A Congestion Control Algorithm for Tree-based Reliable Multicast Protocols

Dah Ming Chiu, Miriam Kadansky, Joe Provino
Sun Microsystems Laboratories, Burlington MA
Joseph Wesley†, Hans-Peter Bischof‡, Haifeng Zhu∗
†Nakra Labs Inc, ‡Rochester Institute of Technology, ∗Carnegie Mellon University.

Abstract— **This paper contains a detailed description of the congestion control algorithm of TRAM, a tree-based reliable multicast protocol. This algorithm takes advantage of regular acknowledgements from the receivers that propagate back to the sender via the repair tree. This scalable feedback mechanism is used to collect receiver credits. Complementing the windowing mechanism, packet transmission is smoothed by using a data rate commensurate with the window size. Additional details, such as how to prune slow receivers, and how to implement the rate scheduler on non-real-time systems are also discussed. The performance of the congestion control algorithm is then evaluated in extended LANs, and wide area networks. The fairness of bandwidth-sharing with other (TCP) traffic is also evaluated.**

I. Introduction

The design of a congestion control mechanism for a multicast transport protocol is quite complex.

Effective congestion control relies on accurate and timely feedback on the prevalent network condition. With more than one receiver, multicast first presents a challenge in how to economically, accurately and speedily collect feedback information.

Since a badly designed (or malfunctioning) algorithm used by multicast can cause more damage to the network during congestion, it is very important for multicast congestion control to be very robust, and to demonstrate it will share network resources fairly with other traffic. Yet, what constitutes a reasonable policy for allocating congested resources that are shared by both unicast and multicast traffic is a hard question.

In this paper, we report our experiences in designing a congestion control[1] algorithm for a tree-based reliable multicast transport protocol. Such a protocol uses a repair tree to localize retransmission of lost packets. We exploit the unique property offered by this repair tree - *regular* acknowledgements flowing from every receiver to the sender, and the feedback information being *aggregated* by the interior nodes of the repair tree - thus ensuring economically and accurately collecting feedback information. We leverage on this infrastructure to implement a primarily window-based congestion control algorithm because it has been demonstrated to be the most robust in our experiments. To compensate for the lack of timely feedback,[2] the sender uses a rate-based traffic shaper to smooth the transmission. One of the key contributions of this paper is the study of how to dynamically adjust the sender's rate based on the congestion window states. We also report our implementation experiences and experimental results.

[1]Actually, we prefer the term *flow control*, as the control is on-going whether there is congestion or not; and irrespective of whether congestion is in the network or at a particular end host, the same control is used.

[2]Multicast acknowledgements are sent once for a window of packets.

II. Previous Work

By far the majority of traffic on the Internet is transported by TCP. The TCP congestion control algorithm [1] has also been a hot topic of research in the literature for many years. In setting a guideline for requirements of multicast congestion control, the Internet Engineering Task Force (IETF) required any standard multicast congestion control to be *TCP-friendly* [2].

The TCP congestion control algorithm is a distributed control algorithm. Each sender uses the loss of packets as a negative feedback signal. Upon congestion indication, the controller uses an additive increase and multiplicative decrease algorithm [3] to react to congestion (or the lack thereof) as well as to achieve fair sharing of congested resources with other flows.

Various researchers studied models of TCP and derived a relation between the throughput of a TCP flow as a function of the packet loss rate and the roundtrip time [4], [5]. This became known as the TCP formula. The notion of TCP-friendliness, therefore, can be loosely described as the behavior of a congestion control algorithm that yields the same throughput (given a loss rate and round trip time) as the TCP formula.

Several efforts were undertaken to design a TCP-friendly congestion control algorithm for multicast. These efforts can be categorized as follows:

- TCP emulation
- TCP formula-based rate control

A recent TCP emulation algorithm is reported in [6]. The idea is quite simple. A *representative* receiver is selected from the multicast group. This receiver and the sender then run a TCP-like algorithm to control the transmission of packets. As far as the sender and the representative are concerned, they are essentially the same as the sender and receiver of a unicast TCP flow. The rest of the multicast group is merely listening in. The main challenge of this approach is to select the *right* representative, which should be one that is using the most congested resources in the network. Arguably, if the point of congestion moves as the multicast session goes on, then it is continuously necessary for a new representative to be selected and the control algorithm to be moved to the new representative.

Earlier, [7] also explored how to emulate the TCP windowing mechanism. Instead of relying on feedback from a representative as in [6], the sender in [7] reacts to receiver feedback signals probabilistically. In other words, each receiver would be acting as the representative some of the time, probabilistically speaking. It was shown that this approach can achieve a relaxed notion of fairness. We found the discussion of relative TCP-friendliness very interesting. However, without using a representative, or a feedback tree, their scheme does not seem easily scalable.

Another TCP emulation protocol is MTCP [8] which uses a feedback tree. Their approach is based on a TCP-like mechanism to adjust the congestion window at each receiver. Further, each receiver estimates the number of packets *in-flight* from the sender. These two pieces of information are aggregated (the minimum value of the congestion window and the maximum value of the in-flight packets) and reported back to the sender, and the sender uses the difference of these two numbers to limit the number of new packets it sends. This approach seems rather complicated.

