MTCP: Scalable TCP-like Congestion Control for Reliable Multicast

Injong Rhee[†] *Nallathambi Balaguru*[‡] *George N. Rouskas*[†]

[†]Department of Computer Science
North Carolina State University
Campus Box 7534
Raleigh, NC 27695-7534

[‡]Department of Mathematics
and Computer Science
Emory University
Atlanta, GA 30322

Abstract

We present MTCP, a congestion control scheme for large-scale reliable multicast. Congestion control for reliable multicast is important, because of its wide applications in multimedia and collaborative computing, yet nontrivial, because of the potentially large number of receivers involved. Many schemes have been proposed to handle the recovery of lost packets in a scalable manner, but there is little work on the design and implementation of congestion control schemes for reliable multicast. We propose new techniques that can effectively handle instances of congestion occurring simultaneously at various parts of a multicast tree.

Our protocol incorporates several novel features: (1) *hierarchical congestion status reports* that distribute the load of processing feedback from all receivers across the multicast group, (2) the *relative time delay* (RTD) concept which overcomes the difficulty of estimating round-trip times in tree-based multicast environments, (3) *window-based control* that prevents the sender from transmitting faster than packets leave the bottleneck link on the multicast path through which the sender's traffic flows, (4) a *retransmission window* that regulates the flow of repair packets to prevent local recovery from causing congestion, and (5) a *selective acknowledgment* scheme that prevents independent (i.e., non-congestion-related) packet loss from reducing the sender's transmission rate. We have implemented MTCP both on UDP in SunOS 5.6 and on the simulator `ns`, and we have conducted extensive Internet experiments and simulation to test the scalability and inter-fairness properties of the protocol. The encouraging results we have obtained support our confidence that TCP-like congestion control for large-scale reliable multicast is within our grasp.

Keywords: Reliable multicast, congestion control

1 Introduction

As the Internet becomes more diversified in its capabilities, it becomes feasible to develop and offer services and applications that were not possible under earlier generations of Internet technologies. The Multicast Backbone (MBONE) and IP-multicast are two Internet technologies that have enabled a wide range of new applications. Using multicast, large-scale conferencing involving hundreds to thousands of participants is possible over the Internet. As multicast technologies become more widely deployed, we expect to see new multicast-based applications that demand more bandwidth and higher speed. Many of these applications will require reliable data transfer.

Multicast traffic generated by these applications can be of two types: *quality-of-service guaranteed* and *best effort*. QoS guaranteed traffic requires the underlying network to provide per-flow resource reservation and admission control services. Unless these services become widely deployed over the Internet and made sufficiently inexpensive for general use, they will likely be available only to a small fraction of future Internet traffic, and multicast traffic will be primarily best-effort. This paper is concerned with the flow and congestion control of *best-effort* multicast traffic.

Congestion control is an integral part of any best-effort Internet data transport protocol. It is widely accepted that the end-to-end congestion control mechanisms employed in TCP [1] have been one of the key contributors to the success of the Internet. A conforming TCP flow is expected to respond to congestion indication (e.g., packet loss) by drastically reducing its transmission rate and by slowly increasing its rate during steady state. This congestion control mechanism encourages the fair sharing of a congested link among multiple competing TCP flows. A flow is said to be *TCP-compatible* or *TCP-like* if it behaves similar to a flow produced by TCP under congestion [2]. At steady state, a TCP-compatible flow uses no more bandwidth than a conforming TCP connection running under comparable conditions.

Unfortunately, most of the multicast schemes proposed so far do not employ end-to-end congestion control. Since TCP strongly relies on other network flows to use congestion control schemes similar to its own, TCP-incompatible multicast traffic can completely lock out competing TCP flows and monopolize the available bandwidth. Furthermore, multicast flows insensitive to existing congestion (especially congestion caused by their own traffic) are likely to cause simultaneous congestion collapses in many parts of the Internet [3]. Because of the potential far-reaching damage of TCP-incompatible multicast traffic, it is highly unlikely that transport protocols for large-scale reliable multicast will become widely accepted without TCP-like congestion control mechanisms.

The main challenge of congestion control for reliable multicast is scalability. To respond to congestion occurring at many parts of a multicast tree within a TCP time-scale, the sender needs to receive immediate feedback regarding the receiving status of all receivers. However, because of the potentially large number of receivers involved, the transmission of frequent updates from the receivers directly to the sender becomes prohibitively expensive and non-scalable.

Another challenge is the isolation of the effects of persistent congestion. As a single multicast tree may span many different parts of the Internet, TCP-like congestion control will reduce the sender's transmission rate upon indication of congestion in any part of the tree. While such a feature fosters fairness among different flows (*inter-fairness*), it does not address the issue of fairness among the receivers in the same multicast group (*intra-fairness*) [4]. Specifically, it would be unfair for non-congested receivers to be subject to a low transmission rate just because of some isolated instances of congestion.

In this paper, we introduce *Multicast TCP (MTCP)*, a new congestion control protocol for reliable multicast that addresses the inter-fairness and scalability issues. The issue of intra-fairness is outside the scope of this paper, and it will be addressed in future work. Our protocol is based on a multi-level logical tree where the root is the sender, and the other nodes in the tree are receivers. The sender multicasts data to all receivers, and the latter send acknowledgments to their parents in the tree. Internal tree nodes, hereafter referred to as *sender's agents* (SAs), are responsible for handling feedback generated by their children and for retransmitting lost packets. MTCP incorporates several novel features, including:

1. *hierarchical congestion status reports* that distribute the load of processing feedback from all receivers across the multicast group,
2. the *relative time delay* (RTD) concept which overcomes the difficulty of estimating round-trip times in tree-based multicast environments,
3. *window-based control* that prevents the sender from transmitting faster than packets leave the bottleneck link on the multicast path through which the sender's traffic flows,
4. a *retransmission window* that regulates the flow of repair packets to prevent local recovery from causing congestion, and
5. a *selective acknowledgment* scheme employed at SAs to prevent independent (i.e., non-congestion-related) packet loss from reducing the sender's transmission rate.

We have implemented MTCP both on UDP in SunOS 5.6 and on the simulator `ns`, and we have conducted extensive Internet experiments and simulation to test the scalability and inter-fairness properties of the protocol. The encouraging results from these experiments indicate that MTCP is an effective flow and congestion control protocol for reliable multicast.

Tree-based protocols are not new and have been studied by many researchers [5, 6, 7, 8, 9]. However, little work has been done on TCP-like congestion control for these protocols. Instead, most previous work has focused on the issues of error recovery and feedback implosion. In [5, 10] it has been analytically shown that tree-based protocols can achieve higher throughput than any other class of protocols, and that their hierarchical structure is the key to reducing the processing load at each member of the multicast group. However, the analysis does not consider the effects of congestion control. Tree-based protocols such as RMTP [6] and TMTP [8] do not incorporate end-to-end congestion control schemes an do not guarantee inter-fairness. In [9, 11] it was proposed to use a tree structure for feedback control, and a detailed description of how to construct such a tree was provided, but no details on congestion control were given. A more detailed discussion on related work can be found in Section 5.

The rest of the paper is organized as follows. In Section 2 we provide an overview of MTCP, and in Section 3 we present a detailed description of the protocol and its operation. In Section 4 we present results from Internet experiments and simulation. In Section 5 we discuss related work, and we conclude the paper in Section 6.

2 Overview of MTCP

MTCP was designed with two goals in mind: TCP-compatibility and scalability. Compatibility with TCP traffic is needed because TCP is the most commonly used transmission protocol in the Internet, and also because the utility of TCP depends on all other network flows being no more aggressive than TCP congestion control (i.e., multiplicative decrease on congestion occurrence, and linear increase at steady state). Non-TCP-compatible flows can easily lock out TCP traffic and monopolize the available bandwidth. Scalability is necessary because the target applications of reliable multicast may involve a very large number of receivers. Below we give an overview of MTCP, and in the next section, we provide a detailed description of the protocol.

Packet loss detection and recovery via selective acknowledgments. A sender multicasts data packets using IP-Multicast [12] to all receivers. SAs in the logical tree store packets received from the sender in their buffers, and set a timer, called *retransmission timer*, for each packet they buffer. The sender also sets a retransmission timer for each of the packets it transmits. Each receiver may send a positive acknowledgment (ACK) or a negative acknowledgment (NACK) to its parent in the tree. Received packets are reported in ACKs and missing packets are reported in NACKs. An SA (or the sender) discards a buffered packet when it receives an ACK from all of its children. On the other hand, an SA (or the sender) retransmits a packet via unicast (a) upon receiving a NACK reporting that the packet is missing, or (b) if it does not receive a positive acknowledgment (ACK) for the packet from all its children in the logical tree before the timer associated with the packet expires.

Hierarchical congestion reports. Each SA independently monitors the congestion level of its children using feedback received from them. When an SA sends an ACK to its parent, it includes in the ACK a summary of the congestion level of its children (called *congestion summary*). The parent then summarizes the congestion level of its own children, sends the summary to its parent, and so on. The sender regulates its rate based on its own summary. The congestion summary carries an estimate of the *minimum* bandwidth available along the multicast paths to the receivers contained in the subtree rooted at the SA that sends the summary. An SA computes its summary using the summaries it has received from its children and a TCP-like congestion window maintained using feedback from its children. As a result, the summary computed at the sender represents the current available bandwidth in the bottleneck link on the paths to all receivers in the multicast group. By sending only as much data as the bottleneck link can accommodate, the sender will not aggravate congestion anywhere in the network.

TCP-like congestion window. Each SA (including the sender) estimates the minimum bandwidth available in the multicast routes from the sender to its children by maintaining a TCP-like congestion window (cwnd). An SA maintains its cwnd using TCP-Vegas [13] congestion control mechanisms such as slow start and congestion avoidance. The only differences with TCP-Vegas are that (1) the congestion window is incremented when an SA receives ACKs for a packet from *all* of its children, and (2) receivers send NACKs for missing packets, and an SA immediately retransmits the packets reported missing. For instance, slow-start is implemented as follows. When an SA receives a new packet from the sender, it buffers the packet and sets the retransmission timer (RTO) of the packet to an estimated time before which the packet should be acknowledged. At the beginning of transmission or after a retransmission timer expiration, an SA sets its cwnd to one. The SA increments its cwnd by one when a packet is acknowledged by all of its children. Once the cwnd reaches a threshold called ssthresh, congestion avoidance is invoked and the cwnd is incremented by $1/\texttt{cwnd}$ each time a new packet is acknowledged by all of its children.

Congestion summary. The congestion summary sent by an SA whose children are leaf nodes, consists of two pieces of information: the size of its congestion window (cwnd), and the estimated number of bytes in transit from the sender to the SA's children (twnd). twnd is initially set to zero, it is incremented when a new packet is received from the sender, and it is decremented when a packet is acknowledged by all of the SA's children. The congestion summary of the other SAs consists of (1) the minimum of their cwnds and the cwnds reported by their children (minCwnd), and (2) the maximum of their twnds and the twnds reported by their children (maxTwnd). maxTwnd estimates the number of unacknowledged bytes in transit to the receivers in the tree and minCwnd estimates the congestion level of the bottleneck link on the multicast routes to the receivers in the tree. The sender always transmits data in an amount less than the difference between the values of maxTwnd and minCwnd that it computes. This window mechanism prevents the sender from transmitting faster than packets leave the bottleneck link.

Relative time delay (RTD). Unlike TCP where the sender maintains the congestion window based on feedback received from one receiver, MTCP does not provide closed-loop feedback. This is because SAs have to adjust their windows based on the ACKs for packets that another node (the sender) transmitted. The main problem with this open-loop system is that an SA cannot accurately estimate the round trip time (RTT) of a packet. This problem arises due to the unpredictable delay variance in the network and the fact that the sender's and SA's clocks are not synchronized. In MTCP, we measure the difference between the clock value taken at the sender when a packet is sent, and the clock value taken at the SA when the corresponding ACK is received from a child node. We call this time difference the *relative time delay* (RTD). The RTD to a child receiver can be easily measured by having each ACK carry the transmission time of the packet being acknowledged. Thus, RTD measurements can be taken every time the SA receives an acknowledgment. A weighted average of RTDs is used to estimate the retransmission timeout value (RTO) of packets. An SA sets the retransmission timer of a packet to expire only after the sum of the send time of the packet and the RTO of the SA becomes less than the current clock value of the SA. The use of RTD for this purpose is appropriate because the protocol uses only the relative differences in RTDs.

Retransmission window for fast retransmission. Retransmission may also cause congestion if many packets are lost in a loss burst and an SA retransmits them without knowing the available bandwidth between itself and its children. Recall that the congestion window at the SA only estimates the amount of data that can be sent from the sender to the SA's children. Because new and repair packets may travel through different routes, the congestion window cannot be used to regulate repair traffic. In MTCP, each SA maintains another window, called the *retransmission window*, used only for repair packets. The retransmission window is updated in the same way as `cwnd` (i.e., slow start, congestion avoidance, etc.). Since SAs receive ACKs for the packets they retransmitted anyway, maintaining the retransmission window does not incur significant overhead.

