MTCP: Scalable TCP-like Congestion Control for Reliable Multicast

Injong Rhee[†] *Nallathambi Balaguru*[‡] *George N. Rouskas*[†]
[†]Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534
[‡]A&T Systems Inc., 12520 Prosperity Drive, Silver Spring, MD 20904

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.*

1 Introduction

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, many of which 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, and 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 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* 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.

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 behave similarly to its own, TCP-incompatible traffic can completely lock out competing TCP flows and monopolize the available bandwidth. Furthermore, multicast flows insensitive to 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 far-reaching damage of TCP-incompatible multicast, 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

0-7803-5417-6/99/$10.00 ©1999 IEEE.

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 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 also incorporates several novel features, described in the next section. 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 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.

In Section 2 we provide an overview of MTCP, in Section 3 present a description of the protocol, in Section 4 present results from Internet experiments, and conclude the paper in Section 5.

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. 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]. SAs in the logical tree store packets received from the sender in their buffers, and set a *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 (ACK) or a negative (NACK) acknowledgment to its parent in the tree. 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 retransmits a packet via unicast (a) upon receiving a NACK reporting that the packet is missing, or (b) if it does not receive an 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. 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.

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 `cwnd`s and the `cwnd`s reported by their children (`minCwnd`), and (2) the maximum of their `twnd`s and the `twnd`s 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, MTCP does not provide closed-loop feedback: SAs have to adjust their windows based on the ACKs for packets that another node (the sender) transmitted. In this open-loop system, an SA cannot accurately estimate the round trip time (RTT) of a packet because of 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 eas-

ily 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.

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 (not presented here) 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.

When an SA receives an NACK for a packet, 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 identify the packets lost by its children and retransmit only them, avoiding unnecessary retransmissions. Each ACK also contains the send time of the packet acknowledged, 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.

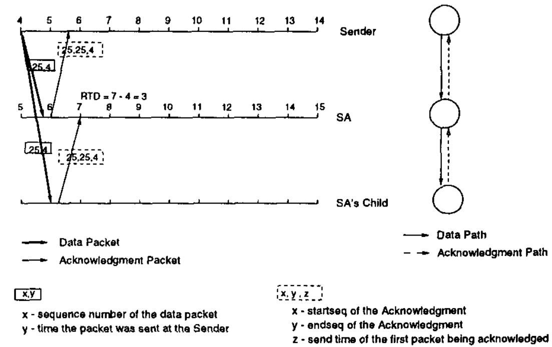

Figure 1: Example of RTD measurement

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 that 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 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 packet carry its transmission time at the sender, and that each ACK carry the transmission time of the packet with sequence number `startseq` $- 1$ (as explained above), to help SAs compute the corresponding RTDs.

We use RTD in the same way as RTT 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 1 illustrates how the RTD is measured. In this example, 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, 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 Timeouts and Timer Backoff

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. 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.

When a packet is initially transmitted by the sender, or is received for the first time by an SA, the sender(or SA) buffers it and set its retransmission timeout 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 2 and 3 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 2 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).

Figure 2: Example of timeouts based on RTD

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 3, 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.

3.5 Slow Start

MTCP employs a slightly modified version of slow start [1] 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 the *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

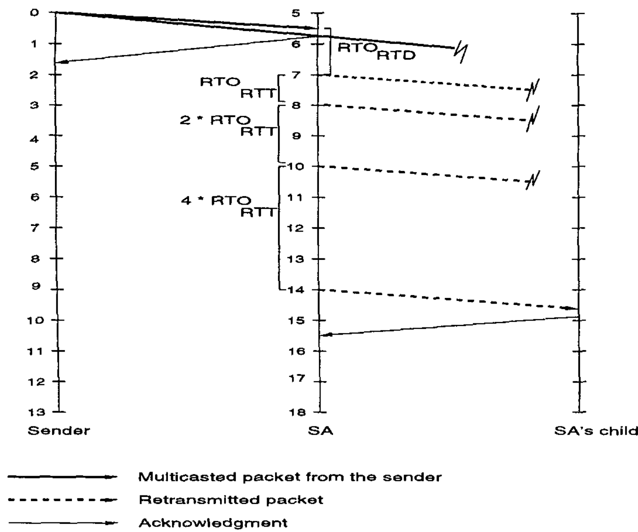

Figure 3: Example of timeouts and exponential retransmission timer backoff

seen so far, and mRTD which is the currently measured RTD. 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.

3.6 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 Diff = mRTD − 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 Diff $< \alpha$, we increase cwnd by 1/(segment size) and when Diff $> \beta$, we decrease cwnd by cwnd/8. cwnd remains unchanged if $\alpha <$ 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.

3.7 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 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). The size of the window changes in the same way that TCP-Vegas modifies cwnd (i.e., slow start, congestion avoidance, and fast recovery).

