Strawman Specification for TCP Friendly (Reliable) Multicast Congestion Control (TFMCC)

Mark Handley, Sally Floyd
ACIRI
mjh@aciri.org/floyd@aciri.org

Brian Whetten
Talarian Corporation
whetten@talarian.com

June 2, 1999

1 Introduction

This document presents a *strawman* specification for TCP-Friendly Multicast Congestion Control (TFMCC). It is not a standard of any form. We do not suggest basing products on this specification. Eventually, if further research shows this specification to perform acceptably, we envisage moving this document into the IETF for standardization, but the document is not currently at this stage. Many people have contributed to this specification in one form or another - by the time it is ready for standardization we hope that many more people will have contributed ideas and performed simulation work to test the details.

This specification proposes a set of basic algorithms for a reliable multicast rate controller, which can work with a wide range of NACK-based and Hierarchical ACK-based protocols. In addition, it proposes a set of NACK-based algorithms, which provide solutions for some of the problems that are dependent on the feedback mechanisms used in the protocol. It also proposes a set of HACK-based algorithms, which work with hierarchical reliable multicast protocols. These extensions provide solutions for some of the problems that are feedback-dependent, and help eliminate the drop to zero problem that the basic algorithms face. While this document discusses this congestion control mechanism in the context of reliable multicast applications, the same congestion control mechanisms could be used for unreliable multicast as well.

The basic algorithms include the following:

- A response function that provides a reasonably accurate model of TCP
- A mechanism for measuring loss rates at receivers
- A policy for the rate at which a sender can increase/decrease its throughput

The NACK-based algorithm include a mechanism for NACK suppression from the receivers, an algorithm for measuring RTT at the receivers, and an algorithm for handling senders which go quiescent.

Similarly, the HACK-based algorithms include an algorithm for measuring RTT at the receivers and an algorithm for handling senders which go quiescent, along with mechanisms for dealing with late joining receivers and control node failures, and a distributed algorithm for computing the variables needed for the congestion control calculations.

The requirements that a reliable multicast congestion control scheme must meet, in order to be considered for widespread Internet deployment, include a wide adaptive range, fairness and stability:

A Wide Adaptive Range. The congestion control scheme must have a wide adaptive range, in terms of the realistic range of values for throughput achieved by the congestion control scheme. As discussed in [WC98], the adaptive range can be a function both of the loss rate and of the roundtrip time.

Fairness. The congestion control scheme should be provably fair relative to TCP, under steady state analysis. This is primarily a function of the response function chosen, as well as of the way values for P and RTT are chosen.

Stability. The implementation mechanisms should handle a wide range of dynamic changes, over a wide variety of network infrastructures, in a way that does not cause unsafe oscillating behavior. In particular, it should not be vulnerable to self-reinforcing feedback loops if it becomes the dominant mechanism for the traffic on a given network segment.

The additional objectives are strongly recommended, both in order to provide added network safety, and in order that the scheme provides enough benefits to its users to make it attractive when compared to other alternatives such as replicated TCP.

Responsiveness. The scheme should adapt to changes in the network load as rapidly as is feasible.

Discontinuous Sender or Receiver Behavior. When a sender starts up, a sender temporarily goes quiescent, or a new receiver joins the group, the algorithm should adapt to this in a way that is responsive, fair, and stable.

Performance. The system should allow performance at least equal to the alternative of replicated TCP. In particular, it should avoid the drop to zero problem, so that performance does not decrease as the number of receivers increases.

1.1 Assumptions

We make the following assumptions about the application in use:

- The application can tolerate unbounded variation in the time to transmit a fixed amount of data.
- The data flow is one-to-many. Some many-to-many applications may be viewed of consisting of multiple independent one-to-many flows, and so this specification may apply to them too, but we shall not address them explicitly here.
- The data flow is continuous. It does not stop because the application temporarily runs out of data to send.[1]

[1] We specify how transmission may restart after a sender goes quiescent, but we assume that when an application sends, it does so for a considerable length of time.