It is also desirable that repair packets be given higher priority than new packets. However, in best-effort networks, enforcing different priorities to traffic streams originating from different sources is not trivial. In the current version of MTCP, both new and repair traffic are given the same priority. We are investigating various techniques to assign higher priority to repair traffic that will be incorporated in future versions of the protocol. Allowing a larger initial size for the retransmission window during slow-start is one such technique.

Handling of Independent Loss. MTCP decreases the sender's transmission rate if any link of the multicast routing tree is congested. This may raise a concern that the protocol is too sensitive to independent packet loss: since for large multicast groups, almost every transmitted packet may experience independent loss, it might be argued that the overall throughput will be reduced to zero. However, in MTCP, most occurrences of independent loss trigger NACKs to SAs which immediately retransmit the lost packets. Only packet loss accompanied by indication of congestion, such as retransmission timeouts or several consecutive duplicate NACKs, reduces the congestion window. Simulation results to be presented later confirm that independent packet loss is immediately recovered by SAs and does not have a negative effect on the overall throughput.

3 Detailed Description of MTCP

3.1 Selective Acknowledgment Scheme

In MTCP, we use a selective acknowledgment (SACK) scheme in which each feedback contains information about all the received packets. We also adopt a delayed acknowledgment scheme in which each acknowledgment is delayed for a few tens of milliseconds before its transmission. Since an SA can quickly detect the packets lost by a receiver and retransmit them, these schemes reduce the number of acknowledgments and retransmissions. Also, our SACK scheme provides a good means to recover from independent, uncorrelated losses. In MTCP, each acknowledgment contains four fields:

- The start sequence number (`startseq`) indicating that a receiver has received all packets up to and including the one numbered `startseq` - 1.
- The end sequence number (`endseq`) of the packets it acknowledges.
- A fixed size `bitvector` which indicates the received and the missing packets at the receiver that sends the acknowledgment. `MAXBVSIZE` is the maximum size of the bitvector, so that (`endseq` -

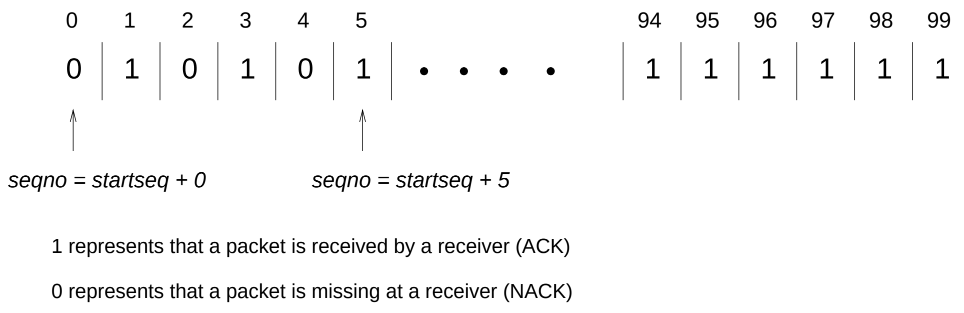
0 1 2 3 4 5 94 95 96 97 98 99
0 1 0 1 0 1 • • • • 1 1 1 1 1 1
seqno = startseq + 0
seqno = startseq + 5
1 represents that a packet is received by a receiver (ACK)
0 represents that a packet is missing at a receiver (NACK)

Figure 1: A sample bitvector representation in a SACK

`startseq`)$\leq$ `MAXBVSIZE`. The i-th bit in the bitvector indicates the reception status of the packet with sequence number `startseq`$+i$, $0 \leq i \leq$ (`endseq` - `startseq`). If this bit is set, then it is a positive acknowledgment (ACK). Otherwise it is a negative acknowledgment (NACK). The number of packets that a receiver acknowledges is given by `endseq` - `startseq` + 1.

- A type field which is either `PACK` or `PNACK`. If it is `PACK`, then the value of `endseq` indicates that the receiver has received up to and including the packet numbered `endseq` - 1. The bitvector is not used when the type field is `PACK`. If the type field is `PNACK`, the bitvector is used to find out the sequence numbers of the received and the missing packets.

Figure 1 shows a sample bitvector. Suppose that an acknowledgment has `startseq` equal to 150, `endseq` equal to 155, and the bitvector in Figure 1. It acknowledges 6 packets starting from `startseq` to `endseq`. From the bitvector we find that the receiver has received packets 151, 153, and 155 but has not yet received packets 150, 152, and 154. This acknowledgment also indicates that the receiver has received all packets up to and including packet 149. A similar bit vector is used in RMTP [6].

When an SA (or the sender) receives an indication that a packet is lost, it immediately unicasts the missing packet to the receiver that sent the NACK, unless the same packet was (re)transmitted to this receiver within a time period equal to the current estimated round trip time between the SA and the receiver (which is one of the SA's children). Since the SACK scheme indicates exactly which packets have been received, the SA (or the sender) can determine the packets that were lost by its children and retransmit only them, avoiding unnecessary retransmissions. Each acknowledgment also contains the send time of the packet with sequence number `startseq` $-$ 1, the congestion summaries, and the number of buffers available at the receiver. This information is used for flow and congestion control as described in the following sections.

3.2 Relative Time Delay (RTD) Measurement

In MTCP, an SA sets a retransmission timer for each packet it receives. The timer for a packet must be set to the mean time period between the time the packet was transmitted by the sender and the time that the SA expects to receive an ACK from all of its children. However, this time period is hard to measure because of the clock differences between the SA and the sender. To overcome this difficulty, we introduce the concept of *relative time delay (RTD)*, defined as the difference between the clock value, *taken at the SA*, when the

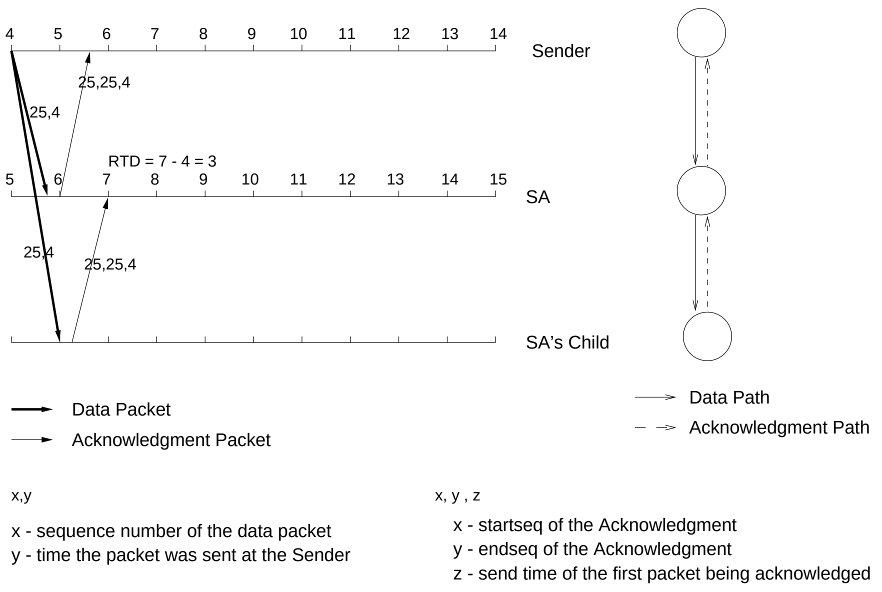
4 5 6 7 8 9 10 11 12 13 14
Sender
25,25,4
25,4
RTD = 7 - 4 = 3
5 6 7 8 9 10 11 12 13 14 15
SA
25,4
25,25,4
SA's Child
Data Packet
Acknowledgment Packet
Data Path
Acknowledgment Path
x,y
x - sequence number of the data packet
y - time the packet was sent at the Sender
x, y , z
x - startseq of the Acknowledgment
y - endseq of the Acknowledgment
z - send time of the first packet being acknowledged

Figure 2: Example of RTD measurement

ACK for a packet is received from a child node, and the clock value, *taken at the sender*, when the same packet was transmitted. MTCP requires that each data packet carry its transmission time at the sender, and that each ACK carry the transmission time of the packet with sequence number $\texttt{startseq} - 1$ (as explained above), making it easy for SAs to compute the corresponding RTDs.

The RTD is used in the same way that the round trip time is used in TCP-Vegas. For instance, the difference between the minimum measured RTD and the currently measured RTD to a child node is used to estimate the number of packets in transit (i.e., the actual throughput) from the sender to the child. Also, a weighted average of RTDs and their deviations in time are used to estimate the retransmission timeout value RTO_{RTD}. Using the RTD for these purposes is appropriate because MTCP uses only the relative differences in the RTDs. Given a value for RTO_{RTD}, MTCP sets the retransmission timer of a packet to expire only after the sum of the send time of the packet (according to the sender's clock) plus the RTO_{RTD} of the SA becomes less than the current clock value of the SA.

Figure 2 illustrates how the RTD is measured. In this example, the clocks of the sender and the SA are not synchronized, but the SA's clock is ahead of the sender's by one time unit. The sender transmits a packet at time 4 which is received by both the SA and its child, which then send an acknowledgment to their respective parents in the logical tree. Recall that the data packet includes its send time, and that the receivers copy this send time in their acknowledgments. Then, the RTD measured at the SA is 3 in this example, since the packet was transmitted by the sender at time 4, and it was received by the SA at time 7.

3.3 Round Trip Time (RTT) Measurement

An SA will retransmit a packet whose timer has expired, at which time it needs to reset the retransmission timer. Recall, however, that retransmissions are unicast from an SA to its children and may follow a different path than the original multicast transmission from the sender. As a result, while the retransmission timeout RTO_{RTD} for the *first* retransmission of a packet should be based on the RTD, for subsequent retransmissions an SA must adjust its timer to the round-trip time (RTT) between itself and its children.

In order to estimate the RTT, an SA periodically polls its children by sending a probe packet. In the current implementation of MTCP, the polling period is set to 2 seconds. Upon receiving this probe, a child immediately sends an acknowledgment to its parent. The SA can then measure the difference between the time when it sent a probe and the time it received the corresponding acknowledgment. These RTT measurements are used to set the retransmission timeout RTO_{RTT} for packets retransmitted at least once.

3.4 Estimation of RTD- and RTT-Based Retransmission Timeouts

The timeout and retransmission mechanisms employed in MTCP require the estimation of retransmission timeout values RTO_{RTD} and RTO_{RTT}. As these values can change over time because of routing and network traffic changes, MTCP needs to track these changes and modify its timeout values accordingly. We use the techniques employed by TCP [1] to set the RTO_{RTD} and RTO_{RTT} based on the average measured length of RTDs and RTTs, respectively, and their deviations in time. Specifically, an SA performs the following computations to adjust the timeout periods for a child r:

$$\begin{aligned} Err_r &= M_r - A_r \\ A_r &= A_r + g_A \cdot Err_r \\ D_r &= D_r + g_D \cdot (|Err_r| - D_r) \\ rto_r &= A_r + 4D_r \end{aligned}$$

where M_r is the measured RTD or RTT for child r, A_r is the averaged RTD or RTT for r, g_A and g_D are gain terms between 0 and 1, D_r is the deviation in RTD or RTT for r, and rto_r is the estimated RTO_{RTD} or RTO_{RTT} for r.

In MTCP, each SA (including the sender) sets its RTO_{RTD} (respectively, RTO_{RTT}) to the maximum of the computed RTO_{RTD}s (respectively, RTO_{RTT}s) for all of its children.

3.5 Use of RTOs and Exponential Retransmission Timer Backoff

When a packet is initially transmitted by the sender, or when it is received for the first time by an SA, it is buffered by the sender or SA and the retransmission timeout of the packet is set to RTO_{RTD}, which represents the estimated amount of time for the packet to travel through the multicast tree to the children of the SA and for the corresponding ACKs to arrive at the SA. As we explained in Section 3.2, the retransmission timer for the packet is set to expire when the sum of the send time of the packet plus the RTO_{RTD} becomes less than the current time. When a packet is retransmitted because of a NACK or the expiration of its retransmission timer, the latter is set to RTO_{RTT}, since retransmissions are performed by an SA via unicast. Following this retransmission, whenever the timer expires, the timeout value is multiplied by two (exponential backoff) until an acknowledgment from all children is received.

Figures 3 and 4 illustrate how the timeouts are used in a part of a multicast tree involving three nodes: the sender, an SA who is a child of the sender in the logical tree, and a child of the SA. Figure 3 shows how the RTD-based timeouts are computed at the SA. In this scenario, the sender's clock is assumed to be ahead of the SA's clock by three time units. The sender transmits packet 25 at time 5; the send time is included in the packet header. The SA and its child transmit their acknowledgments, which also include the send time of the packet, to their respective parents. The SA receives the ACK at time 4, and calculates the RTD as -1 (=4-5). Assume that the estimated RTO_{RTD} is also -1. The sender then transmits packet 26 at time 7.2. The SA receives the packet and buffers it for possible retransmission. It also sets its retransmission timer to the send time of the packet plus the RTO_{RTD}, i.e., to 7.2+(-1)=6.2. The packet is lost on the route to the SA's child. Since the SA has not received an ACK for the packet when the timer expires at time 6.2, it retransmits packet 26 to its child. At that time, the SA also sets its retransmission timer to RTO_{RTT} (not shown in the figure).