The TCP formula-based control algorithm takes a different approach than doing incremental adjustments as in a TCP controller. Instead, the controller collects information about packet loss rates and roundtrip times in the background. As soon as these measurements are considered credible, they are plugged into the TCP formula. The derived throughput is then used as the rate to regulate transmission. So this approach necessarily relies on a rate-based control (in contrast to the window-based control used by TCP). The challenge for this approach is how to measure the packet loss rate and roundtrip time. It is proposed that the *right* packet loss rate and roundtrip time to use are those observed by a representative receiver that is behind the most congested resources: in other words, the same representative as the algorithm in [6]. Again, the challenge with this algorithm is how to dynamically keep focusing on the right representative so as to collect the right packet loss rate and roundtrip time information. The studies of this approach have been reported by several researchers, for example see [9], [10] and [11].

An important theoretical discussion of the different mechanisms for congestion control (rate or window), and the different notions of fairness of resource allocation can be found in [12]. In the same paper, Golestani also describes an interesting algorithm for doing window-based flow control. In TCP, the window is controlled at the sender, based on feedback from the receiver. Golestani argues that the control of the window can be moved to the receiver side without changing the effectiveness of the control, in the unicast case. In the multicast case, however, he demonstrates that there is an important difference between implementing the window at the receiver side versus the sender side. Effectively, implementing the window at the receiver side achieves the same behavior as if the sender is controlled by the worst-case representative as discussed above, whereas implementing the window at the sender side would be equivalent to using the worst packet lost rate and the worst roundtrip time. In the latter case, the achieved throughput would be much worse than the former case when the worst packet losses and worst roundtrip time do not occur along the same path to a single representative.

III. Our Algorithm

A. Rationale

The congestion control algorithm described in this paper has been implemented as part of TRAM [13], a tree-based reliable multicast transport protocol.

TRAM adopts a primarily window-based congestion control scheme. A window is simply a range in the packet sequence space. We use the term *left edge* and *right edge* to refer to the lower and higher ends of the window. The sender is allowed to send only those packets in the current window. As packets are received and acknowledged by receivers, the left and right edges of the window are both advanced. This is why a window-based control is also known as a *sliding window* control. The movement of the *left edge* is governed by the acknowledgements (of consecutively received packets); whereas the movement of the *right edge* is governed by the window size. If the window size is constant, then the left and right edges will be maintaining a constant distance. In a window-based congestion control scheme, the window size is dynamically adjusted according to the prevailing network (congestion) conditions. When the sender finishes sending the packet at the window's right edge before an acknowledgement arrives to slide the window, the sender is temporarily held back from sending more packets. We call this condition *window closed.*

TCP's congestion control algorithm is a well-known window-based algorithm. The window-based algorithm in TRAM is different in the following ways.

In a unicast protocol (such as TCP), there is no significant difference whether the sender or the receiver adjusts the window size. In TCP, the sender controls the window size. In a multicast protocol, as explained by [12], it is best to let the receivers maintain the window size. This allows each receiver to maintain a separate window size depending on the network condition on the path from the sender to that receiver. Each receiver tells the sender its "right edge" of the window (piggybacked on acknowledgement packets). The sender then takes the smallest value (of "right edges" from all receivers) and uses that as the right edge of its window. This is the first difference.

In a unicast protocol, it is acceptable to send an acknowledgement for each data packet. For TCP, an acknowledgement is sent for every two data packets (usually). Such frequent acknowledgements allow the sender and receiver to be closely synchronized with the network conditions, and slide the right edge of the window very smoothly (by one or two packets at a time). When the control is operating perfectly, the window is barely open and each new acknowledgement allows and triggers the sender to send a new packet. This is described as *self clocking*.

In a multicast scenario, the overhead of acknowledgement is multiplied by the number of receivers. Many multicast protocols ([14], [15]) try to avoid regular acknowledgements altogether. The more conservative (from a reliability point of view) protocols ([13], [16]) implement regular acknowledgements using a hierarchical tree of repair servers to distribute the processing of the acknowledgements. To limit overhead, each receiver only sends an acknowledgement once per acknowledgement window (typically a relatively large number, e.g., 32 or 64). This means:

- Since the congestion window[3] size must not be smaller than the acknowledgement window size, we are dealing with relatively large congestion windows.
- The large acknowledgement window means the sender is less in synch with the network conditions.
- The movement of the right edge of the window is less smooth (compared to TCP).

Since the sender tends to be allowed to send multiple packets

[3]We now use the term *congestion* window to differentiate it from *acknowledgement* window.

when the window is open, the *self clocking* property is likely lost.

For this reason, TRAM also implements a sender data rate control, as a complement to the window-based control. The sender monitors the prevailing data rate and adjusts it to keep the congestion window open. This data rate is then used to schedule packet transmission. Even if the sender is allowed to send many packets, they are not sent back-to-back, but rather smoothly according to the data rate. Such a packet transmission scheduler restores the "self clocking" mechanism that is lost. See [17] for a discussion on the implementation of the scheduling algorithm.

The data rate monitoring achieves another purpose. There are good reasons for the sender to keep a minimum (R_{min}) data rate. Multicast is group communication. If one receiver is far slower than the others, it is sometimes *right* to remove the slow receiver from the group and let the rest of the group achieve a higher throughput. One simple way to make this tradeoff is to let the multicast session be configured with a minimum data rate (R_{min}). Any receiver that cannot sustain this minimum rate is removed. See [18] for a expanded discussion of this topic.

