (TDC) phase which goes from the root of the tree to the leaves.

Let us assume that some switch in the network spontaneously initiates the startup phase and becomes the root of the spanning tree. It transmits a message on all its ports asking its neighbors to join the spanning tree. Each node accepts the first offer it receives to join the spanning tree and rejects subsequent offers from the same root. When a node accepts an offer to join the spanning tree, it asks its neighbors to join the spanning tree and starts a reply timer. It is assumed that all replies to a join message arrive within the reply timer interval. Note that each node either accepts or rejects the join offer immediately; hence the local reply timer is of minimal duration.

At the end of the reply timer interval, if a node has received rejects from all its neighbors, it assumes that it is a leaf node and initiates the startup completion phase by sending a SCP message to its parent. The parent waits for this message from all its children. When it receives all outstanding SCP messages, it transmits a SCP message to its parent and so on

till this message reaches the root of the tree. When the root detects that all its children have completed the startup phase, it signals the completion of SCP and initiates the topology discovery complete (TDC) phase by sending a TDC message to its children, which further transmit this message till it reaches the leaves. The TDC message is an indication to a node that it can safely begin forwarding data on its parent link and all links connected to its children. The set of all such forwarding links constitutes the spanning tree.

In the case of multiple switches claiming to be the root, their multiple spanning trees compete against one another during the SP of the different trees. When a node receives an offer to join a tree, it can be in one of three states: (a) It has already accepted an offer to join the tree in the message. (b) It hasn't received any join offer, or (c) It has already accepted an offer to join a different tree. The first and second cases are simple. In the third case, the node compares the root identifier of the new join message with its counterpart from the tree that it has joined. If the root identifier in the new join message is numerically smaller than that of the tree it has joined, it accepts the new join message, erases its previous state and broadcasts a join-message with the new tree, to all its neighbors. The algorithm stabilizes at the end of the SCP, and the root with the smallest identifier wins. An important difference here is that the smallest identifier contest is only among competing roots and not among all switches as in IEEE 802.1d.

This algorithm operates in a greedy manner with each root trying to get switches to join its spanning tree. The single major advantage over IEEE 802.1d is that the termination condition is detected when the winning root receives SCP messages from all its children. This approach does not involve conservative global timers and makes no assumptions about the size of the network. The spanning tree reconfiguration time measurements presented in section 4 show that this approach works with minimal interruption in service (in terms of time) due to spanning tree reconfiguration.

3.3 FAULT DETECTION

In the target switched *Ethernet* environment, our fault detection mechanism detects the following kinds of faults:

1. Failure of hosts connected to peripheral switches
2. Failure of links and switches
3. Changes in the network topology caused by reconnection of switches or addition of new switches

Our fault detection mechanism uses hello messages between neighbors to indicate that the source of the message is alive. At the periphery of the network, hosts send periodic hello messages to their adjacent switch. If consecutive hello messages fail to arrive, it is assumed that the host is down,

and all connections originating or terminating at that host are closed.

Switches in the network send periodic (100ms in the current implementation) inter switch hello messages on all their ports. Arrival of hello messages on previously unconnected ports or absence of hello messages on existing connected ports indicates the presence of a new switch, or failure of an existing one. Substitution of switches is detected by comparing the source switch identifier embedded in the inter switch hello with the identifier stored at the port on which the hello was received. The source identifier of the adjacent switch connected to each port is stored during the spanning tree discovery phase. Switches that detect any failure condition except host failure, initiate a fault recovery operation.

3.4 FAULT RECOVERY

When the fault detection mechanism detects a change in the network topology, it notifies the fault recovery mechanism and indicates the offending link. In the interior of the network, the fault would be independently detected by switches both upstream as well as downstream of the fault. The fault recovery mechanism on each switch that detects the fault, performs the following functions:

1. Discard all best effort traffic
2. Terminate all connections going through the offending link.
3. Rebuild the spanning tree
4. Reestablish the connections that were torn down

When a failure is detected, we discard all best effort traffic. The rationale here is that, when the route to the destination of a packet is unknown, forwarding of best effort traffic results in a broadcast. Since broadcast at the link layer requires a spanning tree to avoid loops, we avoid the problem by discarding best effort traffic.

The fault recovery mechanism then terminates all real time connections going through the offending link and informs the source of the connection that the network is unstable. The switches that detect the failure then initiate a spanning tree reconfiguration, with themselves as root. At the end of the spanning tree rebuild phase, all the hosts connected to the network are informed that the network is stable. Hosts that had their connections torn down due to network failure now initiate a new connection to their destination. In our implementation on *EtheReal*, reestablishment of torn down connections is done by the Real Time Communication Daemon, and is transparent to the host application, which merely sees a small interruption in service.