In Figure 4, the RTO_{RTD} and RTO_{RTT} values are assumed to be 7 and 1, respectively, and the SA's clock is five units of time ahead of the sender's clock. The sender sends a packet at time 0, which reaches the SA at time 5.5. When the SA buffers the packet, it sets the retransmission timer of the packet to expire at time 7, based on the sum of the send time and the RTO_{RTD}. Since the SA does not get an acknowledgment within that time period, it retransmits the packet at time 7. At that time, the SA also sets the retransmission timer of the packet to expire at time 8, based on the RTO_{RTT} value of 1. Since it does not receive an ACK within the timeout period, it retransmits the packet again at time 8 and doubles the timeout value to 2. This doubling of the timeout value continues until an ACK for the packet is received.

3.6 Slow Start

Slow start is an algorithm used by TCP [1] to find the available bandwidth on the path of a connection. Slow start is invoked at the beginning of transmission or after the retransmission timer of a packet expires. As in TCP, MTCP also employs a slightly modified version of slow start to estimate the available bandwidth in the multicast paths. Specifically, the sender and SAs maintain a TCP-like congestion window whose size is indicated by `cwnd`. The algorithm starts by initializing `cwnd` to one *segment size*. When an SA (or the sender) receives an ACK for a segment *from all of its children*, it increments `cwnd` by one segment size. This increases `cwnd` exponentially until it reaches a threshold called `slow start threshold` (*ssThresh*). *ssThresh* is initially set to 64 Kbytes. When `cwnd` is greater than `ssThresh`, the SA increases its `cwnd` by 1/(segment size), effectively increasing the `cwnd` at a linear rate. This linear increase period is called the *congestion avoidance* period.

Since during slow start `cwnd` increases exponentially, this process itself can cause congestion. MTCP adopts the slow start mechanisms of TCP-Vegas [13] to detect congestion even during slow start. For this purpose, the sender and the SAs maintain two additional variables:

- `baseRTD` which is the minimum of the RTDs seen so far, and
- `mRTD` which is the currently measured RTD.

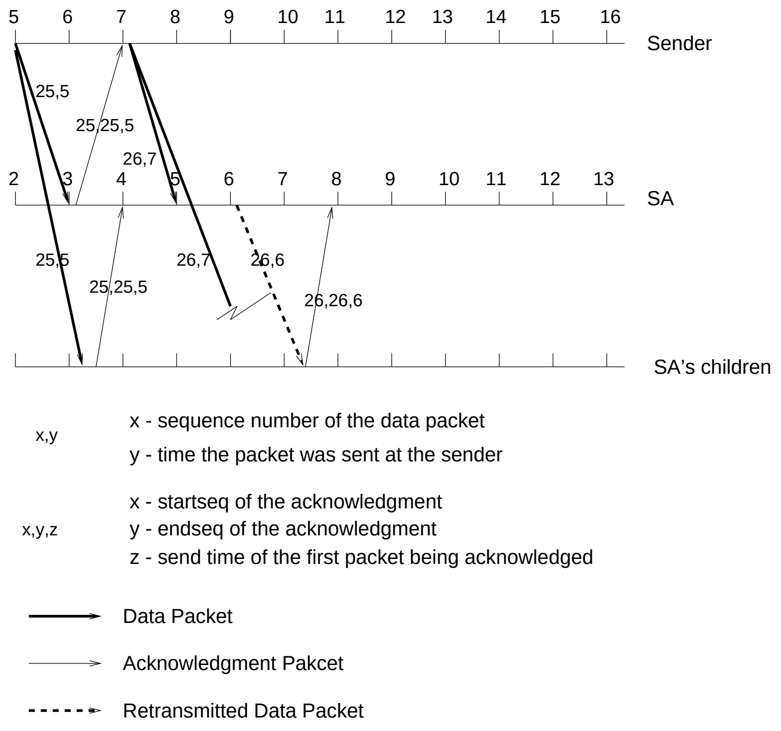
5
6
7
8
9
10
11
12
13
14
15
16
Sender
25,5
25,25,5
26,7
2
3
4
5
6
7
8
9
10
11
12
13
SA
25,5
25,25,5
26,7
26,6
26,26,6
SA's children
x,y
x - sequence number of the data packet
y - time the packet was sent at the sender
x,y,z
x - startseq of the acknowledgment
y - endseq of the acknowledgment
z - send time of the first packet being acknowledged
Data Packet
Acknowledgment Pakcet
Retransmitted Data Packet

Figure 3: Example of timeouts based on RTD

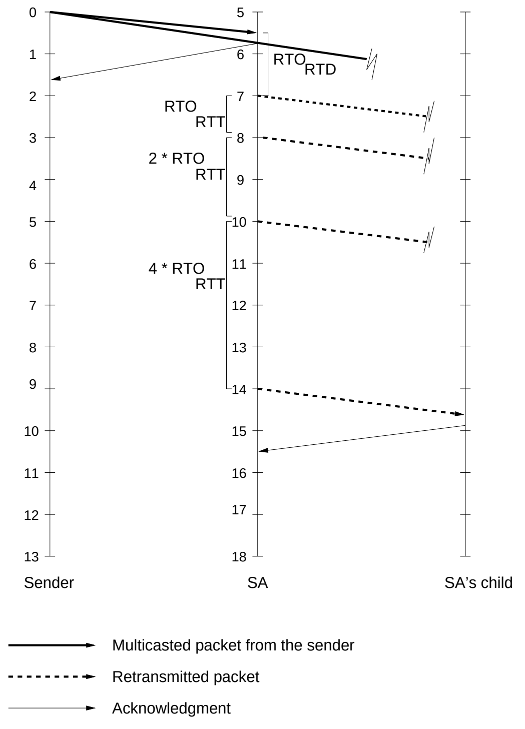

Figure 4: Example of timeouts and exponential retransmission timer backoff

If the difference between `mRTD` and `baseRTD` is less than a threshold γ, `cwnd` is increased exponentially. If the difference is greater than γ, the algorithm enters the congestion avoidance period.

3.7 Congestion Avoidance

During congestion avoidance, `cwnd` increases or decreases linearly to avoid congestion. When an SA (or the sender) receives ACKs for a packet from every child, it compares the `mRTD` and `baseRTD` of the slowest child. Let $\texttt{Diff} = \texttt{mRTD} - \texttt{baseRTD}$. As in TCP-Vegas, we define two thresholds α and β, $\alpha < \beta$, corresponding to having too little or too much, respectively, extra data in the network. When $\texttt{Diff} < \alpha$, we increase `cwnd` by 1/(segment size) and when $\texttt{Diff} > \beta$, we decrease `cwnd` by `cwnd`/8. `cwnd` remains unchanged if $\alpha < \texttt{Diff} < \beta$.

When a timeout occurs, `ssThresh` is set to one half of `cwnd`, `cwnd` is set to one segment size, and slow start is triggered. If, on the other hand, three consecutive NACKs are received, `ssThresh` is set to one half of `cwnd`, `cwnd` remains the same and the algorithm remains in congestion avoidance.

3.8 Retransmission Window

The packets retransmitted by an SA to one or more of its children may take a different route than the multicast path these packets followed when originally transmitted by the sender. Consider the situation arising when an SA receives an ACK reporting a list of lost packets. If the SA is allowed to retransmit a large number of packets regardless of the available bandwidth between itself and its children (recall that `cwnd` estimates only the bandwidth between the sender and the SA's children), it may cause another instance of congestion. To overcome this problem, each SA maintains another window, called the *retransmission window* for each child, which is used only for retransmitted packets. Maintaining the retransmission window is possible because SAs receive ACKs for the packets they send (i.e., a closed-loop system). In the current implementation, the initial window size of the retransmission window is set to one. The size of the window changes in the same way that TCP-Vegas modifies `cwnd` (i.e., slow start, congestion avoidance, and fast recovery). However, the size of the retransmission window is not reset to 1 even if retransmission is quiescent. Intermittent retransmission can be accommodated immediately with the window set for the previous retransmission session (or burst). Unless the previous retransmission session undergoes slow start, the window will be larger than 1.

3.9 Window-Based Flow Control

Flow control ensures that the sender does not send more data than the current capacity of a receiver. The current capacity of a receiver is indicated by the number of buffers available. In MTCP, each receiver advertises the number of buffers available to its parent. We define the *advertised window* of a node to be the minimum of the buffers reported by the children of the node. The sender always sends no more than its own advertised window.

Another important issue which affects the congestion control mechanism is how fast the sender transmits packets. In MTCP, the sender uses ACKs as a "clock" to strobe new packets in the network. Each SA (and the sender) maintains a variable called *transit window*, `twnd`. `twnd` is initially set to zero, it is incremented when a new packet is received from the sender, and it is decremented when a packet is acknowledged by all of the SA's children. Whenever a retransmission timeout occurs, `twnd` is set to zero. The information about `twnd` is propagated up the tree to the sender and it is used to regulate the transmission rate of the sender, as explained below.

The congestion summary an SA sends to its parent consists of two parameters:

1. parameter `minCwnd`, which is the minimum of the SA's `cwnd` and the `cwnd`s reported by its children, and which estimates the congestion level of the bottleneck link on the multicast routes to the receivers in the tree, and

2. parameter `maxTwnd`, which is the maximum of the SA's `twnd` and the `twnd`s reported by its children, and which estimates the number of unacknowledged bytes in transit to the receivers in the tree

The difference between `maxTwnd` and `minCwnd` is called the *current window*. The sender always transmits data in an amount no more than the current window. This window mechanism prevents the sender from transmitting faster than packets leave the bottleneck link.

3.10 Hierarchical Status Reports

If the sender is allowed to determine the amount of data to be transmitted based only on its own `cwnd` and `twnd`, which it maintains using feedback from its immediate children alone, it is highly likely that the sender will cause congestion somewhere in the multicast routes. This possibility arises from the fact that the sender's `cwnd` and `twnd` provide information only about the multicast paths to the sender's immediate children in the logical tree; the sender receives no first-hand information about the congestion status of the multicast paths to other nodes. To ensure that an MTCP session will not cause congestion anywhere in the multicast routes, we require that the sender regulate its rate based on the congestion status of all receivers in the tree. This is accomplished by using a hierarchical reporting scheme, in which information about the status of each receiver propagates along the paths of the logical tree from the leaves to the root (the sender) in the form of the congestion summaries discussed in the previous subsection.

Figure 5 illustrates how congestion summaries propagate from leaf receivers to the sender along the edges of the logical tree. In the figure, `cwnd` and `twnd` are expressed in units of number of segments (they are normally defined in bytes). SAs whose children are leaf nodes, send their `cwnd`s and `twnd`s to their parents. Referring to Figure 5, node 7 sends its `cwnd`(30) and `twnd`(8) to its parent (node 3), node 9 sends its `cwnd`(15) and `twnd`(10) to its parent (node 3), and node 1 sends its `cwnd`(25) and `twnd`(5) to its parent (node 0). Upon receiving this information, the parent SAs send to their own parents the minimum of the received `cwnd`s and their own `cwnd`, and the maximum of the received `twnd`s and their own `twnd`. In Figure 5, for instance, node 3 sends to node 0 the minimum of the `cwnd`s(15) and the maximum of the `twnd`s(10). The sender at node 0 computes the minimum of all `cwnd`s in the tree to be 15 and the maximum of all `twnd`s to be 10. Therefore, the current window is 15-10=5 segments. The sender sends no more than the minimum of the current window and the advertised window.

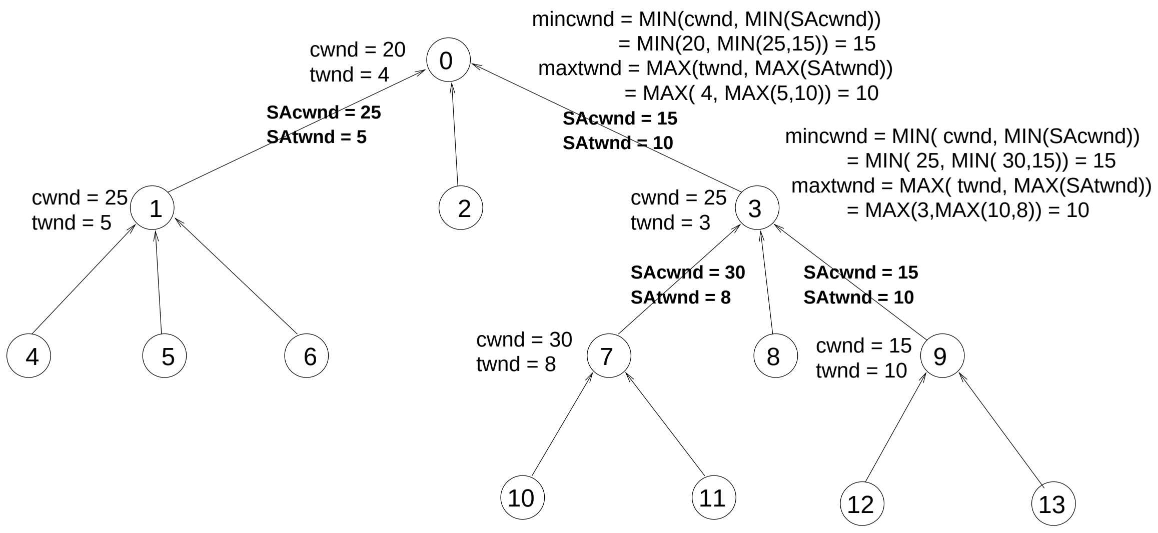

Figure 5: Example of hierarchical status reports

Note that the minimum bandwidth within the subtree rooted at a given SA is computed based only on information reported in the congestion summaries sent to this SA by its children. Since the maximum number of the children of a node is limited to a small constant, this scheme achieves good load-balancing.