3.8 Window-Based Flow Control

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 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.

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 cwnds 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 twnds 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 mechanism prevents the sender from transmitting faster than packets leave the bottleneck link.

3.9 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 4 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 (normally defined in bytes). The SAs of leaf nodes send their cwnds and twnds to their parents. Upon receiving this information, the parent SAs send to their own parents the minimum of the received cwnds and their own cwnd, and the maximum of the received twnds and their own twnd. The sender sends no

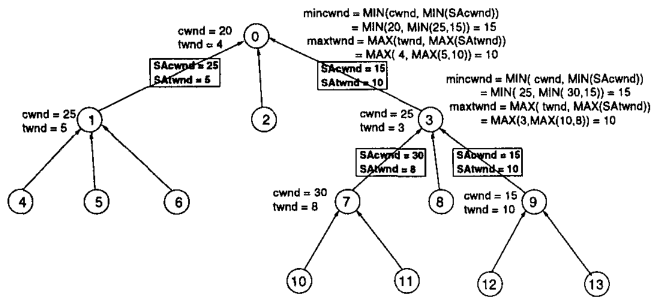

Figure 4: Example of hierarchical status reports

more than the minimum of the current window and the advertised window. 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 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.

3.10 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. 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.

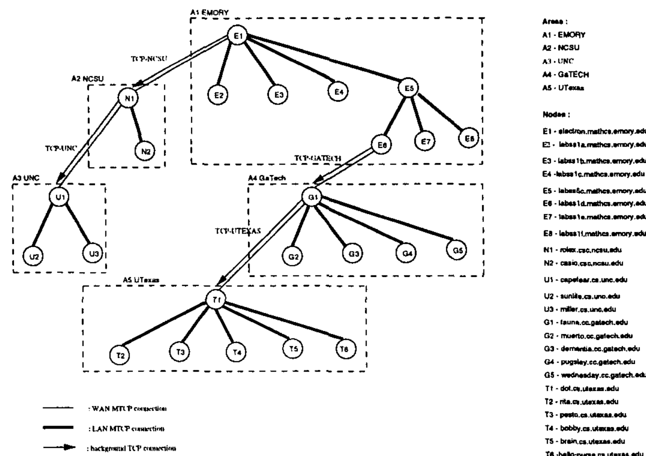

Figure 5: Tree used for experiments

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

We have implemented MTCP on top of UDP in Posix Threads and C, in SunOS 5.6. The members of the multicast group in the Internet experiments were distributed in the five different sites shown in Figure 5. We have used an assortment of SPARC Ultras, SPARC 20s and SPARC 5s at each site, organized into the logical tree in the figure. 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 `mcasters`. An `mcaster` simply "tunnels" incoming packets by first multicasting them to its own subnet via IP-multicast, and then forwarding them to the `mcasters` of its child sites in the tree via UDP.

As the experiments are limited by the number of testing machines we can access in the Internet (the tree in Figure 5 consists of 23 receivers and one sender), we have also implemented MTCP on the network simulator `ns` to test it in a larger scale. Due to lack of space, however, no simulation results are reported here.

4.1 Scalability

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. 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 in-

creases. 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.

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 250 attached to a 100 Mbps LAN.

Figure 6 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 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.

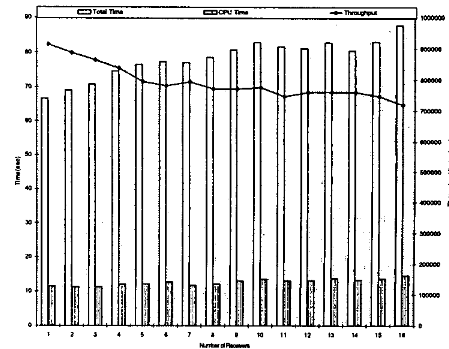

Figure 6: One-level scalability test – LAN experiment

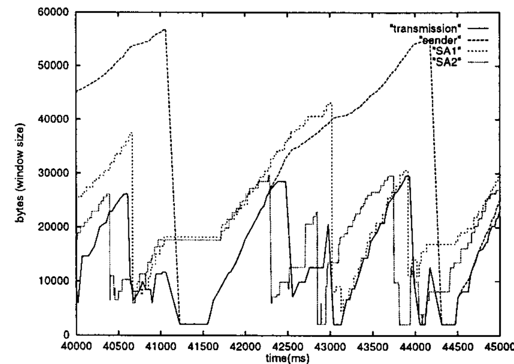

Figure 7: Multi-level response time test – Internet experiment

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 7 shows a five second segment of the experiment. In this experiment we found UTexas to be the bottleneck, which caused SA2 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 7 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.