In the *EtheReal* connection oriented network architecture, end to end real time connections are composed of point to point unicast connections between switches along the connection

route, with connection remapping occurring at each switch. Once a real time connection is setup, each switch along the route knows the next hop of the connection. Real time connection oriented traffic is never broadcast in the network.

Recall that the fault detection mechanism detects switch failure, addition of a new switch as well as substitution of switches. Absence of a detected fault along a connection path indicates that there has been no physical change along the path. The fact that real time traffic never causes broadcasts, coupled with the absence of a detected fault leads to a key observation. *Once a real time connection is setup, it can follow the same route through the network without causing network loops, independent of spanning tree reconfiguration, as long as there is no change detected along its connection path.* We use this argument to forward real time connection oriented traffic over a link, independent of whether the link belongs to the current spanning tree, or is blocked. In a similar vein, in the absence of a detected fault, we can continue forwarding real time connection oriented data during the spanning tree rebuild process.

The main contribution of our fault recovery mechanism is that it localizes the effect of a fault and allows real time connection oriented traffic to flow through the physically unaffected parts of the network without interruption in service.

4 IMPLEMENTATION AND EVALUATION

Our fault detection and recovery mechanisms have been implemented on a 3 switch, 2 host EtherReal network. The switches are interconnected as a loop by a point to point topology. Both, the switches and the hosts are off-the-shelf 200 Mhz Intel Pentium Pro processor based PCs with 32 MB RAM, running the Linux Operating System. Each switch has four PCI based Intel EtherExpress 100 Mbps Fast Ethernet cards, implementing a four port switch with the PCI bus as the switching backplane. The links interconnecting the switches operate in full duplex mode and can transmit and receive 100 Mbps in both directions simultaneously. The RTCD and RTTR library run on the hosts connected to the network. Performance measurements were made with the 5 ns resolution clock tick counter (Model Specific Register) on the Pentium Pro CPU.

4.1 FAULT DETECTION

Since fault detection is implemented by periodic hello messages, absence of which indicates a failure; the hello timer interval determines the responsiveness of fault detection. The implementation uses two hello timers. The first timer is used to detect end host failure and operates at the periphery of the network, between the end hosts and their adjacent switches. The only recovery possible in end host failure is releasing any resources reserved for connections to/from the failed host. Hence, the hello timer uses a coarse timer resolution to minimize hello message overhead and timer errors caused by

high CPU utilization at the end hosts. Our prototype uses a timer interval of 30 seconds and a hello message size of 64 bytes. Absence of two consecutive hello messages from the end host indicates a failure.

The second hello timer is used to detect switch/link failures and operates in the interior of the network. Because switch failures can partition the network and void real time guarantees, failures have to be detected quickly. Our prototype uses an interswitch hello interval of 100 ms, with a message size of 64 bytes. Absence of 2 consecutive hello messages initiates the fault recovery mechanism. Since our prototype switch is implemented in software on Linux, which uses a hardware timer resolution of 10ms, timer interrupt errors can result in false reconfigurations. To avoid this, the hello timeout interval is adds a safety margin of 20 ms, leading to a fault detection time of 220ms. Recall that hello messages travel between adjacent switches. The hello timer interval is independent of the size of the network. Switches resynchronize their hello timeout interval on each hello message received to prevent the accumulation of clock skew errors. The relatively large fault detection time is an artifact of our hardware independent implementation. We are looking into hardware specific solutions to detect link failures using existing *Ethernet* fault detection hardware. This could significantly reduce the cost of fault detection, to the extent of solely relying on hardware carrier failure rather than the current periodic software hello message.

4.2 FAULT RECOVERY

To analyze the scalability of our approach, we instrumented our prototype to measure the overhead of the various components of a fault recovery operation. From the perspective of the affected connections, the fault recovery mechanism spends time in:

1. Terminating connections going through the faulty link/switch.
2. Reconfiguring the spanning tree
3. Re-establishing the connections that were torn down.

To terminate the connections through a failed link we need to (a) identify the connection identifiers of the failed connections (b) send a teardown message to the upstream/downstream sources of outgoing connections and (c) tear the connection to the upstream/downstream destinations if incoming connections. The CPU overhead of processing the connection queue is 95 ns/connection, which includes time spent in freeing up resources used by the connection. The cost of sending a teardown message to the next upstream hop involves formulating the message and setting up a DMA transaction to transfer the message, which accounts for a total of 2us/message/hop.