We also note that the delay between the time when an SA detects congestion and the time when the sender reduces its transmission rate in response to this congestion, may be longer than the TCP time-scale. Since the congestion status report (in the form of a summary) has to travel all the way to the root of the tree, this delay can be larger than a round trip delay. However, unless congestion is reported directly to the sender (an approach that inevitably leads to ACK implosion), this extra delay is unavoidable. Furthermore, as it has been pointed out [14], extra delays up to a few seconds can be tolerated because network links where a single flow can create severe transient congestion are likely to employ an appropriate queue management mechanism such as random early detection (RED) [15, 2]. We have observed through Internet experiments and simulation that the delay in MTCP is well within this range; more details are given in Section 4 (also refer to Figure 8).

3.11 Window Update Acknowledgments

In MTCP, congestion summaries are normally piggybacked on every ACK and NACK. Thus, congestion summaries are reported by an SA whenever a new packet is received (recall also that ACKs/NACKs are delayed for a few tens of milliseconds for an SA to receive ACKs from its children). However, if congestion summaries are reported only upon reception of data packets, deadlocks are possible since the window size at an SA may change even if the sender does not transmit any packets. Consider the following scenario. The sender transmits a number of packets and receives ACKs for the packets from all of its children. One of the SAs, say, SA A, on the other hand, does not receive ACKs for these packets from its children. Thus, SA A will report a high value for the `twnd` in the congestion summary it sends to its parent, and which will propagate to the sender. It is possible that this high `twnd` value will reduce the size of the current window of the sender to zero, in which case the sender will not transmit any more packets. Since the SA will not receive any packets from the sender, it will not send any ACKs either. When the SA finally receives ACKs from its children, its `twnd` decreases from the previously high value. Since no ACKs on which to piggyback this window update are generated, the sender will never learn of this updated window, and in turn, it will not send any packets at all, resulting in a deadlock.

To overcome this problem, we require each receiver to periodically send a congestion summary to its parent. This information is called a *window update acknowledgment*, and it is sent *only if* a congestion summary has not been sent within the last period. In the current implementation, the period within which a window update acknowledgment is sent is initially set to 500 ms, and it is incremented by 1 second each time the receiver sends the acknowledgment. The period is reset to 500 ms when a new packet is received from the sender. This periodic window update information sent by the SAs effectively resolves the deadlock problem.

4 Internet Experiments and Simulation

We have implemented MTCP on top of UDP in Posix Threads and C, in SunOS 5.6. We have not yet incorporated a tree construction module into MTCP, therefore we manually set up the logical trees used for the Internet experiments presented in this section. In the future, MTCP will evolve to include algorithms to dynamically construct and maintain the logical tree. The members of the multicast group in the Internet experiments were distributed in five different sites: North Carolina State University, Raleigh, NC (NCSU), University of North Carolina, Chapel Hill, NC (UNC), Georgia Institute of Technology, Atlanta, GA (GaTech), Emory University, Atlanta, GA (Emory), and University of Texas, Austin, TX (UTexas). We have used an assortment of SPARC Ultras, SPARC 20s and SPARC 5s at each site, organized into the logical tree shown in Figure 6.

For the routing of MTCP packets, we have implemented a special process at each site, called `mcaster`, whose function is similar to that of `mroutd` in the MBONE. The packets generated by the sender at the root of the tree are routed along the edges of the tree using `mcaster`s. An `mcaster` simply "tunnels" incoming packets by first multicasting them to its own subnet via IP-multicast, and then forwarding them to the `mcaster`s of its child sites in the tree via UDP.

Since the experiments are limited by the number of testing sites and machines we can access in the Internet (the tree in Figure 6 consists of 23 receivers and one sender), we have also implemented MTCP on the network simulator `ns` to test the protocol on a larger scale. The experiments presented in this section were designed to evaluate the potential of MTCP for large-scale reliable multicast by addressing three key features of the protocol: scalability, inter-fairness, and sensitivity to independent loss.

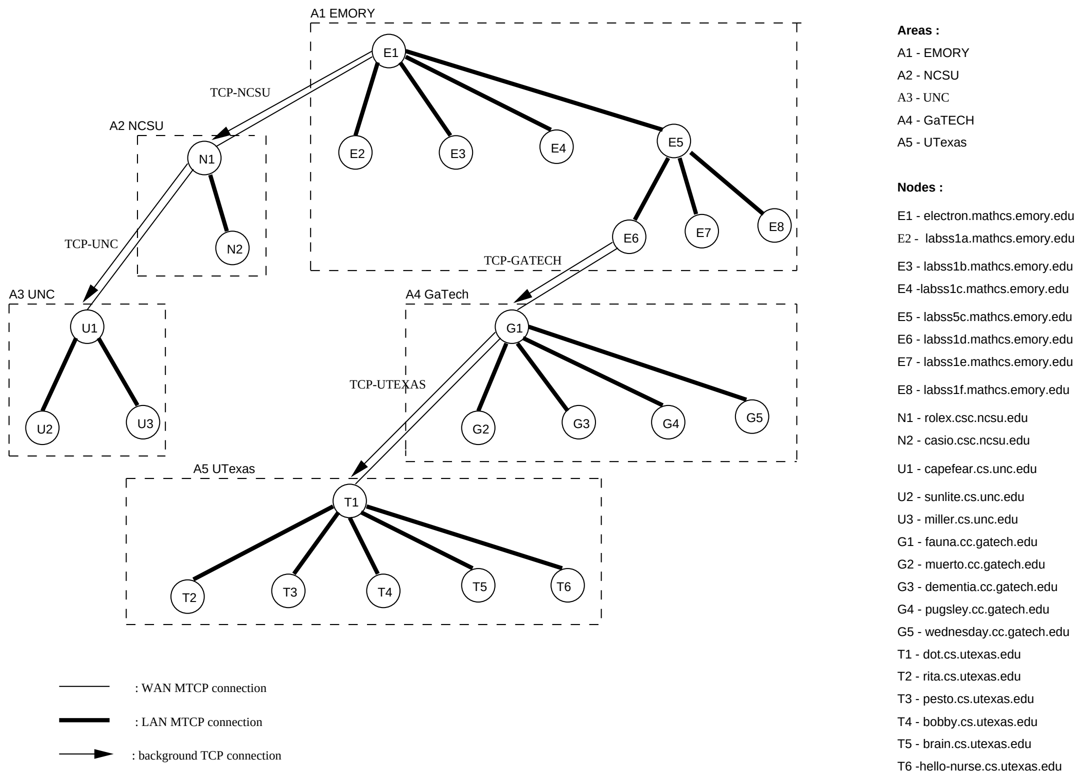

Figure 6: Tree used for experiments

4.1 Scalability

Scalability is an important issue in reliable multicast, since the target applications may involve hundreds or even thousands of receivers. In the case of MTCP, the maximum fanout and the height of the logical tree are two important parameters in determining its ability to scale to large numbers of receivers. In this section, we study the effect of these parameters on the performance of MTCP.

Let us first discuss the significance of the height of the tree. It is well known that the scalability of reliable multicast protocols is directly related to the degree of throughput degradation as the number of receivers increases. Since MTCP emulates TCP on a logical tree, the throughput behavior of MTCP is similar to that of TCP and can be approximated as [3]: $T = \frac{c \cdot s}{RTT\sqrt{p}}$, where s is the packet size, RTT is the round trip time, p is the packet loss rate, and c is some constant. When the maximum fanout is limited to a small constant, the only factor in the expression for T affected by the number of receivers is the round trip time. In MTCP, the RTT grows linearly with the height of the tree, since the sender recognizes congestion through feedback that propagates from a congested node to the sender via the ancestors of the node. In the best case, the throughput of MTCP will degrade in proportion to $\log_f n$, where f is the maximum fanout of the tree and n is the number of receivers. The worst case occurs when the height of the tree grows linearly with n. Consequently, we expect MTCP to achieve a high throughput, even for large numbers of receivers, when a well-balanced tree with a relatively small number of levels is employed.

The second parameter of interest is the number of children that an SA can accommodate, which determines the maximum fanout of the logical tree. In light of the limitations on the tree height, it is desirable to construct trees with a large fanout in order to support a large number of receivers. On the other hand, the larger the number of children attached to an SA, the higher the load imposed on the SA who has to receive and process feedback (ACKs, NACKs and congestion summaries) from its children. Therefore, unless the maximum fanout of the tree is bounded, SAs may become overloaded and the throughput of MTCP will suffer.

4.1.1 Internet Experiments

Our first experiment investigates the maximum fanout of a logical tree that can be supported by MTCP without inducing an excessive load on each SA. The experiment involved a sender transmitting a 70 MB file to multiple destinations on the same LAN. The nodes were organized in a one-level tree rooted at the sender, with all receivers on the same level. We measured the throughput and CPU load at the sender as we increased the number of receivers. We limited the number of receivers to 16, since if each SA can accommodate 16 children, MTCP can support 69,904 receivers organized in a four-level tree. All the machines used in the experiment were Ultra-Sparc Model 250 attached to a 100 Mbps LAN.

Figure 7 plots the throughput, the total transfer time, and the CPU time of the sender, against the number of receivers. The CPU time represents the amount of time that the CPU is used during the transfer of the file, while the total transfer time is the time it takes to transfer the file and includes the time spent by the sender waiting for ACKs. We observe that as the number of receivers increases, the throughput does decrease, but not significantly. We also see that the CPU load (i.e., the CPU time as a fraction of total time) also decreases with the number of receivers. This behavior can be explained by observing that, as the number of receivers increases, the sender spends a greater amount of time waiting for ACKs, and thus total transfer time also increases. Our results indicate that even if the sender and the SAs have as many as 16 children, the processing of ACKs does not pose a problem. In view of the fact that the experiment was performed in a high-speed LAN (where the sender can transmit at a fast rate, and also receives ACKs at a fast rate), the number 16 appears to be a reasonable upper bound on the number of children each SA can have in the logical tree, suggesting that MTCP is suitable for large-scale implementation.

The purpose of our second experiment was to test whether the protocol can respond to congestion within a TCP time-scale, as well as to measure the time delay involved in responding to congestion. To this end, we set up a four-level tree and examined how long it takes for the congestion window of the sender to be adjusted in response to changes in the congestion window of SAs in the path to the congested receiver. The tree involves one machine from each of the following sites: NCSU (the sender), Emory (the first SA, SA1), GaTech (the second SA, SA2), and UTexas (the leaf receiver). The experiment involved a source transmitting a 70 MB file to the three destinations. During the experiment we recorded the congestion window sizes at the sender and the SAs.

Figure 8 shows a five second segment of the experiment. In this experiment we found UTexas to be the bottleneck, which caused SA2 (at GaTech) to have the smallest window size. The sender's window size is the largest. Recall that in MTCP, the sender regulates its transmission rate based on the minimum of all the reported congestion summaries and its own window. Let us call this minimum the *transmission window*. As we can see, the transmission window closely follows the window of SA2. Furthermore, we observe that whenever any site runs into a slow start, the sender reduces the size of its transmission window drastically within about 200 ms to 250 ms. For example, in Figure 8 we see that SA2 initiated a slow start at around 43 seconds, and that about 250 ms later the transmission window also dropped to match the window of SA2.

4.1.2 Simulation

We used simulation to investigate the throughput behavior of MTCP on a larger scale. We simulated the network topology shown in Figure 9, consisting of 101 nodes (one sender and 100 receivers) with each node having at most five children. The links connecting the nodes have a bandwidth of 1.5 Mbps and a delay of 10 ms. The queue size at the nodes was set to 8, and the DropTail queueing discipline was used.

We conducted two experiments to study the effect of the logical tree structure on the throughput of MTCP. In the first experiment, the nodes were organized in a one-level tree with the sender at the root and all receivers as children of the sender. With this arrangement, the sender has to process feedback from all receivers. In the second experiment, the nodes were organized in a perfect multi-level logical tree in which all non-leaf nodes have exactly five children (except, perhaps, the rightmost non-leaf node at the lowest level which may have fewer than five children). As a result, the load of processing feedback was evenly distributed among the SAs in the tree. During each experiment we recorded the throughput of MTCP as we increased the number of receivers from 1 to 100. The total simulation time for each run was 250 seconds.