There are also good reasons for setting a maximum data rate (R_{max}). Many multicast applications care more about network efficiency than throughput. There is often an application-specific maximum desired rate. Setting a maximum data rate not only helps to limit the bandwidth usage by the multicast session, it also serves to limit the instantaneous rate the scheduler will use to send packets. Although TRAM's rate adjustment algorithms help adapt the data rate to a suitable value, a conservative maximum data rate helps TRAM in this respect.

Finally, scheduling packet transmission using a data rate also ensures the retransmissions are transmitted smoothly. Due to the large window used by a multicast transport, sometimes many packets need to be retransmitted, and it is important to control the burstiness of the retransmission traffic.

B. Window Adjustments

Under normal operations, each receiver sends one acknowledgement packet to its parent for every W_a data packets received, where W_a is known as the size of the acknowledgement window.

Each receiver maintains a congestion window, W_c. Initially,

$$W_c = 2W_a$$

The value of W_c is dynamically adjusted, once every W_a packets. The congestion window size for the i^{th} acknowledgement window is denoted $W_c(i)$.

In order to be TCP-friendly, the adjustment follows the *additive increase, multiplicative decrease* rule [3]. If there is no congestion in the i^{th} acknowledgement window then,

$$W_c(i+1) = W_c(i) + 2$$

On the other hand, if congestion was detected during the i^{th} acknowledgement window, then,

$$W_c(i+1) = 0.75W_c(i)$$

The value of 2 and 0.75 are the additive and multiplicative components of the algorithm. The value of W_c is always bounded by the following range:

$$W_a \leq W_c \leq MW_a$$

where M is a configuration parameter called the congestion window multiplier. The value of M is 2 or larger.[4]

The definition of congestion during an acknowledgement window is when there are equal or more lost packets in the current window compared to the last, and the loss level is at least $L_{th}W_a$, where the loss threshold is $L_{th} = 0.25$. The motivation for using a loss threshold greater than 0 is to not let occasional and random losses affect the congestion window.

The receiver-maintained congestion window can be thought of as a credit system. Each receiver issues credit to the sender based on the network (and local) conditions. In good times, you want to keep extending the credit. It is good practice, however, to put some upper limit on W_c, for the same reason you would on a credit system. If W_c is too large when an abrupt congestion condition occurs, the multicast group is exposed to a vast amount of repair in the aftermath. In TRAM, the maximum value of W_c is set to $5W_a$.

C. Feedback and Aggregation

Each time a receiver sends an acknowledgement, it includes the following values:

- The highest consecutively received packet sequence number, H_r
- The highest allowed sequence number, H_a

This is essentially a representation of the receiver's congestion window.

TRAM uses a repair tree to localize repairs. This repair tree is also used to aggregate feedback from the receivers to the sender. At each level in the tree, the receiver sends an acknowledgement containing its own H_r, but the minimum H_a based on all the values of H_a from the subtree below it. In other words, H_r propagates up one level only (for the purpose of reliability) but H_a gets aggregated and propagates all the way to the sender (the root of the tree). Therefore, the sender's value of H_a is the minimum of all the values of H_a in the whole tree.

Ideally, this minimization is done at the same time over the whole tree. In practice, however, the minimization is done in a distributed fashion and over a period of time depending on how each receiver (including the repair heads) schedules its acknowledgements.

To expedite the sliding of the congestion window, especially in the case when the window is held back by one (or a few) very slow receiver, sometimes acknowledgements can be triggered before their scheduled time. Each time the value H_a is updated (normally after receipt of a retransmission or a new acknowledgement packet from a child) to H'_a, the following condition is checked:

$$(H'_a - H_a) > \frac{W_a}{2}$$

If true, an acknowledgement packet is sent immediately.

The above condition is a compromise between optimizing the speed of updating the sender's window and the potential overhead of additional acknowledgement packets.

[4]In TRAM, the default value of M is set to 5

D. Rate Adjustments

The scheduler transmits packets by using the current data rate, R. The sender adjusts R based on monitoring the session.

Rate adjustment is different in the *slow-start* (at the beginning of the session) versus the *steady-state* phases.

In the slow-start phase, the goal is to start from the minimum rate R_{min} and *quickly* find a suitable rate to use. A reasonable approach would be to set

$$\Delta R = \frac{R_{max} - R_{min}}{n}$$

and increase the rate by ΔR after each W_a packets sent until the end of the slow-start. Here n is an integer that reflects the tradeoff between the speed of convergence and potential overshoot, as discussed in [3]. The problem with this approach is when R_{max} is misconfigured (to be exceptionally high), then this initial rate increase may overshoot too much.

To guard against inappropriately configured R_{max}, we choose to make ΔR independent of R_{max}, but adapt gradually from a small minimum value. Initially, ΔR is set to a constant, 2500B/sec. This value also serves as the minimum value of ΔR (denoted ΔR_{min}) throughout the session.

During slow-start, the following rate adjustments are made after each packet transmission:

$$R = R + \Delta R$$

$$\Delta R = \Delta R + r$$

$$r = 1KB/sec$$

The value of R is kept no higher than R_{max}, and the value of ΔR is kept no higher than $\frac{R_{max}-R_{min}}{4}$ at all times during the session. These successive increases are continued until the end of slow-start.

The end of slow-start is reached when one of the following conditions is true:

- R reaches maximum data rate R_{max};
- a congestion report is received for the first time;
- the window is closed for the first time.