4.2 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 conducted a large number of experiments over the part of the Internet shown in Figure 5 in order to study the interaction between MTCP and TCP traffic under real-world scenarios. In Figures 8 to 10 we show results from three different experiments. Each experiment involves independent TCP connections running over the WAN routes in the tree of Figure 5. 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 5), 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 result of the first experiment (over areas A1 and A4) is shown in Figure 8. We run MTCP and TCP connections for 300 seconds and recorded the receiving rates of MTCP and TCP receivers. Figure 8 shows the receiving rates of the MTCP and TCP and receivers averaged over 5-second intervals. It is evident from the graph that MTCP and TCP share approximately the same bandwidth of about 280 KBps.

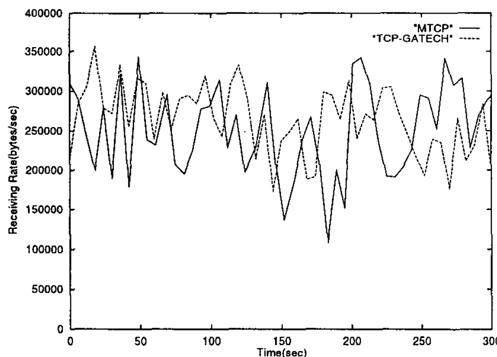

Figure 8: Receiving rates averaged over 5-second intervals (first Internet experiment, areas A1 and A4)

The receiving rates recorded during the second experiment (over areas A1, A2, A3, and A4) are shown in Figure 9; 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

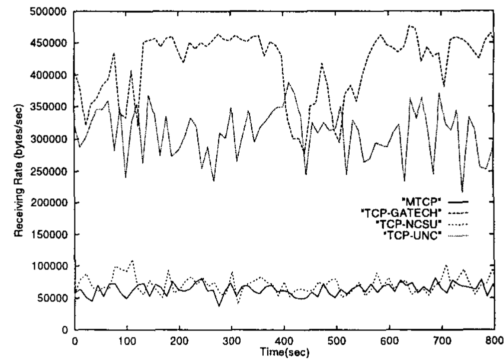

Figure 9: Receiving rates averaged over 5-second intervals (second Internet experiment, areas A1, A2, A3 and A4)

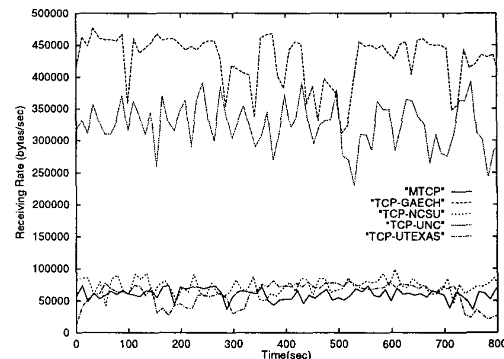

Figure 10: Receiving rates averaged over 5-second intervals (third Internet experiment, areas A1, A2, A3, A4 and A5)

in Figure 10, 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 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.

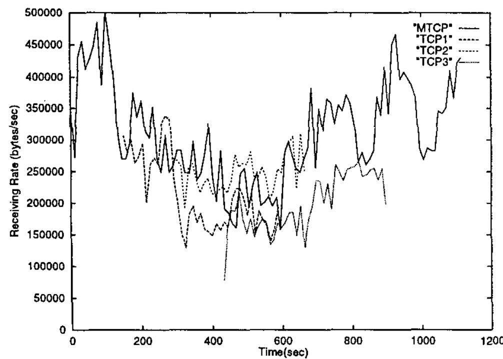

Figure 11: Receiving rates averaged over 5-second intervals (fourth Internet experiment, areas A1 and A4)

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 11 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.

5 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.

References

[1] V. Jacobson. Congestion avoidance and control. *Proc. of SIGCOMM*, pages 314–329, Aug. 1988.

[2] B. Braden, *et al.* 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. Tech. Report, LBL, Feb. 1997.

[4] T. Jiang, M. H. Ammar, and E. W. Zegura. Inter-receiver fairness: A novel performance measure for multicast ABR sessions. *Proc. of SIGMETRICS*, pages 202–211, 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. *Proc. of Multimedia*, pages 365–376, 1996.

[6] S. Paul, *et al.* Reliable multicast transport protocol (RMTP). *Proc. of INFOCOM*, March 1996.

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

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

[9] M. Hofmann. A generic concept for large-scale multicast. *Proc. of IZS '96*, Springer Verlag, Feb. 1996.

[10] B. N. Levine and J. J. Garcia-Luna-Aceves. A comparision of known classes of reliable multicast protocols. *Proc. of ICNP*, Oct. 1996.

[11] M. Hofmann. Adding scalability to transport level multicast. *Proc. of 3rd COST 237 Workshop*, Springer Verlag, Nov. 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. *Proc. of SIGCOMM*, pages 24–35, 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.