- The data flow is large - the duration of the flow is expected to be a large number of network round trip times.
- The desired behaviour is for the sender to transmit at the short-term mean rate that would be achieved by a TCP flow from the sender to the slowest of the receivers.
- It is acceptable for a transport protocol to respond to packet losses on a slower timescale than the one-RTT timescale that TCP uses.

All of these conditions should be true for an application to use TFMCC. It may be possible to modify TFMCC to cope with applications that do not fit these assumptions. For example, a unicast feedback mechanism may be substituted for the multicast mechanism if alternative means are devised to prevent response implosion. We do not deal with such modifications in this document. Companion documents may do so in the future.

The application we have in mind is a bulk-data transfer application, where a large amount of data must be distributed to a large number of sites. However, other applications may also satisfy these assumptions, including unreliable multicast applications. This specification does not address reliability - it is up to the reliable multicast protocol to achieve this within the bandwidth bounds provided by TFMCC.

2 Basic Algorithms

2.1 The Response Function

TFMCC is based around modeling the long term behaviour of TCP, and using such a model to achieve similar throughput for a multicast flow. There are several models of TCP long-term behaviour . The range of bandwidths that TFMCC is applicable for depends on the choice of model. Of the existing published models[3][2][1], our simulations show that only the one by Padhye et al[1] is suitable for general purpose use. The other models do not account for TCP retransmission timeouts, and fail to throttle bandwidth appropriately at loss rates above about 5%.

We choose the approximate solution from [1] and ignore the term due to receiver window limiting - this is a self-imposed handicap unrelated to network congestion, so we can safely ignore it. We can also remove the min term for the sake of simiplicity - the effect of this is only to make the RM protocol slightly less aggressive at loss rates above 30%. The equation given is:

$$B(l) = \frac{s}{t_{RTT}\sqrt{\frac{2bl}{3}} + t_0 min\left(1, 3\sqrt{\frac{3bl}{8}}\right) l(1+32l^2)} \qquad (1)$$

l is the loss fraction, $B(l)$ is the data rate in bytes/sec, and s is the packet size in bytes. In addition, b is the number of packets acknowledged by each TCP ACK, t_0 is the TCP retransmission timeout value, and t_{RTT} is the round trip time from sender to receiver and back.

When applying this equation for multicast congestion control, we face several challenges:

- Choosing a suitable number of transmitted packets to sample the loss rate over.

- Communicating the loss fraction from the receiver that would have the lowest TCP rate in a timely manner without causing a response implosion at the sender.
- Discovering the RTT to the receiver that would have the lowest TCP rate.
- Approximating t_0 at the receiver that would have the lowest TCP rate.

To send at the correct rate, the sender must continuously know the values of $B(l)$ (or l, t_{RTT} and t_0) at all receivers and not just the value of l. This is because the receiver with the highest loss fraction is not necessarily the receiver that would achieve the lowest rate with TCP because there may be a receiver with a slightly smaller loss rate and a much higher RTT.

In the general case, this cannot be done without causing a response implosion, although it may be possible for some small groups using ACK-based protocols. Thus we specify ways to approximate the behaviour that would be obtained with the full information using only incomplete information. Clearly protocols that can provide more complete information than we assume can be achieve better approximations.

Note that later versions of this specification may substitute different functions for $B(l)$ that remedy deficiencies of the Padhye equation when RED gateways are employed and when modeling TCP-SACK. These revised functions should be able to be plugged into this specification without requiring further changes and result in slight improvements to performance.

2.2 Calculating loss fraction at the receiver

In every data packet, the sender transmits the current data rate it is transmitting at, the estimate of worst case t_{RTT}, the timestamp when it sent the packet, and two sequence numbers, MSEQ and DSEQ. MSEQ is the message sequence number which is increased by one for every data packet sent regardless of whether it is a retransmission, and DSEQ which uniquely identifies the data packet itself for the purposes of retransmission requests. We do not specify the form that DSEQ takes. Some protocols may be be able to use MSEQ only.