At this point, we enter the steady-state phase.

In the steady state phase, the goal is to transmit a sequence of packets in the queue (as window opens) in a steady rate commensurate with the current window size, and to help the window size converge to a steady operating range. The algorithm for rate adjustment depends on monitoring the following parameters:

- the achieved data rate (throughput) during the recent past (denoted $\overline{R}$);
- the average congestion window size[5] during the last W_a packets, (denoted $\overline{W_c}$);
- whether the window is open or not.

The value of $\overline{R}$ is calculated once per W_a packets. Let y be the total amount of data transmitted in the last T seconds, then

$$\overline{R} = \frac{y}{T}$$

where T (5 seconds) is a constant, known as the *time window*.

[5]as seen by the sender

Let N denote the sequence number of the next packet to send, and $H_a(N)$ be the highest allowed sequence number when packet N is sent. For each W_a packets, we compute

$$\overline{W_c}(i) = \frac{\sum_{j=(i-1)W_a+1}^{iW_a}(H_a(j) - j)}{W_a}$$

This is the sender's estimate of the average window size for the i^{th} acknowledgement window of packets.

After every W_a packets, the sender resets the current data rate R and ΔR based on the above monitored state:

1. If the window is open, and opening wider (i.e. $\overline{W_c}(i) > \overline{W_c}(i-1)$), then[6]
$$R = \overline{R} + \Delta R$$
2. For all the other cases[7],
$$R = \overline{R}$$
3. The value of ΔR is reset to
$$\Delta R = max(0.15 * \overline{R}, \Delta R_{min})$$

Note, the value of R is adjusted once per W_a packets. During the course of transmitting these W_a packets, the sender can also monitor the direction in which the window is changing. If the window is gradually closing, it helps to further smooth out the transmission by taking that into account. For this purpose, the actual rate (call it R_s) used for scheduling transmission is modified (before its use) as:

$$R_s = \begin{cases} R_{min} + w(R - R_{min})/W_a & \text{if } w < W_a \\ R & \text{if } w >= W_a \end{cases}$$

where w is the sender estimated window size at the moment, i.e. $(H_a - N)$.

To summarize, the above rate adjustment algorithms might seem complicated, but they are designed to be adaptive and robust. There are three algorithms:

- slow start: Try to quickly reach a rate in the right ball-park. The strategy is to increase the rate exponentially (starting from the minimum) until the onset of congestion.
- window-based rate adjustment: Once per acknowledgement window of packets, reset the rate to the monitored average rate achieved in the recent past, with one exception. When the average congestion window size is growing wider, it signals a likely climate for higher possible throughput. In this case, the rate is set to the average achieved rate plus an increment.
- fine tuning: Before using the rate for scheduling each packet, a separate scheduling rate is derived and used if the congestion window is observed to be close to shutting down.

The single goal is to compute a rate that is commensurate with the current congestion window, and use it to smooth the transmission of successive packets. These algorithms are rather similar to how a production line is operated based on inventory, or how a budget is managed based on the level of consumption by an organization.

[6]In our original design, the new rate is derived from the current rate rather then the current average rate, namely $R = R + \Delta R$. This achieved higher throughput performance with a small number of receivers. However, we noticed the use of $\overline{R}$ produced a much smoother rate of transmission in the steady state, and resulted in better performance as the number of receivers increased. This point will be elaborated on further in the section where we discuss experimental results.

[7]In an earlier design, we chose to set R to a fraction of $\overline{R}$ after the window closes. We notice, however, since we use a slower scheduling rate as the window draws to a close, the average rate $\overline{R}$ is already reduced sufficiently.

E. Packet Scheduling

Packets are placed on either the retransmit queue or the data queue. A scheduler takes packets off the queues and sends them. Packets are first taken from the retransmit queue.

The scheduler tries to smooth the transmission of packets based on the data rate described in the last section. Retransmitted packets are sent at the open-window rate. Data packets are sent at the open-window rate if the window is open; otherwise they are sent at a rate of 1 packet per second.[8]

Given a data rate, transmission scheduling is implemented by calling a sleep(t_{sleep}) function after each packet transmission. The input argument, t_{sleep}, is the *sleep time*, computed as

$$t_{sleep} = \frac{P}{R_s}$$

where P is the packet size and R_s is the scheduling rate as defined in the last section.

There are two interesting details related to the granularity of the sleep() function.

E.1 Sleep time is an integer

Unless a *real time* platform is used, the sleep() function can handle sleep times of only a moderate granularity. On the platform we experimented with, t_{sleep} is an integer in milliseconds. The above computation of t_{sleep} is rounded down to be an integer.

When the current data rate, R_s, is sufficiently high relative to the packet size, the resultant sleep time t_{sleep} becomes zero (after rounding). This means packets will be transmitted back-to-back without gaps, and hence yield an effective data rate potentially higher than that prescribed.

Given the granularity of one millisecond and a packet size of P, the data rate must be in the following range

$$R_s \leq 1000P$$

for the scheduling to be accurate. For example, for packet size of 1500 bytes, R_s should be no more than 1.5 MB/sec.

This problem can be alleviated to some extent by monitoring the amount by which the actual data throughput exceeds the prescribed rate and compensating for that, provided the prescribed rate does not keep changing.