As mentioned in section 3.2, reconfiguration of the spanning tree occurs in 3 phases, the top down startup phase (SP), the

bottom up startup completion phase (SCP) and the top down topology discovery complete phase (TDC), each of which has a message processing time component and a message transmission time component. The end of the SP is reached when some nodes detect that they are leaves and send a SCP message to their parent. Each node waits for the reply timer interval mentioned in section 3.2 and assumes that it is a leaf if it receives rejects for all its join messages. The reply timer interval is the main time component of SP, since leaf detection and subsequent SCP is performed after its expiry. A suitable value for this timer interval is be obtained by observing that the processing delay in *EtheReal* is bounded by its real time scheduler cycle time of 10 ms. In the worst case the processing of the join message and the subsequent reply processing at the source can be maximally delayed, which yields a worst case delay of 20 ms. Accounting for transmission delays, we have conservatively chosen a reply timer interval of 30 ms. Since the reply timer runs between a switch and its neighbor, its interval is independent of the size of the network. The SCP and TDC phases do not use any timers. Hence, their main cost is in message processing and transmitting phase success/failure messages upstream.

The total message processing cost depends on the number of competing spanning trees and the number of switches in the network, which dictates the number of messages to be processed. In the simple case of a new switch being connected to a single switch in the current network, the measured message processing time per switch in SP, SCP and TDC were 35ns, 20ns and 15 ns respectively. Note however, that the message processing time increases linearly with the number of ports in the switch. As before, the cost of composing and sending a message takes 2us.

Fault Recovery Component		Time Spent
Connection Termination	Connection Queue Processing	95 ns/connection
	Message transmission	2us/message
Spanning Tree Reconfiguration	SP Reply Timer	30ms
	Message Processing (SP, SCP, TDC)	35ns, 20ns, 15ns per message
	Message transmission	2us
Connection Reestablishment	Total Cost	8-14ms (on the host)
		0.1-0.6 ms/link

Table 1: Timing breakdown of the various components of fault recovery.

When each node receives the TDC message, it broadcasts a message to the hosts connected to it, indicating that the network is stable. The RTCD on the hosts then attempts to reestablish the connection. Connection reestablishment takes 8-14ms at the end host (depending on the CPU load on the machine), 0.6ms at the first switch (since it has to route the packet) and 0.1ms at each switch thereafter. Only when the interrupted connection is re-established, can fault recovery be considered complete. Table 1 summarizes the breakdown of the various components of fault recovery.

To put the above timing breakdown in perspective, we mention the fault recovery time for two scenarios, with no real time connections flowing through the network. In the first scenario, a new switch was added to the network by connecting it to a single switch in the current network. The measured fault recovery time was 30.2ms from start of SP to arrival of the TDC message at the new switch. In the second case, three switches were connected in a loop topology and one link was broken after the initial spanning tree was setup. This demonstrates the case with competing spanning trees. The measured fault recovery time was 30.6ms. In both cases, the reply timer interval of 30ms dominates recovery time. As a final note, fault tolerance involves both fault detection and fault recovery and is dominated by the fault detection time of 220ms, which corresponds to a minor jitter in audio/video communication. In contrast, the IEEE 802.1d spanning tree algorithm has a detection time of 1sec and a recommended recovery time of 30sec.

5 RELATED WORK

There has been relatively little published work on the issue of automatic fault detection and recovery in the context of local area networks and none with reference to transparent real time switched *Ethernet* networks. [1] describes the automatic reconfiguration mechanisms in AUTONET, a proprietary switch based local area network with hosts and switches interconnected 12 by 12 crossbar switches using 100 Mbps full duplex point to point links. AUTONET is a packet switching network where packets are forwarded using a precomputed link by link path. AUTONET uses a form of the propagation order spanning tree to (a) generate a loop free topology and distribute it to all the switches and (b) reconfigure the network in the case of detected failures. The network topology is then used to precompute a link by link route from the source to the destination and encode it in the network assigned destination address as a part of the packet header. AUTONET implements fault detection in hardware.

[2] describes the fault tolerance mechanisms built into the PARIS[3] gigabit experimental network, as a part of the AURORA testbed. PARIS maintains the entire topology of the network by having every switch send out periodic status updates of the links connected to them. PARIS uses the topology description to create source routed connections. In the case of failures, PARIS localizes the effect of its software managed spanning tree changes to the affected link. Connections using other parts of the network remain unaffected by the failure.

[4] describes the fault tolerance features built into RETHER, which uses a software based token passing protocol to arbitrate channel access and deliver bandwidth guarantees over shared segment *Ethernet* LANs. Since RETHER runs in a shared segment *Ethernet* network, fault tolerance translates to mechanisms for token regeneration/recovery in the case of packet corruption or host failure.