The receiver keeps track of the MSEQ space maintaining information about which packets were received and which were lost. For now, we assume that it keeps an infinite history, but will address how to make this finite later.

The receiver calculates the inverse of the function $B(l)$ using the rate parameter supplied by the sender and it's locally calculated t_{RTT} and t_0 values (see below). This gives the fraction of packets, l_{exp}, that would be expected to be lost to give this data rate. The value k/l_{exp} then gives us the number of packets of history to examine to determine the loss fraction. k is a constant value that is used to trade off responsiveness against sampling accuracy. We recommend a default value of $k = 4$, but this should be the subject of future study.

To calculate the loss fraction we count loss events rather than lost packets. Thus lost packets are ignored if they are within one RTT of a lost packet that was not ignored. This models TCP-SACK behaviour where only one backoff is performed per RTT. If a receiver has an estimate of its own RTT it uses that. If not, it uses the RTT advertised by the sender.

If k/l_{exp} is greater than the length of the history, the length of the history should be used instead. The receiver should not report any losses until at least two loss events have been observed.

If k/l_{exp} does not include several loss events, the behaviour may be unstable and cause oscillation. Thus if k/l_{exp} does not include E loss events, we calculate the loss fraction by including all the packets back to the E^{th} most recent loss event or the beginning of the history if E loss events have not occurred. We recommend a default value of $E = 4$ to be used based on provisional simulation results, but this should be the subject of future study.

2.3 Calculating the RTT

Ideally we'd like each receiver to be able to calculate or approximate the RTT to the sender. Mechanisms for calculating RTT are dependent on the feedback mechanism available, and so are discussed in the context of a particular feedback mechanism below.

2.4 Estimating t_0

t_0 is the TCP retransmission timeout value. In TCP it is normally calculated as:

$$t_0 = t_{srtt} + 2\ t_{rttvar}$$

where t_{srtt} is the smoothed RTT estimate and t_{rttvar} is the variance of the RTT. Both are calculated using EWMA filters from the current sampled RTT, $t_{sampleRTT}$, by:

$$t_{srtt,i+1} = 7/8\ t_{srtt,i} + 1/8\ t_{sampleRTT}$$

$$t_{rttvar,i+1} = 3/4\ t_{rttvar,i} + |t_{sampleRTT} - t_{srtt,i}|$$

However, whereas the RTTs for TFMCC are measured as accurately as possible, in TCP, $t_{sampleRTT}$ is often sampled with course clock granularity. This varies from TCP to TCP, but 500ms is not uncommon. In such systems the minimum value for t_0 is 500ms, not matter what the real RTT is.

Trying to model TCP accurately in this respect is impossible without an RTT measurement for every few data packets from each receiver. This clearly does not scale past very small receiver groups. Also there's a difficult decision of which TCP to model - those with poor clock granularity or those with fine clock granularity. There is no clearly right answer to this question.

In the absence of any better model for TCP that can be scalably deployed in a multicast congestion control mechanism, we recommend:

$$t_0 = Max[20ms, 4 * t_{RTT}]$$

The RTT is the minimum plausible value for t_0. The term $20ms$ models the limitations imposed by a TCP with a moderate clock granularity. [WC98] discusses one justification for the term $4 * t_{RTT}$.

2.5 Increase/Decrease Mechanism

Each receiver calculates the loss fraction, l, it's own RTT, t_{RTT}, and it's own value for t_0 as indicated above. Using this information, the receiver calculates $B(L)$ using 1. The value of B from the worse rate receiver must be conveyed back to the sender in a timely fashion. How this is done depends on the precise details of the protocol under consideration, and different feedback mechanisms are discussed below.

At the sender, the values of the worst case B arrive at a rate of at least the minimum of the data send rate and once per RTT. Whenever one of these reports arrives, the sender compares this ideal sending rate with it's current sending rate B_{actual}, and adjusts the sending rate as follows:

If $B_{report} < B_{actual}$, the current sending rate is reduced to R_{report}.