E.2 Oversleeping

Another problem with the sleep() function is that it may sleep more than t_{sleep}. On the platform we experimented with, the actual time slept is t_{sleep} rounded up to the next multiple of ten milliseconds. For example, sleep(12) would result in a pause of 20 milliseconds, and sleep(23) would result in 30 milliseconds, and so forth.

Oversleeping causes the resultant data throughput to be lower[9] than the prescribed data rate, R_s. This problem is described in detail in [17] and a number of algorithms to compensate for oversleeping have been suggested.

[8]In this sense, the 1 packet per second rate is the *instantaneous* minimum data rate for a TRAM multicast session. The configuration parameter, R_{min}, is used as a threshold; when the instantaneous rate becomes consistently lower than this threshold, slow receivers are pruned to restore a reasonable session rate.

[9]sometimes significantly lower

F. Pruning

While the maximum data rate, R_{max}, is used to limit burstiness (in the worst case), the minimum data rate, R_{min} is used to guarantee some level of data throughput. When there is a large receiver group, it is increasingly likely that some receiver is much slower than others and thus slowing down the data rate of the whole group of receivers. We use R_{min} as a yardstick to select the receivers that are too slow and *prune* them.[10]

Isolating those receivers that ought to be pruned is not easy. Since all receivers receive at the same rate at which the sender is sending, it is not possible to determine which receiver(s) caused the sender to slow down by measuring each receiver's receive rate (since they would all be the same). This is how pruning is tied into congestion control. The congestion control mechanism slows down the data throughput of the multicast session to below R_{min}, when there are receivers who cannot sustain a receive rate of R_{min}. When the sender detects the session throughput is below R_{min}, all the repair heads in TRAM collectively identify those receivers that caused this condition - not by measuring each receiver's receive rate, but by noticing those receivers with higher loss rates and which have fallen behind in sending acknowledgements. To avoid pruning borderline receivers by mistake, the repair heads use a distributed algorithm to determine the slowest receivers and prune them one at a time until the congestion control algorithm restores the session data rate above R_{min} again. [18] contains a detailed study of this topic.

IV. Experiments

In this section, we describe a set of experiments we used to test the performance of TRAM and hence its congestion control algorithm. We are interested in how fast it runs when all the hosts are on the same LAN, and how it scales in this environment. We then study how bandwidth limitations between the receivers and sender affect the performance. Finally, we try to characterize how TRAM and TCP traffic share limited network bandwidth.

All these experiments were run using a Java implementation of TRAM that is publicly available [19].

Table 1 contains the values of some of the TRAM configuration parameters used in our experiments, unless noted explicitly later. The acknowledgement window (W_a) was set to 64 which yielded slightly better performance.[11] The congestion window multiplier, M, was set to 5 (default). The minimum data rate (R_{min}) is deliberately set to a very low value (1KB/sec) so that there will be no pruning. The maximum data rate (R_{max}) is deliberately set to a very high value (1.5MB/sec) so that the maximum speed of TRAM can be tested. Each repair head is limited to have no more than 5 members in its repair group. A relatively small Maximum Members helps ensure that no repair head becomes a bottleneck.

A. The Effect of Increasing Population Size

The goal of our first experiment is to see how fast TRAM can run and how well it scales when the number of receivers increases.

[10]When a receiver is pruned, it means that that receiver no longer gets reliability service from the transport. That receiver may still listen to the multicast.

[11]The default value of W_a in TRAM is 32.

TABLE I

PERFORMANCE-SENSITIVE CONFIGURATION PARAMETERS

parameter	default value
ACK Window	64 packets
Congestion Window Multiplier	5
Minimum Data Rate	1 KBytes/sec
Maximum Data Rate	1.5 MBytes/sec
Maximum Members per Repair Head	5

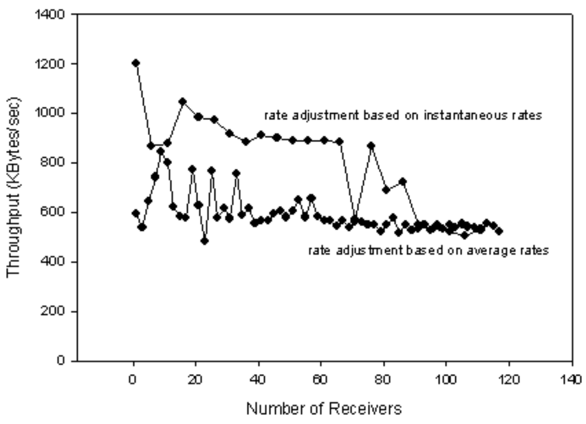

Fig. 1. LAN Measurements

This is done in a high-speed LAN environment where the network is not the bottleneck. All the measurements reported in this section were performed at the Rochester Institute of Technology. The test programs are publicly available[20].

A sender sends (using TRAM) 100,000 packets of size 1400 bytes to n receivers. The same test is repeated for increasing numbers of receivers, n. At the end of each test, the number of receivers still alive is verified to be the same as the number that started the test. The throughput is plotted against the number of receivers in Figure 1.

There are two curves in Figure 1. In one case, the rate adjustment algorithm (when the congestion window is opening wider) is based on using the average rate measured for every W_a, as described in section 3.4. The other case is based on an earlier design where the current rate is directly incremented. The earlier design actually performed better for smaller numbers of receivers. However, a detailed examination of the dynamics of the sessions revealed that the use of the average rate produced much smoother transmission. It is also more robust as we change other configuration parameters, such as the congestion window multiplier. Furthermore, it competes more fairly with other traffic. For these reasons, we decided to give up a little performance and adopted the algorithm based on using average rates for adjustments.