Figure 10 plots the data throughput (not including header and other control overhead) against the number of receivers. As we can see, when the nodes are arranged as a one-level tree, the throughput decreases with the number of receivers. It is interesting that, although the network topology used in this simulation (see Figure 9) is very different than that used in the LAN experiment in Figure 7, the results obtained are very similar within the same range of the number of receivers (recall that the LAN experiment involved up to 16 receivers). Specifically, in both cases we observe a rather significant drop in throughput after the first few receivers have been added, but then the throughput remains essentially constant until the number of receivers has reached 16 (in the LAN experiment) or 25 (in the simulation). However, in the simulation, the throughput decreases drastically when the number of receivers increases beyond 25. This result is due to the significant increase in the load of the sender who has to process feedback from all receivers.

On the other hand, when a multi-level tree is employed, the throughput does decrease when the number of receivers becomes greater than two, but not significantly. More importantly, the throughput remains at around 125 KBps (1 Mbps) as the number of receivers increases from three to 100, despite the corresponding increase in the height of the tree from one to three levels. These results indicate that the tree structure is the key to reducing the processing load at the sender, and that MTCP provides a high degree of scalability when a proper (i.e., well-balanced) tree is employed. We also note that MTCP achieves a data throughput (excluding overhead) of approximately 1 Mbps on 1.5 Mbps links, which demonstrates that MTCP is successful in capturing the available network bandwidth.

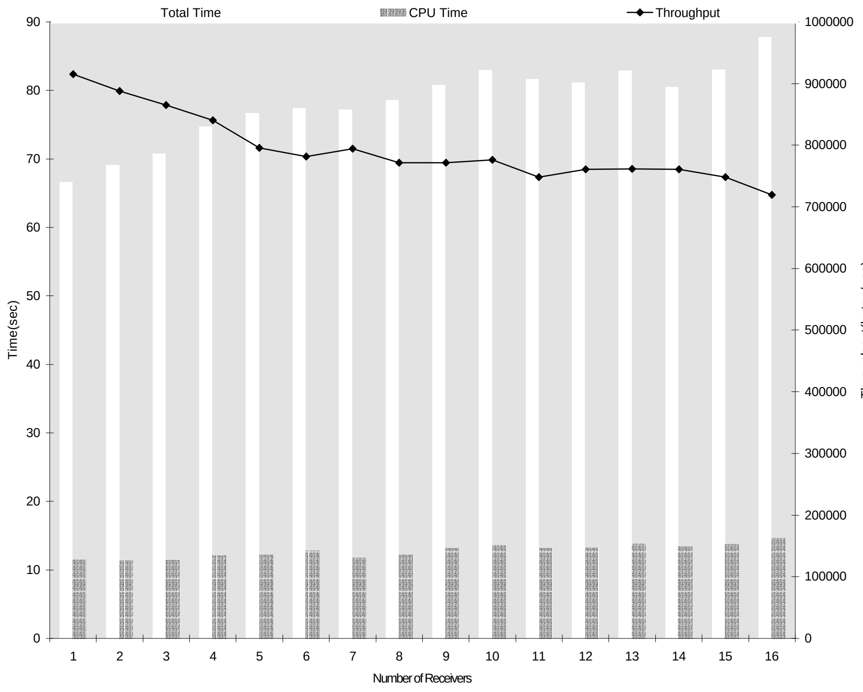

Figure 7: One-level scalability test – LAN experiment

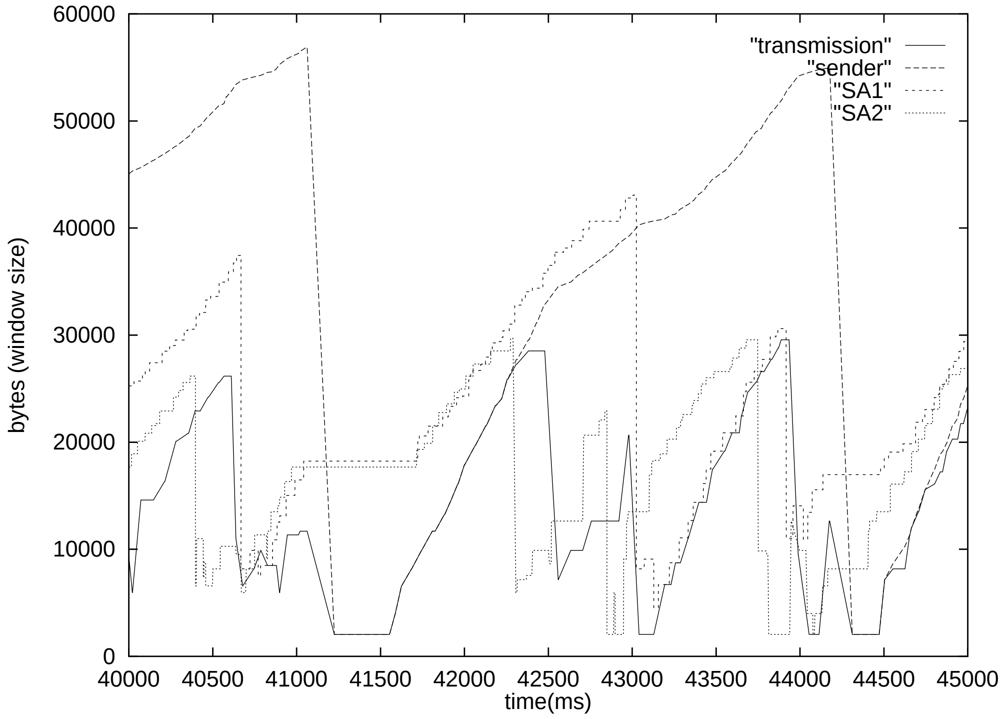

Figure 8: Multi-level response time test – Internet experiment

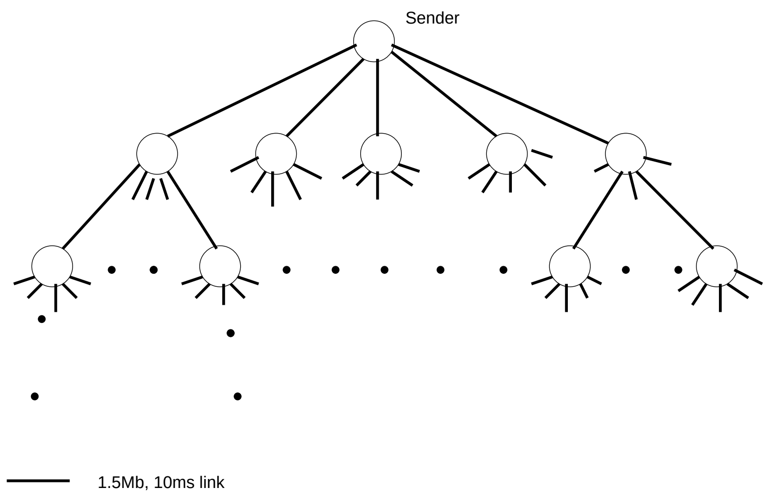

Figure 9: Network topology for scalability tests – Simulation

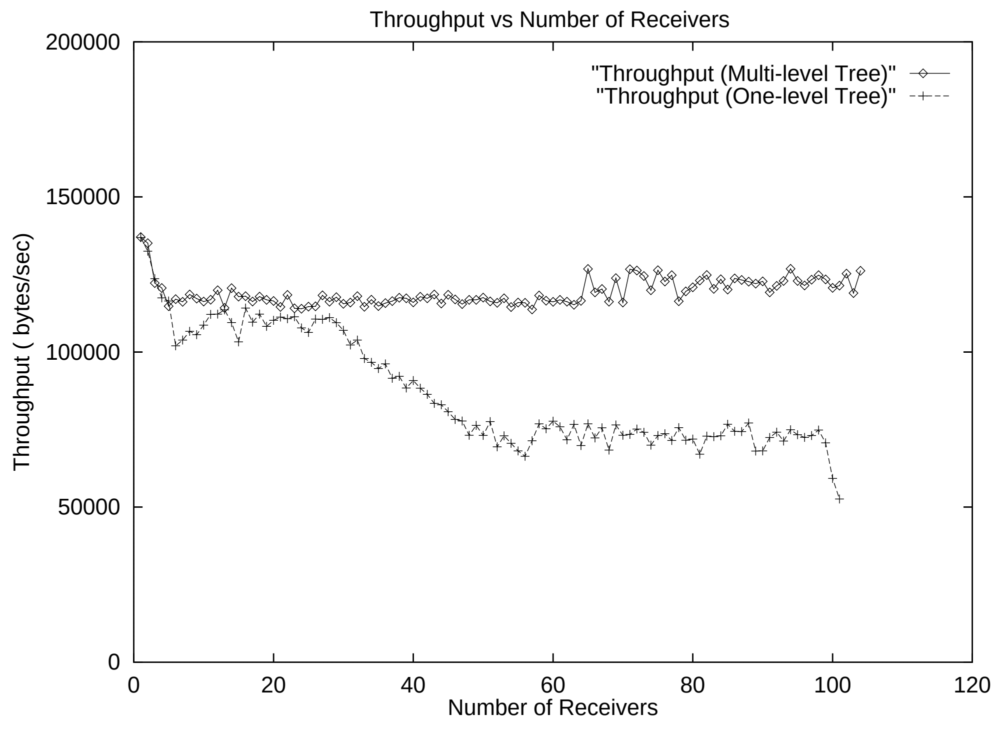

Figure 10: One-level and multi-level scalability tests – Simulation

4.2 Inter-fairness

A protocol is said to be inter-fair if it uses no more bandwidth than a conforming TCP traffic would use on the same link. We have run both Internet experiments and simulation to investigate the inter-fairness properties of MTCP. The Internet experiments test whether MTCP can successfully co-exist with competing TCP traffic, while the simulations test the behavior of competing MTCP flows.

4.2.1 Internet Experiments

We have conducted a large number of experiments over the part of the Internet shown in Figure 6 in order to study the interaction between MTCP and TCP traffic under real-world scenarios. In Figures 11 to 14 we show results from three different experiments. Each experiment involves independent TCP connections running over the WAN routes in the tree of Figure 6. Recall that MTCP packets are routed over the WAN via UDP, thus, the TCP and MTCP traffic between the same sites in our experiments take the same WAN routes. Since WAN links are the ones most likely to be a bottleneck, this setup is appropriate for studying how the bandwidth of a link is shared between MTCP and TCP connections.

The first experiment involves areas A1 and A4 (refer to Figure 6), the second experiment involves areas A1, A2, A3, and A4, and the third involves the entire tree. In these experiments, the MTCP sender and TCP senders transmit data as fast as it is allowed by their congestion control protocols. Each TCP sender starts transmitting at approximately the same time as the MTCP sender. We expect MTCP to match its sending rate to the minimum bandwidth available in the tree, therefore every MTCP receiver should receive at approximately the same rate as the TCP receiver on the bottleneck connection in the tree.

The results of the first experiment (over areas A1 and A4) are shown in Figures 11 and 12. We run MTCP and TCP connections for 300 seconds and recorded the receiving rates of MTCP and TCP receivers. Figure 11 shows the receiving rates of the MTCP and TCP and receivers averaged over 5-second intervals, while Figure 12 shows the receiving rates recorded every second for the first 50 seconds of the experiment. It is evident from both graphs that MTCP and TCP share approximately the same bandwidth of about 280 KBps.

The receiving rates recorded during the second experiment (over areas A1, A2, A3, and A4) are shown in Figure 13; only the average rates over 5-second intervals are plotted in this case. From the figure, it is clear that the route from Emory to NCSU is the bottleneck because the TCP connection between these two sites gives the minimum receiving rates. MTCP matches the TCP receiving rate over this bottleneck route at around 70 KBps.

The results of the third experiment (over the entire tree) are shown in Figure 14, where again we plot the receiving rates averaged over 5-second intervals. The route from Georgia Tech to the University of Texas is now the bottleneck. As we can see, the TCP connection between GaTech and Utexas has the minimum receiving rate of about 60 KBps, indicating that the route between these two sites is the bottleneck link in the whole tree. Again, we observe that MTCP is successful in matching its rate to the receiving rate of the TCP connection on the bottleneck link.