If $B_{report} > B_{actual}$, the current sending rate is increased once per RTT by $\alpha * B_{actual}$.

The constant α is in the range 0 to 1. The actual value for α still needs to be determined, but a value of approximately 0.1 should work reasonably.

2.6 Policy-Based Throughput Constraints

Two policy-based parameters that could be used to characterize a session include a Minimum Throughput and a Maximum Throughput. The Minimum Throughput would be the smallest data rate that the sender would use. For a session with a Minimum Throughput, receivers unable to keep up with that sending rate would have to unsubscribe from the multicast group. In addition, some mechanism would have to be in place to prevent the session from continuously transmitting on a link due to a succession of receivers subscribing and immediately unsubscribing due to the high packet drop rate.

A receiver can detect whether or not it is below the Minimum Throughput if it knows its measured loss rate and its RTT to the sender, along with the session parameters of the current sending rate and packet size.

2.7 Flow Control

The job of flow control is to ensure that a sender does not overrun its receivers. The flow control implemented in an ACK-based protocol will be different from that in a NACK-based protocol. Because this document limits itself to issues relating to congestion control, this document does not discuss flow-control in detail.

3 ACK Feedback Mechanism

In a small-scale ACK-based protocol where every few data packets elicit an ACK from every receiver, the calculated value of $B(l)$ and the echo of the senders timestamp should be reported by all receivers in every ACK packet.

4 NACK or Supressed Feedback Mechanisms

In NACK-based protocols with large receiver sets, it is usually infeasible for all receivers to send a report every few packets.

Where a NACK-based protocol uses on-demand packet-level FEC, there will typically be a NACK-suppression mechanism that ensures that the receiver missing the most packets responds indicating how many packets are missing. Such a response would then normally also carry the feedback listed above. However, this is not sufficient to perform congestion control, because the receiver with the highest loss rate may not be the receiver with the lowest value of $B(l)$. Thus a similar suppression mechanism needs to be run in parallel to the loss feedback mechanism so that the receiver with the lowest value of $B(l)$ also sends feedback. We do not recommend a particular loss suppression scheme for this case because a simple variant of the existing scheme for loss reports is probably more appropriate.

For protocols without an existing suppressed feedback mechanism on which to piggy-back congestion feedback, we suggest the mechanism below.

4.1 RTT Calculation

With larger numbers of receivers, this is not possible to achieve without synchronized clocks at the sender and receivers. We STRONGLY RECOMMEND that reliable multicast senders either have an external clock source such as GPS or use NTP to obtain a good clock reference.

If the sender knows unambiguously that it has a clock synchronized using NTP or an external source such as GPS, it indicates this in the data packets along with the sender timestamp. Any receiver that knows unambiguously that it has a clock similarly synchronized SHOULD use its local clock to calculate the one-way propagation delay and double this to approximate the RTT. If the resulting value is less than 100ms, assume 100ms is the RTT until the RTT can actually be measured, because NTP cannot be relied upon to synchronize to very small absolute differences on machines with poor inbuilt clocks. We RECOMMEND that multicast receivers have a clock synchronised to an accurate external time source or use NTP to obtain a good clock reference.

If the sender does not have or does not know it has a synchronized clock, or if a receiver does not have a synchronized clock, the congestion control scheme can still function, but it will be more conservative in its estimates of RTT. Such a receiver should initially assume that the RTT is 500ms. Although the real RTT may be greater than this, it will not be greater by more than an order of magnitude on links that do not leave the Earth/Moon system. We do NOT RECOMMEND using this mechanism for interplanetary communication.

Assuming that we have a baseline approximation for the RTT at a receiver, either the default value above, or one obtained from synchronized clocks, then the receiver uses this value to calculate the $B(l)$. As we indicate below, the receiver with the lowest value of $B(l)$ will report this value along with the arrival time of a recent data packet, the sender timestamp from that packet, and the time the response was sent.