The throughput remained quite high up to the highest number of receivers available in the testbed, which is around 120. The testbed is not a controlled environment, so there is some variability in the results, especially for higher throughput values. But this test gives us a rough idea of how fast the protocol can run and how it scales up.

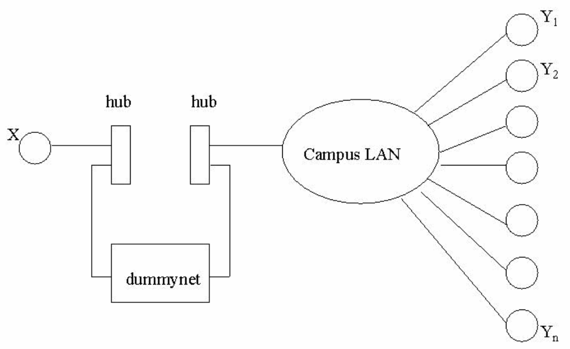

Fig. 2. Dummynet test environment

B. The Effect of Limited Network Bandwidth

For various reasons, controlled multicast tests in a wide-area network are difficult to set up. So we chose to use Dummynet[21] to emulate a network with limited bandwidth. We found Dummynet a great alternative to simulation, because it allowed us to exercise a real implementation of our algorithm.

The measurement network is as shown in Figure 2. Dummynet runs on a PC. We verified that the Dummynet (PC) can easily emulate bandwidth up to 400 KB/sec.[12] The Dummynet PC comes with two Ethernet interfaces. One interface was connected (via a hub) to a dedicated UNIX machine, X; and the other interface was connected (via a hub) to the campus highspeed LAN through which we can reach many dedicated UNIX machines $(Y_1, Y_2, ..., Y_n)$. Although this is not a completely isolated test environment due to the campus LAN, we carefully chose to run our experiments during times there was little or no traffic on the campus LAN.

In these experiments, the multicast sender is always running on machine X, and the one receiver each on Y_i. The only traffic going through Dummynet is a single multicast session. We tested a multicast session with 1, 7 and 14 receivers. For each case, the bandwidth emulated by Dummynet is varied from 50KB/sec up to 400KB/sec. The results are shown in Figure 3.

At low bandwidths, the multicast throughput matches quite closely with the available bandwidth, independent of the number of receivers.

As the number of receivers increases, TRAM's achieved throughput begins to fall short of the available bandwidth. Since all the receiver machines are connected to a 10Mbit/sec Ethernet, one explanation is the effect of collisions. When we checked the collision counters at the network interfaces of these machines, we did notice a significant amount of collisions.

C. TCP-friendliness

In this section, we evaluate how TRAM shares network bandwidth with TCP traffic. We did not try to study fairness at the

[12]We did not try to find out what the maximum bandwidth this PC running Dummynet could emulate.

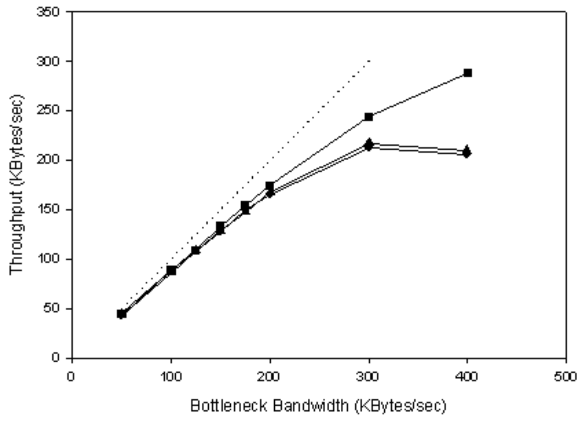

Fig. 3. TRAM performance in a network of limited bandwidth

packet-by-packet scheduling level, nor are we advocating a particular metric for comparing traffic fairness. Our goal is to study traffic sharing at a more macroscopic level where it is only important to compare throughput achieved for sending a fixed amount of data in a session.

For this study, we felt it adequate to run each TCP session as a file transfer using FTP. We first run a set of tests to see how multiple simultaneous FTP sessions share a given configured Dummynet bandwidth (in this case 400KB/sec). We increase the simultaneous FTP sessions from 1 to 7. The result is shown in the second and third columns of Table 2. The second column shows the average throughput for each FTP session, whereas the third column gives the range of values. It is clear that these FTP sessions were able to use up all the limited bandwidth, and share them roughly equally.

When comparing TRAM's throughput and TCP's throughput when they are running simultaneously, we took the following strategy:
1. First measure the throughput of FTP sessions (each transferring 10MB of data), in the middle of a TRAM session (transferring 30MB of data).
2. Then measure the throughput of a TRAM session (transferring 30MB of data), in the middle of a set of simultaneous FTP sessions (transferring 50MB of data).
The first set of tests tell us how TCP shares bandwidth with a *background* TRAM session; and the second set of tests tell us how TRAM shares bandwidth with a background set of FTP sessions. Unless otherwise specified, all TRAM sessions below have 7 receivers.