These three experiments indicate that MTCP uses no more bandwidth than a TCP connection uses on the

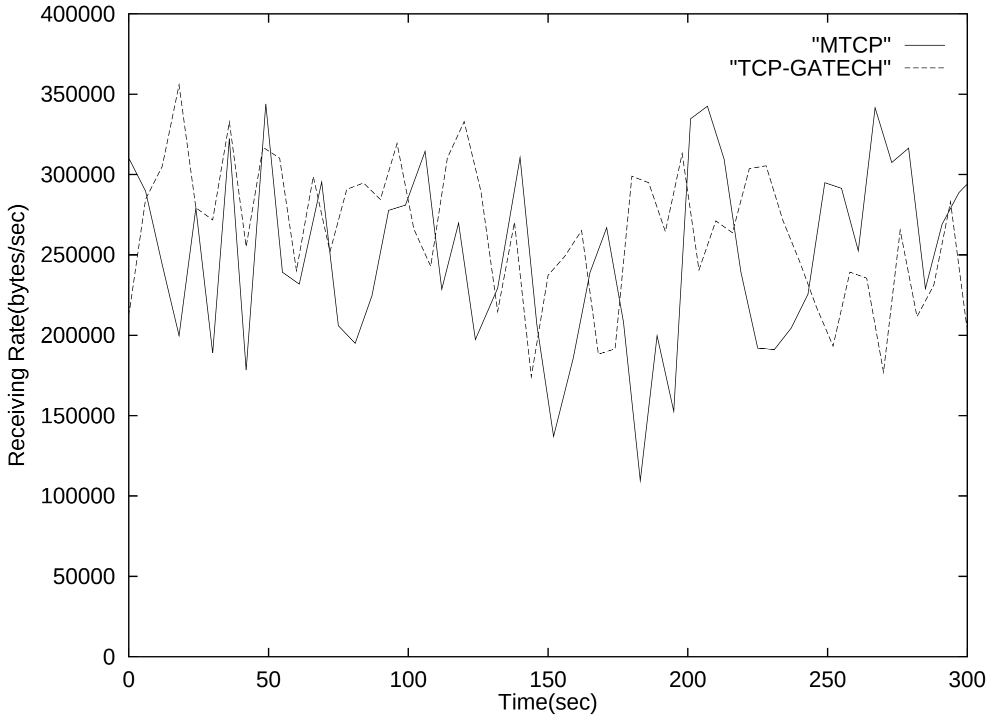

Figure 11: Receiving rates averaged over 5-second intervals (first Internet experiment, areas A1 and A4)

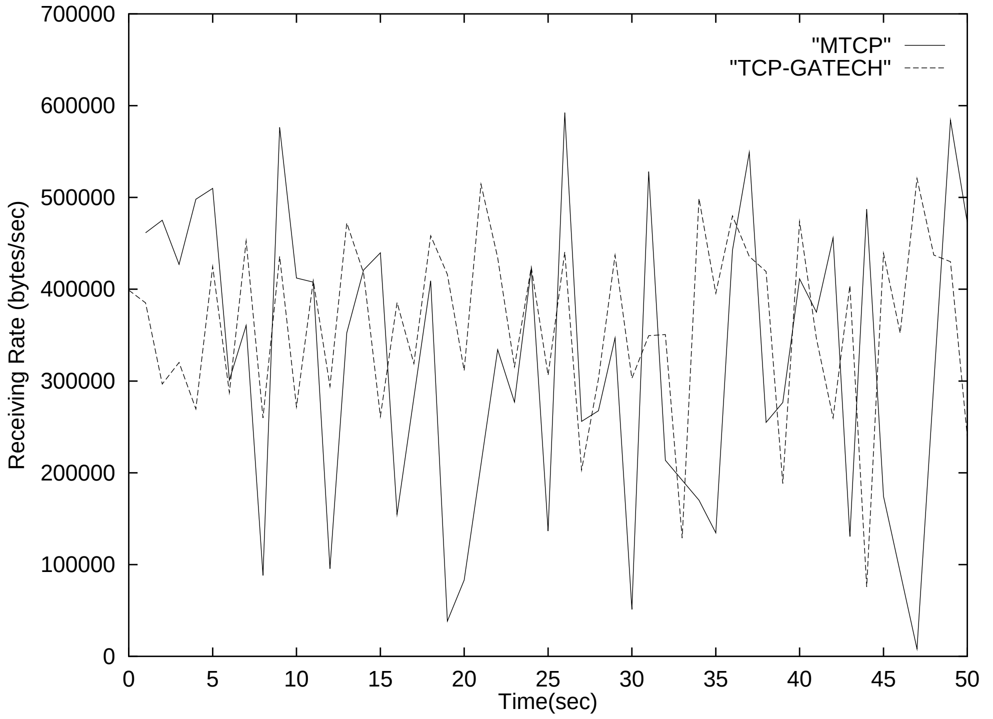

Figure 12: Receiving rates recorded every second (first Internet experiment, areas A1 and A4)

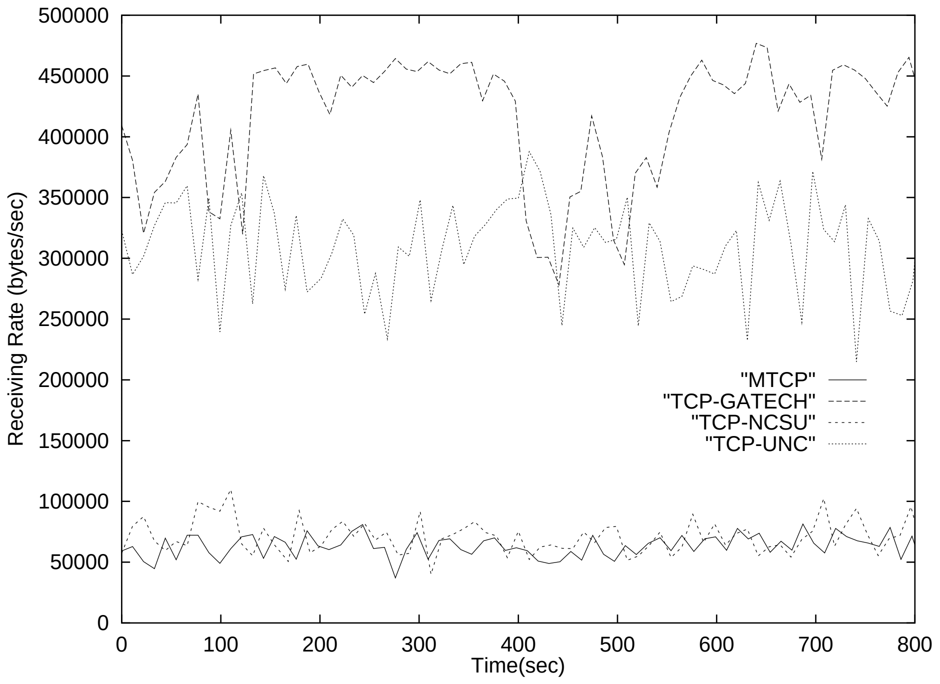

Figure 13: Receiving rates averaged over 5-second intervals (second Internet experiment, areas A1, A2, A3 and A4)

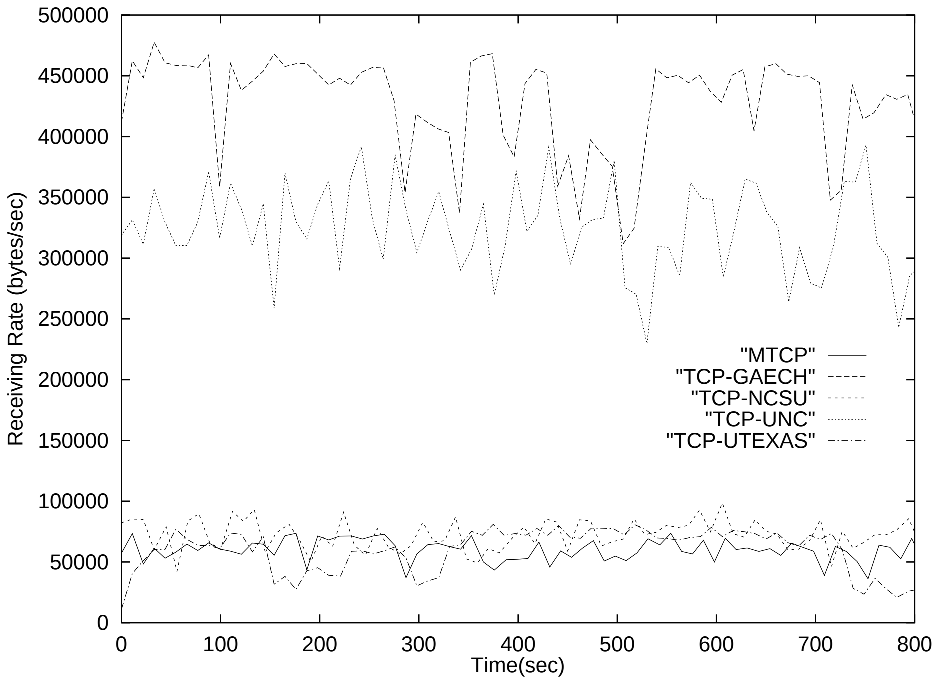

Figure 14: Receiving rates averaged over 5-second intervals (third Internet experiment, areas A1, A2, A3, A4 and A5)

bottleneck route of a given tree configuration. Although a bottleneck link may be located several levels away from the root, MTCP is capable of adjusting its rate according to the available bandwidth on that link. In all experiments, the fluctuation of MTCP's receiving rate is not perfectly synchronized with that of TCP's. This is because MTCP and TCP are not the same protocol, and the way that they detect congestion is different. In addition, MTCP reacts to every instance of congestion within a tree while TCP reacts to congestion only between two end points.

To study the performance of MTCP when sharing a link with multiple TCP connections, we run a fourth experiment involving areas A1 and A4. In this experiment, while MTCP was transmitting, we run three TCP connections, all along the WAN route between Emory and GaTech, each of which is started at a different time. Specifically, the three connections TCP1, TCP2, and TCP3 were started at around 150, 300, and 410 seconds, respectively, after MTCP was started. All TCP connections were made between different host machines to eliminate the effect of computational overhead. We expect to see MTCP adjust its rate to match the current level of bandwidth available over the link between Emory and GaTech.

Figure 15 shows the results of this experiment. When MTCP runs alone, its receiving rate reaches around 400 KBps. When TCP1 is added, MTCP reduces its rate from 400 KBps to 300 KBps while TCP1 traffic slowly increases its rate to around 300 KBps. As soon as TCP2 is added, both TCP1 and MTCP reduce their rates. TCP1 goes down to 180 KBps while MTCP matches its rate with TCP2 around 240 KBps. When TCP3 is added, both MTCP and TCP2 reduce their rates slightly. MTCP still does not use more bandwidth than TCP2. As soon as TCP1 finishes its transmission, MTCP's rate bounces up to match that of TCP2. TCP3 also increases its rate. It appears that TCP3 always uses less bandwidth than TCP2. The difference is about 50 KBps. There could be a couple of reasons for this difference. First, although the two TCP connections use the same route, their end points are different. So there could be other background job activities at the end points of TCP3 affecting its overall receiving rate. Second, TCP itself sometimes can be too conservative in its estimate of the available bandwidth. When TCP2 ends, both TCP3 and MTCP increase their rates quite a bit. MTCP settles at around 330 KBps while TCP3 goes up to 260 KBps. The difference is close to that between the receiving rates of TCP2 and TCP3. As soon as TCP3 ends, MTCP restores its rate quickly to 400 KBps. From this experiment, we observe that MTCP seems to adjust its rate as quickly as TCP, according to the current available bandwidth on the bottleneck link in a given tree.

4.2.2 Simulation

To gain insight into the inter-fairness properties of multiple competing instances of MTCP, we have used `ns` to simulate a large number of network topologies. In this section we present results for the topology shown in Figure 16, which is very similar to the one used in [16, Figure 2]; results obtained from other topologies are very similar to the ones presented here. There are four senders and nine receivers in the network, and each MTCP instance involves one of the senders and all nine receivers. We run three different experiments. In experiment $i,\ i = 1, 2, 3$, each sender node was involved in exactly i MTCP sessions, so that a total of $4 * i$ distinct MTCP connections were simultaneously active in the network. The same logical multi-level tree was used for all instances of MTCP, with the sender at the root and each node in the tree having at most three children. The total simulation time for each experiment was 250 seconds.

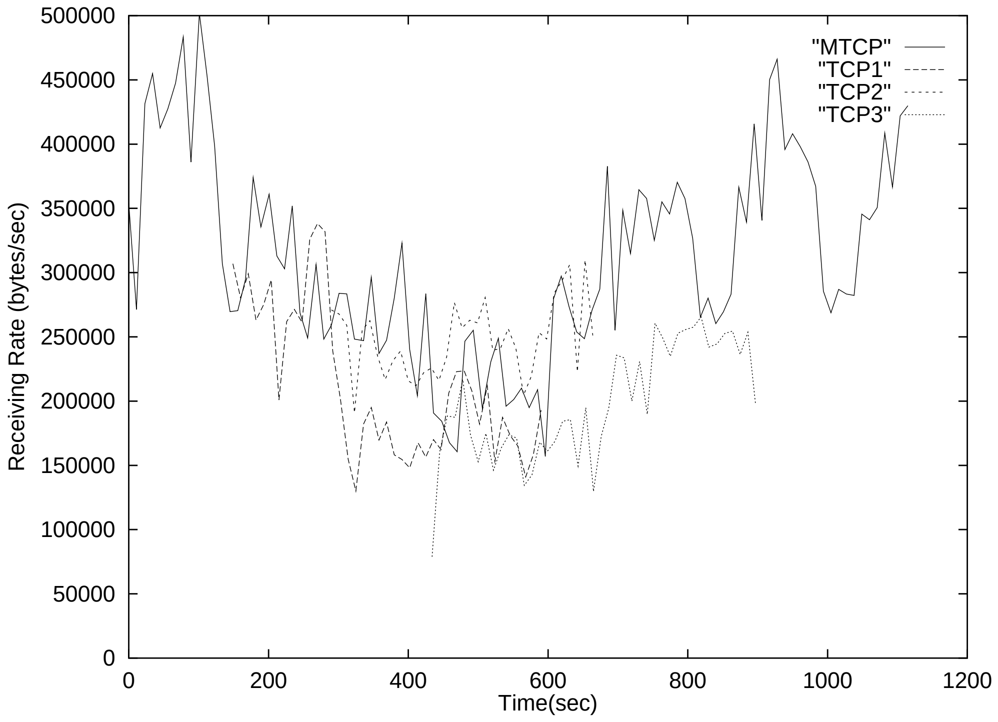

Figure 15: Receiving rates averaged over 5-second intervals (fourth Internet experiment, areas A1 and A4)

In each experiment, we calculated the fairness index defined by Jain *et al.* [17]. The fairness index is a quantitative measure of fairness for a resource allocation system, which is independent of the amount of resource being shared. The fairness index, based on throughput, for the bottleneck link is defined as:

$$FI = \frac{\left[\sum_{x=0}^{N-1} T(x)\right]^2}{N \sum_{x=0}^{N-1} T(x)^2} \tag{1}$$

where $T(x)$ is the throughput of the x-th protocol instance, and N is number of protocol instances sharing the resource. The fairness index always lies between 1 (indicating that all instances get an equal share of the link bandwidth) and $1/N$ (when one of them gets all the bandwidth and all others starve).