The sender calculates the real RTT from this information, and reports this RTT and the receiver's address in a later data packet. When the receiver receives this data packet, it updates its estimate of RTT, t_{RTT} from the measured RTT, t_{sample}, using the EWMA filter:

If $i = 0$
$t_{RTT,i+1} = t_{sample}$
Else
$t_{RTT,i+1} = 0.25 t_{sample} + 0.75 t_{RTT,i}$

If the receiver never sends a report because its calculated rate $B(l)$ does not require it to do so, it would otherwise not be able to reduce its RTT value. If the real RTT to all receivers were significantly less than 500ms, it would require all receivers to exchange packets with the sender before they could all reduce their calculation of RTT and hence allow the data rate to increase.

To allow the RTT estimates to decay in such cases, we allow the sender to report an estimate of the worst RTT for the whole group based on its sampling of the RTT. Receivers without synchronized clocks that have not exchanged any report packets with the sender may use this value of RTT if the sender reduces it from the initial 500ms value. Receivers with synchronized clocks that have not exchanged any report packets with the sender SHOULD still use their own estimate of the RTT.

Several methods also exist by which a receiver may estimate its RTT to the sender without exchanging messages directly with the sender.

4.1.1 Third-Party RTT Estimation Using Synchronized Receiver Clocks

If the sender does not have an externally synchronized clock, but two receivers R_1 and R_2 do have such clocks, then if R_1 measures its RTT to the receiver using a feedback message and timestamps its multicast feedback message, then R_2 may be able to estimate its own RTT to the sender. If the sender sent the packet at time t_s, R_1 received the packet at time t_r, and the measured RTT from the sender to R_1 and back to the sender is t_RTT1, then R_2 can now estimate R_1's clock offset from the sender's clock to be $t_r - (t_s + t_{RTT1}/2)$. Depending on how accurately R_2 believes its clock is synchronized to R_1, this gives it a good measurement of the one-way propagation delay from the sender to R_2. The RTT is estimated by doubling the one-way delay.

If R_1 and R_2 are synchronised by GPS, they may assume their clocks are synchronized to 1ms and R_2 may use the estimate directly. However, if they are synchronized by NTP, the RTT estimate should be used only if it exceeds 100ms, otherwise 100ms should be used as the RTT measurement.

Estimating the offset from RTT information is clearly assuming that the reverse path and the forward path are similar. If the forward path from the sender to R_1 is much different (either longer or shorter) than the reverse path, then the estimate of clock offset will be incorrect by half the difference of the two paths. Thus if R_2 derives a one-way delay value of less than $t_{RTT1}/2$ it should use t_RTT1 as its RTT measurement instead of twice the one-way delay value.

Estimating the RTT from one-way information is unlikely to be a problem so long as the forward (data) path is the one we are principally concerned with.

We are clearly attempting to be conservative here. If the estimate is too low, a receiver may never measure its own RTT, but suffer significant congestion. However, if the estimate is too high, a real measurement will be made if this receiver calculates the lowest value of $B(l)$.

As the senders clock may drift, the estimate of the sender's clock offset should not be used after a long period of time. However, third-party estimation in this manner can significantly speed up the initial convergence of TFMCC.

4.1.2 Third-Party RTT Estimation Using Hierarchy

Estimates of the RTT can also be made by exchanging messages with a nearby receiver which has itself exchanged a message with the sender. Many different ways exist to do this. Some RM protocols may already include such hierarchies to produce scalable session messages or loss feedback, and we encourage such protocols to make use of their hierarchy to scalably estimate RTTs for receivers that have not exchanged feedback with the sender.

An example of a scalable hierarchical control message scheme that can be applied to a non-hierarchical RM protocol to estimate RTT is found in [4].

4.2 NACK Suppression

This area requires significant further experimentation to show that it works OK, and to fine tune the mechanism.