The FTP sessions' throughputs (with TRAM running in the background) are shown in columns 4 and 5 of Table 2. The TRAM sessions' throughputs (with FTP sessions[13] running in the background) are shown in column 6 of Table 2. All throughput numbers are in KB/sec. Again, the FTP sessions' throughput are reported as an average as well as a range.

These results show TRAM to be less *friendly* than TCP. When there are only a small number of FTP sessions, they are able to grab the bandwidth that TRAM is not using (even by itself, TRAM does not use up all the bandwidth). As we increase the

[13]These FTP sessions are started roughly simultaneously.

TABLE II
PERFORMANCE OF FTP SESSIONS WITH OR WITHOUT TRAM (M=5) IN THE BACKGROUND

Num of FTPs	FTP thruput Without TRAM		FTP thruput With TRAM		TRAM thruput with FTP
1	374	(374, 374)	185	(185, 185)	179
2	188	(188, 188)	99	(99, 100)	158
3	126	(126, 127)	76	(74, 78)	149
4	99	(95, 105)	65	(62, 67)	134
5	78	(75, 81)	48	(47, 48)	119
6	65	(62, 71)	44	(42, 45)	112
7	56	(53, 59)	41	(38, 46)	103

TABLE III
PERFORMANCE OF FTP SESSIONS WITH OR WITHOUT TRAM (M=2) IN THE BACKGROUND

Num of FTPs	FTP thruput With TRAM in background		TRAM thruput with FTP in background
1	174	(174, 174)	182
2	113	(113, 113)	122
3	86	(81, 90)	96
4	65	(65, 67)	70
5	60	(57, 61)	65
6	53	(50, 58)	61
7	44	(43, 47)	60

number of simultaneous FTP sessions, TRAM gives up some of its bandwidth to the FTP sessions but tends to use a bigger share than the FTP sessions (which do share the rest of the bandwidth more or less fairly).

This is not entirely surprising. The basic part of congestion control that tries to be friendly with TCP is the congestion window adjustment algorithm. There are several parameters in this algorithm that are not the same as TCP. For example, the additive increase value is 2; and the multiplicative decrease fraction is 0.75. Perhaps more significantly, for ACK efficiency reasons, TRAM uses much bigger congestion windows. The configurable parameter, congestion window multiplier (M) can be used to control how wide the congestion window can open. To optimize TRAM's own performance, M's default value is set to 5. Reducing the value of M should help make TRAM more friendly to other traffic.

We rerun the above tests setting M to 2. The results are shown in Table 3. It confirms our theory.

TRAM also allows users to control its bandwidth usage by setting the Maximum Data Rate to an appropriate value.

Finally, we captured some data to show the progress of the TRAM sessions during the above experiments. We wanted to ensure that TRAM was making smooth adjustments in the face of competing traffic (FTP sessions). These data are plotted as the cumulative data transferred against time for each TRAM session.

Figure 4 shows TRAM's progress when different numbers of

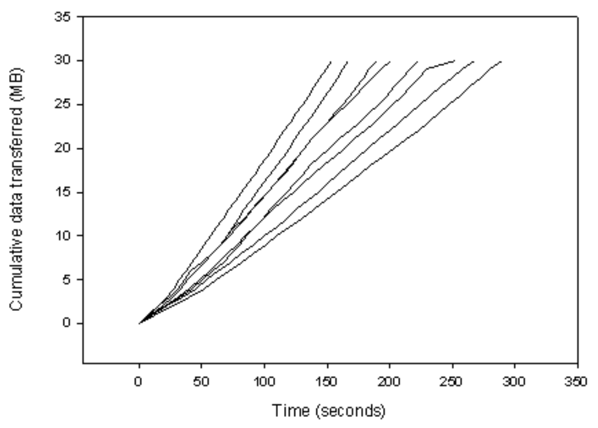

Fig. 4. TRAM's progress with FTP sessions in the background

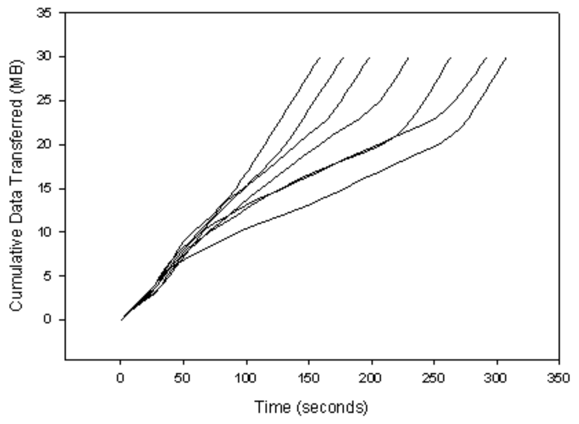

Fig. 5. TRAM's progress in the background, with FTP sessions starting in the middle

FTP sessions are running in the background. The 8 curves correspond to the cases with 0 to 7 simultaneous FTP sessions running in the background. As the number of background FTP sessions increases, the progress curve results in a more gradual slope (indicating slower rate of data transfer). These curves are as expected from the results in Table 2 and 3. The relative smoothness of the curves indicates that the TRAM session ran steadily with varying amounts of competing traffic.

Figure 5 shows TRAM's progress when it is running in the background, and a number of FTP sessions are started during the TRAM session. The seven curves correspond to the number of FTP sessions ranging from 1 to 7. In each curve, the slope starts at a sharper angle, then drops down for some time, and then resumes at a sharper angle. As the number of FTP sessions increases, the decrease in the slope becomes more significant and for a longer duration.