[5] reports the results from a simulation study of several local as well as global fault management schemes. The simulations address issues involved in performing local rerouting and fault recovery given the relative priority of the connections with respect to QoS bounds like delay. However, the focus of the study is effective rerouting of traffic in the presence of QoS bounds in an arbitrary real time network, with no particular emphasis on *Ethernet*. [6] reports a simulation study using dispersity routing to deliver fault tolerance. In this approach, a fault tolerant flow is set up along multiple disjoint routes. Some routes carry application packets, whereas others carry error correction packets. Failure along a fixed number of channels can be recovered by the error correction code. The focus here is on the ability to tolerate faults, rather than detect and recover from them.

In contrast to the above approaches, our fault detection and recovery mechanism is simple, built in software and runs over commodity *Ethernet* hardware. In keeping with the spirit of transparent *Ethernet* switches, the fault detection and recovery mechanisms protocol are transparent to end hosts in the network. Finally, the performance results on the implementation of our approach show that fault detection is responsive and fault recovery localizes the effect of a fault to affected link and the connections going through them without any service interruption to the rest of the network.

6 CONCLUSION

This paper describes the design and implementation of an automatic fault detection and recovery mechanism for real time switched *Ethernet* networks. The prototype implementation of the fault tolerance mechanism on our real time connection oriented switched *Ethernet* network, *EtheReal* demonstrates the feasibility of our software based approach. The performance measurements show that fault detection and recovery times are acceptable for real time networks. The main advantage of our approach is the ability to localize the effect of network faults to the affected links/switches and the connections through them without any service interruption to real time connections in the rest of the network. Connections affected by network faults are automatically reestablished with minimal interruption in service.

In the future, we plan on extending the work in two directions. First, we are studying hardware specific solutions to detect link failures using existing *Ethernet* fault detection hardware. This could significantly reduce the cost of fault detection. Secondly, rather than rebuild the entire spanning tree, we are looking into mechanisms for incrementally changing the spanning tree in the presence of single link failures.

REFERENCES

- [1]: "Automatic Reconfiguration in Autonet"; Thomas L. Rodeheffer and Michael D. Schroeder; Operating Systems Review, Vol 25. Number 5. pp. 183-197.
- [2]: "Distributed Control for PARIS"; Baruch Awerbuch, Israel Cidon, Inder Gopal, Marc Kaplan and Shay Kutten; Proc. 9th ACM Symposium on Principles of Distributed Computing, Aug. 1990
- [3]: "PARIS: An approach to integrated high-speed private networks"; Israel Cidon and Inder Gopal; International Journal of Digital and Analog Cabled Systems 1(2), pp.77-86, Apr. 1988
- [4]: "Fault Tolerance Mechanisms in The RETHER Protocol"; Tzi-cker Chiueh and Chitra Venkatramani; Pacific Rim International Symposium on Fault-Tolerant Systems, Dec. 1997
- [5]: "Recovering guaranteed performance service connections from single and multiple faults"; A. Banerjea, C.J. Parris and D. Ferrari; Proc. IEEE Globecom, pp.162-168, Nov. 1994
- [6]: "Simulation study of the capacity effects of dispersity routing for fault tolerant real time channels"; A. Banerjea; Proc. SIGCOMM Symposium pp. 194-205, Aug. 1996
- [7]: IEEE specification 802.1d MAC Bridges. D9, July 14, 1989
- [8]: "Interconnections , Bridges and Routers"; Radia Perlman; Addison-Wesley Publications
- [9]: "Fast restoration of ATM networks"; J. Anderson, B.T. Doshi, S. Dravida and P. Harshavardhana; IEEE JSAC, vol. 12, no. 1 pp. 128-138, Jan 1994
- [11]: "*EtheReal*: A Host-Transparent Real-Time Fast Ethernet Switch"; Srinidhi Varadarajan and Tzi-cker Chiueh; IEEE ICNP 1998
- [12]: "Fast Restoration of real-time communication service from component failures in multi-hop networks"; S. Han and K.G. Shin; Proc. SIGCOMM Symposium, Sep. 1997
- [13]: "Resource Aggregation for Fault Tolerance in Integrated Services Networks"; Constantinos Dovrolis and Parameshwaran Ramanathan; ACM Computer Communication Review, Apr. 1998
- [14]: "RSVP: A New Resource ReserVation Protocol"; Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker and Daniel Zappala; IEEE Network Magazine, Sep. 1993
- [15]: "Resource ReserVation Protocol (RSVP) - Version 1, Functional Specification"; R. Braden, L. Zhang, S Berson, S Herzog and S Jamin; RFC 2205, Sep. 1997