Our results are presented in Table 1, where we show, for each experiment, the throughput of each MTCP session and the corresponding fairness index. The bottleneck link is the one between nodes 5 and 8 in Figure 16, since it has a capacity of only 1 Mbps. As we can see, the fairness index was always very close to 1. indicating that MTCP sessions fairly share the available bandwidth.

4.3 Sensitivity to Independent Loss

To study the impact of independent packet loss and random background TCP traffic on the throughput of MTCP, we have conducted simulation involving an MTCP session with 100 receivers. For the results presented in this section, we have used the network topology shown in Figure 17, which is an extension of

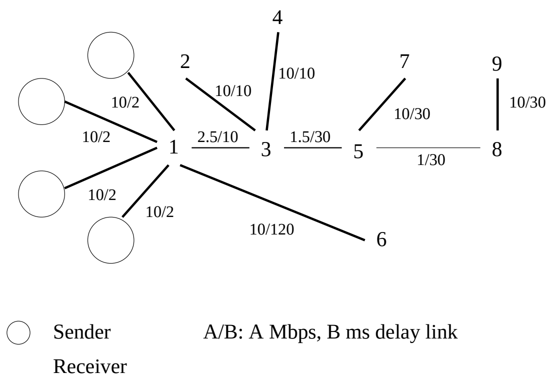

Figure 16: Network topology for inter-fairness experiment – Simulation

the topology used in [18]. This network was generated using Georgia Institute of Technology's Internetwork Topology Models (GT-ITM) tool [19], which has been shown to generate topologies that closely resemble the structure of the Internet. The links connecting the nodes were one of three types, as shown in the figure, the queueing discipline at the routers was DropTail, and the queue size was 20. Several TCP connections were running in the background between the various end-points indicated in Figure 17. TCP connections were started at random times, and each lasted for only a short time period. The MTCP receivers were arranged in a logical tree as shown in Figure 18, with each SA having at most five children.

We run two experiments, each involving a single MTCP connection and random TCP-Reno traffic in the background. In the first experiment, there were no losses at MTCP receivers. In the second experiment, with probability $p = 0.01$, a packet was lost at a MTCP receiver, independently of other receivers and packets. During each experiment, which lasted 1500 seconds, we recorded the throughput at MTCP receivers and TCP sinks. These simulations help us evaluate the throughput of MTCP on a realistic topology under loss patterns that have been observed in real multicast environments.

Figures 19 and 20 plot the throughput of MTCP and of the TCP background connections over time for the no-loss and 1%-loss experiments, respectively. In both cases, MTCP shows about 12 KBps throughput which was a little less than the bandwidth of the bottleneck links in the simulated network topology. When MTCP ran with 1% independent loss, the throughput of MTCP was slightly reduced, but not significantly. The impact of the background TCP traffic also seems very marginal. The graphs show that MTCP does not reduce its rate because of independent uncorrelated losses or due to random TCP traffic. Comparing Figure 19 and 20 we find that the receivers subject to 1% loss have throughput comparable to that of receivers not subject to loss.

5 Related Work

Many reliable multicast protocols have been proposed in the literature [6, 7, 8, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. For the purposes of our discussion, we classify these protocols into three broad categories: *unstructured*, *structured* and *hybrid*. We examine the protocols in each category with an emphasis on their

	Experiment 1 4 MTCP sessions	Experiment 2 8 MTCP sessions	Experiment 3 12 MTCP sessions
MTCP Throughput (bps)	122976.32	71762.67.	65616.889
	120163.84	111270.22	64457.778
	121892.08	119783.11	96119.111
	118946.12	137866.67	85045.333
		130666.67	92062.222
		130311.11	97333.333
		145478.22	68444.444
		126960.00	71111.111
			83555.556
			65777.778
			97777.778
			41452.764
FI	.999836	.970422	.955753

Table 1: Inter-fairness results – Simulation

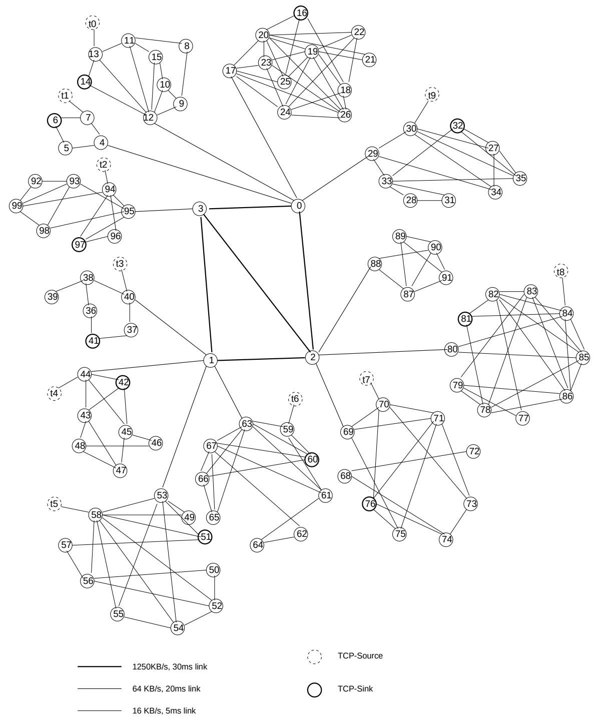

Figure 17: Network topology for experiments with random TCP background traffic – Simulation

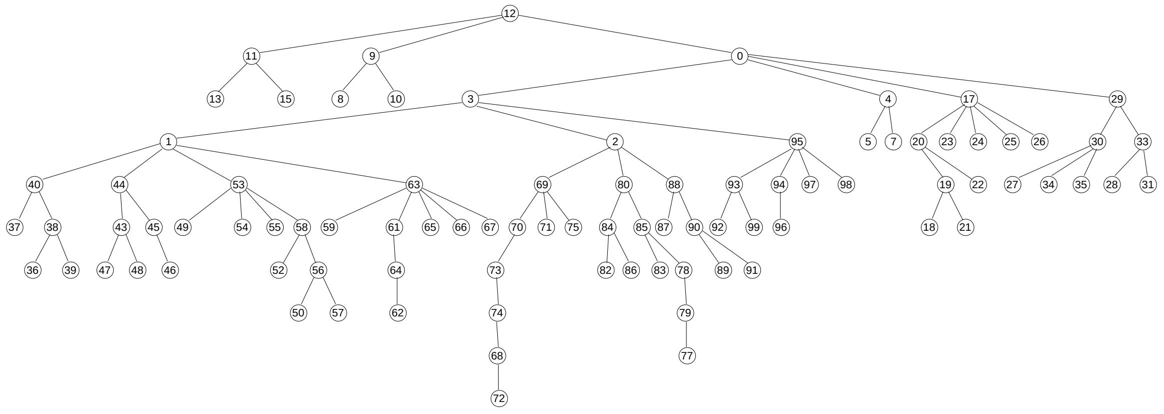

Figure 18: Logical multicast tree for the topology of Figure 17 – Simulation

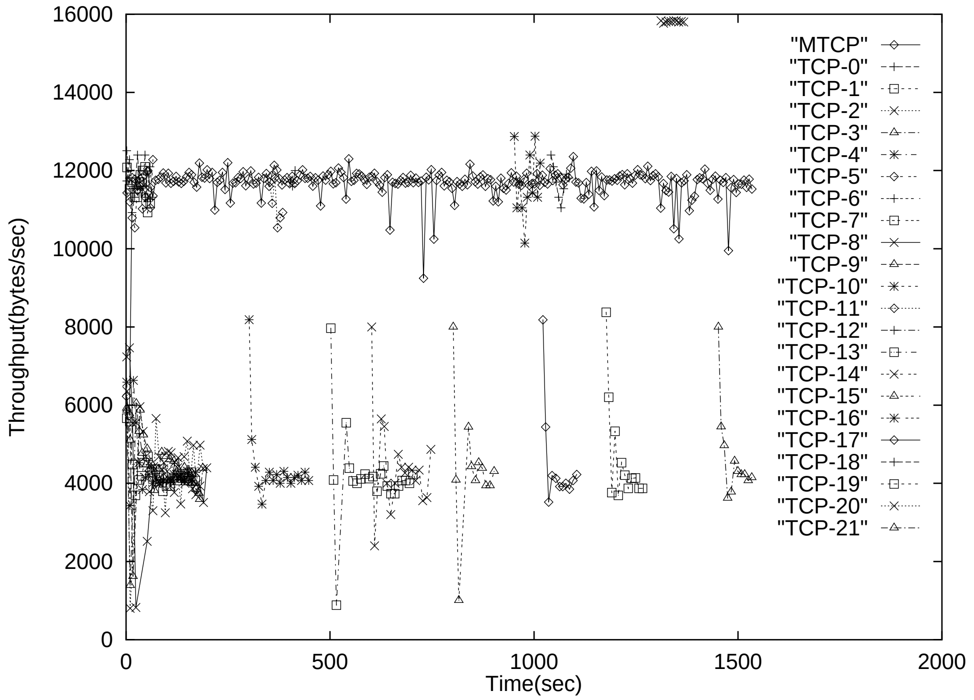

Figure 19: Throughput of MTCP and of background TCP traffic – no loss

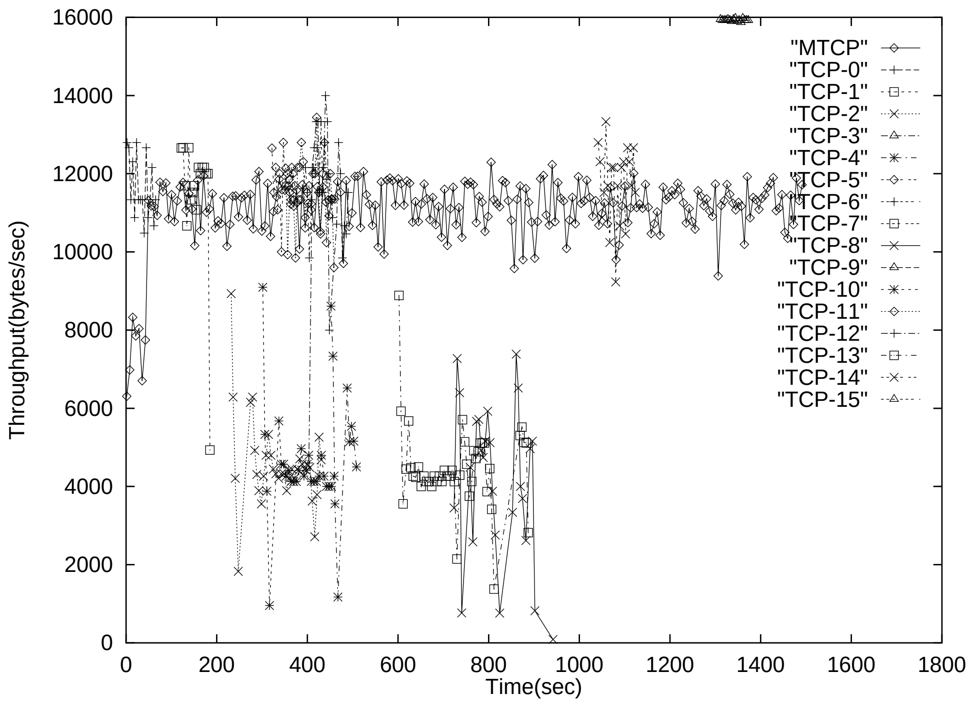

Figure 20: Throughput of MTCP and of background TCP traffic – 1% loss rate at MTCP receivers

congestion control techniques.