V. Conclusions and Future Work

In this paper, we describe a congestion control algorithm suitable for a reliable multicast transport with regular feedback from all receivers. It is similar to other proposals of adapting sliding window flow control to multicast by implementing the windowing algorithm at the receivers and retaining the TCP-like window adjustments. Our contribution is in describing how to dynamically adjust the data rate used to schedule packet transmission at the sender to smooth the transmission. These adjustments are designed to probe the network when the congestion window is opening wider, and take various precautions as the window is closing.

We describe some lessons learned in implementing the packet scheduling algorithm in the transport. When the scheduling is not done in the kernel, there are some subtle problems due to the coarse granularity of the sleep() function. Since maintaining a transmission rate is part of the algorithm, this is an integral part of the congestion control implementation.

We also describe an algorithm called pruning to remove group members which are too slow for the good of the whole receiver group. We explain how pruning is tied to, yet does not interfere with the congestion control algorithm.

The performance of TRAM has been extensively tested in LAN environments. The maximum throughput is close to 1 MBytes/sec. A throughput of greater than 500KBytes/sec can be achieved for up to 120 receivers.

We used Dummynet to emulate a wide-area environment with limited bandwidth to study TRAM's behavior and how it shares traffic with TCP. The throughputs of TRAM and TCP were measured for a varying number of TCP sessions sharing the bottleneck. The results show TRAM has good stability and fairness properties. Certain configuration parameters can be set to achieve good TCP-friendliness.

Our results demonstrate that TRAM's congestion control system (several different algorithms) is sound - it delivers good performance, stability in the face of congestion, and fairness.

As the study of TCP congestion control proves, the incorporation of automatic traffic management algorithms into a protocol is a very complicated problem. Despite our numerous tests, there are many facets of the proposed congestion control algorithm that can profit from further investgation. In particular, the various parameters in our algorithm can certainly be subjected to more sensitivity studies. Our approach in this study is largely experimental. It is very desirable to develop an abstract model that can approximately predict throughput based on receiver population size and other key parameters of the control algorithm.

VI. Acknowledgements

We thank Phil Rosenzweig for starting the reliable multicast project; Steve Hurst who worked with us during the early part of the project; Phillip Conrad for suggesting that we use Dummynet; Luigi Rizzo for providing assistance in getting Dummynet to work; and Jeanie Treichel for providing editorial comments.

References

[1] V. Jacobson, "Congestion avoidance and control," in *ACM Sigcomm88*, August 1988.
[2] A. Mankin, A. Romanow, S. Bradner, and V. Paxson, "IETF criteria for evaluating reliable multicast transport and application protocols," *IETF RFC2357*, June 1998.
[3] D. M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, vol. 17, no. 1, pp. 1–14, 1989.
[4] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the tcp congestion avoidance algorithm," *ACM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, July 1997.

[5] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling tcp throughput: A simple model and its empirical validation," in *ACM Sigcomm98*, September 1998.
[6] L. Rizzo, "A tcp-friendly single-rate multicast congestion control scheme," in *ACM Sigcomm00*, August 2000.
[7] H. A. Wang and M. Schwartz, "Achieving bounded fairness for multicast and tcp traffic in the internet," in *ACM Sigcomm98*, September 1998.
[8] I. Rhee, N. Ballaguru, and G. Rouskas, "MTCP: Scalable tcp-like congestion control for reliable multicast," Tech. Rep., North Carolina State University, Department of Computer Science, TR-98-01, January 1998.
[9] M. Handley and S. Floyd, "Strawman specification for tcp friendly (reliable) multicast congestion control (tfmcc)," Tech. Rep., Reliable Multicast Research Group working document, December 1998.
[10] S. Floyd, M. Handley, and J. Padhye, "Equation-based congestion control for unicast applications," in *ACM Sigcomm00*, August 2000.
[11] B. Whetten and J. Conlan, "A rate based congestion control scheme for reliable multicast," Tech. Rep., Reliable Multicast Research Group working document, December 1998.
[12] S. J. Golestani and K. Sabnani, "Fundamental observations on multicast congestion control in the internet," in *IEEE Infocom99*, March 1999.
[13] D. M. Chiu, S. Hurst, M. Kadansky, and J. Wesley, "TRAM: A tree-based reliable multicast protocol," Tech. Rep., Sun Microsystems Laboratories Technical Report (TR-98-66), 1998.
[14] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking*, November 1996.
[15] Byers et al, "A digital fountain approach to reliable distribution of bulk data," in *ACM Sigcomm98*, September 1998.
[16] J. Lin and S. Paul, "RMTP: A reliable multicast transport protocol," in *IEEE Infocom95*, 1995.
[17] D. M. Chiu, M. Kadansky, J. Provino, and J. Wesley, "Experiences in designing a traffic shaper," in *Fifth IEEE Symposium on Computer and Communication*, July 2000.
[18] D. M. Chiu, M. Kadansky, J. Provino, J. Wesley, and H. Zhu, "Pruning algorithms for multicast flow control," in *NGC 2000*, November 2000.
[19] "Java reliable multicast service," http://experimentalstuff.com/Technologies/JRMS.
[20] "Rochester Inst. of Tech. test programs," http://www.cs.rit.edu/∼hpb/JRMS.
[21] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computing Communication Review*, January 1997.