The basic feedback suppression mechanism relies on multicast responses from one receiver suppressing responses from other receivers that have higher values of $B(l)$. Every so often, as indicated by a flag set in a data packet, every receiver sets a feedback timer to a value calculated randomly as below. If the timer expires, that receiver sends its feedback packet. If, however, it hears another receiver with a lower value of $B(l)$ reporting for the same sender timestamp, the receiver cancels the feedback timer. Note that several feedback timers may be running simultaneously triggered by different flagged data packets. Suppression only affects the one timer associated with the same reported sender timestamp.

As we do not know how many receivers received high loss in a reporting interval, we need to use an exponentially weighted random timer. We additionally bias this timer by the bandwidth rate being reported. If the senders current reported sending rate is B_{send}, and the loss rate a receiver wishes to report is $B(l)$, then we set the feedback timer, t_{delay}, to:

$$t_{delay} = D_1 + Rlog_2(2^{D_2/R} * X + 1)$$

Where R is the *senders* current reported value for worst case RTT, X is a random number chosen from the uniform random interval [0:1], and D_1 and D_2 are determined by:

$$D_1 = TBD$$

$$D_2 = TBD$$

We envisage some function of $B(l)/B_{send}$, perhaps weighted by receiver set size will be used to determine D_1 and D_2, but are reluctant to propose precise functions without further simulation or analysis.

5 Hierarchical ACK Feedback Mechanisms

The previous section proposed additional algorithms to be used for reliable multicast protocols that do not include any positive acknowledgements. This section assumes the existence of a steady stream of hierarchical, positive acknowledgements - HACKs. These HACKs can be infrequent, and combined with NACKs if desired. For purposes of illustration, it uses the terminology of RMTP-II in discussing the hierarchy, but could work with a variety of reliable multicast protocols that employ HACKs.

It provides the following algorithms:

1. A scalable algorithm for measuring RTT to each receiver.
2. An algorithm for slow start, to be used when a sender is starting, or has gone quiescent for an extended period of time.
3. An algorithm for decaying the target rate of a sender when it goes quiescent for a period of time.
4. An algorithm for handling receivers that join a stream when it is already in progress.

5. An algorithm for controlling the sender rate in the face of failures of interior control nodes in the tree.

6. A scaleable algorithm for implementing restricted worst edge calculations, to counter the drop-to-zero problem.

5.1 Measuring RTT at the Receivers

Calculating the RTT from the sender to each receiver and designated receiver is a significant challenge for a large scale reliable multicast protocol. One solution for this problem has been proposed in RMTP-II [WBPT98]. This protocol provides a hierarchical way to calculate RTT to each receiver and interior node. It does so by continually calculating, using an exponentially weighted moving average, the RTT between each child and parent in a tree. This includes the RTT between the sender and the top node, between the top node and each of its children, and between each designated receiver and each of its children. These calculations are done as part of the periodic heartbeat messages sent from each parent to each of its children, and in the heartbeat response messages from the children to their parent.

This approach creates separate RTT estimates for each child and interior node in the tree, in a completely scalable fashion. The primary drawbacks of this approach are:

1. The estimates are not made for each packet sent. Rather, estimates occur at a fixed rate, typically once every 1 to 10 seconds. This corresponds to roughly the same length of time that it takes to get an accurate measurement of loss, so appears to be adequate.

2. This method assumes rough tree congruence between the topology of the transport layer and the topology of the network layer. For cases where this assumption does not hold, RMTP-II provides a second mechanism for computing the worst case RTT. It does so by measuring the time from when a data packet is sent to when a HACK for that data packet is received, and subtracting the time each level in the tree spent delaying the packet. This method can produce a measurement as often as every single HACK, and works even for asymmetrical networks.

Recent work [5][6] proposed a new approach for efficiently calculating RTT values in a hierarchical tree. This approach appears to mitigate or solve the above problems, and provides a possibly superior alternative to calculating RTT values in a HACK tree.