Unstructured protocols do not impose any structure among receivers, and Pingali *et al.* [30] further classify them into *sender-based* [20, 22, 26, 28, 27, 29] and *receiver-based* protocols [24, 25]. In sender-based protocols, every receiver sends ACKs or NACKs directly to the sender, and the sender retransmits lost packets reported in NACKs. The main problem with sender-based protocols is the *feedback implosion* problem: if many receivers send ACKs or NACKs to the sender at the same time, the sender may quickly become overloaded. This problem is especially severe when losses occur near the sender, in which case a large number of receivers will experience packet loss. In a system involving more than a few receivers, the load imposed by the storm of acknowledgments limits the function of the sender.

In a receiver-based protocol, each receiver multicasts NACKs to all members of the group, and any receiver that has received the requested packets multicasts them to the group. Typically, the protocols incorporate randomized NACK and retransmission suppression timers to reduce the number of duplicate NACKs and retransmissions. We find three main shortcomings with receiver-based protocols.

First, Yajnik *et al.* [31] report that most packet loss in the Internet Multicast Backbone (MBONE) occurs not at the backbone, but near end receivers, and that even excluding packet loss occurring near the sender, a small, but still significant, amount of loss (about 1% to 30%) involves more than two receivers. This study suggests that it is highly likely for two receivers not sharing common multicast routes to lose the same packets. A randomized NACK suppression technique may cause some uncorrelated NACKs to suppress correlated NACKs which actually report about a congested link. Since NACKs are multicast, the sender would get NACKs, but possibly from a different receiver each time: it may appear to the sender that NACKs are completely uncorrelated. Thus, the sender may not distinguish correlated packet losses from uncorrelated ones. It is unclear whether the sender should respond to all the NACKs by controlling its rate or ignore "seemingly" uncorrelated NACKs. Either approach seems unreasonable.

Second, in most receiver-based protocols that primarily use NACKs to detect congestion, the absence of NACKs is considered as no congestion or congestion clearance. Some Internet studies [31, 32], however, reveal that almost every experiment trace includes one or more extremely long bursts of packet loss lasting from a few seconds up to a few minutes. During these bursts, no packets are received. As a result, receivers do not detect any packet loss, and do not send any NACKs. A similar scenario arises when the the return path from receivers to the sender is congested, so that all feedback is lost. In either case, the sender would incorrectly translate the lack of feedback as no congestion.

Third, the randomized NACK suppression techniques employed by the receiver-based protocols require each receiver to estimate the round trip time (RTT) to every receiver in the group. This approach requires $O(n^2)$ RTT estimations by every receiver, thus imposing limits on scalability. Grossglauser [33] proposed a distributed deterministic timeout estimation protocol that does not require global information. However the protocol assumes that the end-to-end delay variation is bounded and *a priori* known to all receivers.

Structured protocols impose a logical structure among group members. Two commonly studied structures are *rings* and *trees*. In ring protocols [23], a logical ring of group members is formed. Typically, a token is passed around the ring and only the process with the token may send feedback to the sender. RMP [23] supports TCP-like congestion control based on both ACKs and NACKs. However, since only the token holder can send an ACK, it is unclear how the ACKs are used for purposes of congestion control when there is a large number of nodes in the ring. In RMP, since NACKs are also multicast to suppress other NACKs, the protocol suffers from problems similar to those arising in receiver-based protocols.

In a tree protocol [5, 6, 7, 8, 9], a logical tree structure is imposed on the multicast group, with internal nodes acting as representative receivers for the group. While the sender multicasts data to the entire group, a receiver sends feedback only to its parent. The representatives buffer packets received from the sender, and retransmit any packets reported lost by their children. Since the maximum degree of each node is fixed to a small constant, each node, including the sender, receives only a small amount of feedback within a round trip time. In the following, we discuss the congestion control schemes of RMTP [6] and TMTP [8], since the LBRM [7], LGC [9] and LORAX [5] tree protocols do not incorporate (or do not give much detail about) a congestion control scheme.

The main problem with RMTP is that it does not provide end-to-end feedback. The sender only gets feedback from its own children (called *designated receivers* (DR)) about their receiving status. Hence, the sender has little information about the congestion status of leaf receivers. When congestion occurs at leaf receivers, it may not be possible for the sender to detect the congestion, especially if the DRs and the leaf receivers do not share the same network path. In this case, the sender will continue to transmit at the same rate, aggravating the existing congestion. As a result, RMTP traffic can be completely unresponsive to congestion and may cause congestion collapse.

TMTP also does not provide end-to-end feedback. This protocol relies on a back pressure effect caused by lack of buffers at representative nodes (called *domain managers* (DM) in TMTP terminology). In TMTP, DMs store the packets received from the sender until they receive ACKs for the packets from their children. When the buffers at a DM fill up because of congestion, the DM drops the next incoming packet. Its parent will continue to retransmit the packets not acknowledged by the DM until the parent's buffers also fill up. The sender detects congestion only when the buffers of all DMs between the sender and the congested nodes are completely full. So the congestion is completely neglected until the sender feels the pressure. Since each DM typically maintains a large number of buffers to reduce the number of ACKs returned to it, it could take a long time before the sender feels the pressure and reduces its rate. The fact that TMTP continues to transmit at a fixed rate despite the congestion is unfair to TCP-compatible flows which reduce their rates at the first indication of congestion.

Hybrid protocols [18, 34] combine the packet recovery techniques used in structured and unstructured protocols. As in receiver-based protocols, a receiver can multicast NACKs suppressing other NACKs, while other receivers may respond to the NACKs by retransmitting lost packets. In addition, a small number of representative receivers multicast their feedback immediately without any delay or suppression. The sender uses this feedback to control its transmission rate.

Delucia and Obraczka [18] proposed a hybrid congestion control technique in which the size of the representative set is fixed, but the actual nodes in the set change over time based on the congestion status of receivers. Assuming that a small set of bottleneck links always causes the majority of the congestion problem, the protocol solves the feedback implosion problem, as well as other problems associated with SRM [24] (such as the RTT estimation problem). The scalability and utility of the protocol highly depend on this basic assumption, namely, that the representative set is always small. This assumption may not be realistic, however, since several group members can be independently and simultaneously congested although they do not share the same congested links. No safeguard against this situation is provided.

Handley [34] also recently proposed a hybrid congestion control architecture. His technique works as follows. A small set of representative receivers is selected based on their loss characteristics, and each representative forms a subgroup along with receivers that share similar loss characteristics. For each subgroup, one relay receiver is chosen to receive data from the sender and play them out at a slower rate suitable for the receivers in the subgroup. The idea of representatives is similar to that in [18], but the subgroup idea is new and promising. However, the overhead, complexity, and efficacy of dynamic subgroup formations are not yet explored, justified or proven. In addition, since the group structure is essentially two-level, it is not clear whether the protocol is scalable to very large numbers of receivers.

Other types of protocols that do not fall within the above categories include receiver-driven layered multicast protocols [35, 16, 36]. These protocols implement congestion control by encoding the transmitted data into multiple layers and transmitting each layer to a different multicast group. By joining and leaving different multicast groups, each receiver can control its own receiving rate. Initially, the layering technique was proposed for continuous multimedia data streams which can tolerate some loss. Recently the technique was applied to a reliable bulk data multicast by Vicisano *et al.* [16]. However, the technique is applicable only when a large portion of the data is available for encoding prior to transmission, but not when data is generated in real-time such as during synchronous collaborative conferences.

6 Concluding Remarks

We have presented MTCP, a set of congestion control mechanisms for tree-based reliable multicast protocols. MTCP was designed to effectively handle multiple instances of congestion occurring simultaneously at various parts of a multicast tree. We have implemented MTCP, and we have obtained encouraging results through Internet experiments and simulation. In particular, our results indicate that (1) MTCP can quickly respond to congestion anywhere in the tree, (2) MTCP is TCP-compatible, in the sense that MTCP flows fairly share the bandwidth among themselves and various TCP flows, (3) MTCP is not affected by independent loss, and (4) MTCP flow control scales well when an appropriate logical tree is employed. Thus, we believe that MTCP provides a viable solution to TCP-like congestion control for large-scale reliable multicast. We are currently working on designing and implementing a set of mechanisms for addressing the intra-fairness problem of reliable multicast protocols, i.e., to prevent a slow receiver from slowing down the whole group.

References

[1] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM*, pages 314–329. ACM, August 1988.

[2] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommedations on queue management and congestion avoidance in the Internet. *Internet Draft*, March 1997.

[3] S. Floyd and K. Fall. Router mechanisms to support end-to-end congestion control. Technical report, Lawrence Berkeley Laboratory, February 1997.

[4] T. Jiang, M. H. Ammar, and E. W. Zegura. Inter-receiver fairness: A novel performance measure for multicast abr sessions. In *Procceedings of ACM SIGMETRICS/PERFORMANCE '98*, pages 202–211. ACM, June 1998.

[5] B. N. Levine, D. B. Lavo, and J. J. Garcia-Luna-Aceves. The case for reliable concurrent multicasting using shared ack trees. In *Proceedings of Multimedia 1996*, pages 365–376. ACM, 1996.

[6] S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). In *Proceedings of INFOCOM '96*. IEEE, March 1996.

[7] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of SIGCOMM*, pages 328–341. ACM, August 1995.

[8] R. Yavatkar, J. Griffioen, and M. Sudan. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of Multimedia 1996*. ACM, 1996.

[9] M. Hofmann. A generic concept for large-scale multicast. In *B. Plattner (ed.), Broadband Communications, Proceedings of International Zurich Seminar on Digital Communications (IZS '96)*. LNCS 1044, Springer Verlag, February 1996.

[10] B. N. Levine and J. J. Garcia-Luna-Aceves. A comparision of known classes of reliable multicast protocols. In *Proceedings of International Conference on Network Protocols*. IEEE, October 1996.

[11] M. Hofmann. Adding scalability to transport level multicast. In *Proceedings of Third COST 237 Workshop - Multimedia Telecommunications and Applications*. Springer Verlag, November 1996.

[12] S. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, December 1991.

[13] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM '94*, pages 24–35. ACM, May 1994.

[14] S. Floyd. Requirements for congestion control for reliable multicast. *The Reliable Multicast Research Group Meeting in Cannes*, September 1997.

[15] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[16] L. Vicisano, L. Rizzo, and J. Crowcroft. TCP-like congestion control for layered multicast data transfer. In *Proceedings of INFOCOM '98*. IEEE, April 1998.

[17] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *Computer Communications Review*, 19:56–71, October 1989.

[18] D. DeLucia and K. Obraczka. Multicast feedback supression using representatives. In *Proceedings of INFOCOM '97*. IEEE, April 1997.

[19] E. Zegura, K. Calvert, and S. Bhallacharjee. How to model an internetwork. In *Proceedings of INFOCOM '96*. IEEE, March 1996.

[20] Alfred C. Weaver. *Xpress Transport Protocol Version 4*. University of Virginia.

[21] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. In *Internet Request for Comments RFC 1301*, February 1992.

[22] K. P. Birman and T. Clark. Performance of the Isis distributed computing toolkit. Technical Report 94-1432, Cornell University's Computer Science Department, June 1994.

[23] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems*. LNCS 938, Springer Verlag.

[24] S. Floyd, V. Jacobson, S. McCanne, C. G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of SIGCOMM '95 Conference*, pages 342–356. ACM, October 1995.

[25] B. Sabata, M. J. Brown, and B. A. Denny. Transport protocol for reliable multicast: TRM. In *Proceedings of the IASTED International Conference on Networks*, pages 143–145, January 1996.

[26] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(9):64–70, April 1996.

[27] J. R. Cooperstock and S. Kotsopoulos. Why use a fishing line when you have a net? an adaptive multicast data distribution protocol. In *Proceedings of 1996 USENIX Technical Conference*, January 1996.

[28] A. Koifman and S. Zabele. A reliable adaptive multicast protocol. In *Proceedings of INFOCOM '96*, pages 1442–1451. IEEE, March 1996.

[29] I. Rhee, S. Y. Cheung, P. W. Hutto, and V. S. Sunderam. Group communication support for distributed multimedia and CSCW systems. In *Proceedings of 17th Interanational Conference on Distributed Computing Systems*. IEEE, June 1997.

[30] S. Pingali, D. Towsley, and J. F. Kurose. A comparision of sender-initiated and receiver-initiated reliable multicast protocols. In *Proceedings of SIGMETRICS*. ACM, 1994.

[31] M. Yajnik, J. Kurose, and D. Towsley. Packet loss correlation in the MBONE multicast network. Technical Report CMPSCI 96-32, University of Massachussetts, Amherst, MA.

[32] V. Paxson. End-to-end routing behavior in the Internet. In *Proceedings of SIGCOMM '96*. ACM, August 1996.

[33] M. Grossglauser. Optimal deterministic timeouts for reliable scalable multicast. In *Proceedings of INFOCOM '96*, pages 1425–1432. IEEE, March 1996.

[34] M. Handley. A congestion control architecture for bulk data transfer. In *The Reliable Multicast Research Group Meeting in Cannes*, September 1997.

[35] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of SIGCOMM '96*, pages 117–130. ACM, August 1996.

[36] T. Turletti, J. C. Bolot, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. In *Proceedings of SIGCOMM '94*. ACM, August 1994.