5.2 Slow Start Algorithm

When a connection first starts up, the goal is to increase the rate of transmission quickly, to the appropriate order of magnitude, without dramatically overshooting the available bandwidth. In a rate based scheme this is particularly difficult, because the averages used for measuring loss rates and RTT times are inaccurate at startup. Also, if the loss rates at each receiver are more or less independent, then the number of packets sent before a loss is experienced is inversely proportional to the number of receivers. To get around these problems, the following algorithm is used at startup. Part of this algorithm is borrowed, with modifications, from [**?**].

When the system starts up, the sender starts sending at an initial minimum rate, and increases this rate exponentially as ACKs are received, similar to TCP slow start. As soon as a receiver has experienced two loss events, it generates its first loss report, as described in section 2, and feeds its throughput estimate back to the sender. Slow start continues until the first time that the computed target rate is smaller than the actual rate ($B_{report} < B_{actual}$) after which the system reduces B_{actual} to B_{report}, and then uses the normal algorithms proposed in section 2.

The algorithm only performs a multiplicative rate increase when it gets positive confirmation back from receivers that a packet has been successfully received. These HACKs take longer to return than a single RTT (potentially by an order of magnitude), and so this is a more conservative increase policy than TCP uses. For the cases where the HACKs take much longer to return than one RTT, the steady state increase/decrease policy, specified above, is used as a lower bound on the rate of increase.

For a sender who has received a total of NumACKs acknowledgements from its receivers, its throughput T is,

$$T = \frac{NumACKs(c) * PacketSize(c)}{InitRTT(c)} + Max\left(MinimumThroughput(c), \frac{PacketSize(c)}{InitRTT(c)}\right)$$

where $MinimumThroughput$ is the optional minimum throughput policy constraint, and $InitRTT$ is an initial, conservative worst case estimate of group RTT. $InitRTT$ can either be given a fixed value (we recommend 500ms) or can be calculated as part of each initial HACK. This second option is preferred, and is supported in RMTP-II.

5.3 Quiescent Senders

A sender will likely continue to receive feedback for some time after it goes quiescent, and so can continue updating its target throughput. Doing so is also necessary, in order to track dramatic changes in RTT, such as from a new receiver joining (see below).

A sender is defined as quiescent if the application does not have any data for it to send. If a sender goes quiescent for more than a minimum period of time, $MinimumQuiescent$, it should start reducing its actual allowed throughput rate B_{actual} (which the rate controller uses), at the same rate as B_{actual} is normally increased when B_{report} is larger than B_{actual}. If there is a single other sender sharing a congested link with the sender, and its feedback path takes approximately the same amount of time as $MinimumQuiescent$, then this sender will "consume" the bandwidth of the link at this rate, so that if the first sender starts up again, the system will still be in equilibrium. If there are more senders sharing the link, then the new sender will create some level of additional congestion when it restarts, because the cumulative bandwidth "consumption" of the senders will be more than the backoff by the quiescent sender. However, TCP adapts much faster than this policy, and so in most cases, the TCP connections sharing the link will rapidly consume the available bandwidth, and then rapidly give it back up when the sender goes active again.

While this policy is network friendly, it risks heavily penalizing bursty reliable multicast connections. This can be limited by capping the decrease of B_{actual} so that it is only reduced if the following holds true: $B_{actual} * MaximumQuiescentDecrease > B_{report}$, where a value of 4 is recommended for $MaximumQuiescentDecrea$ In addition, B_{actual} should never drop below the $MinimumThroughput$ policy variable, if used. Since the sender is still receiving periodic RTT feedback, the primary uncertainty involves what the loss rate could have changed to. For small initial values of P, a 4x change in throughput corresponds to a 16x change in loss rate. This confidence decreases at higher loss rates (such as 4x at 2.5% loss rate), but a system is less likely to have its loss rate dramatically increase when if it is already at a high loss rate.

If a sender reaches the limit imposed by $MaximumQuiescentDecrease$, then when it starts sending again, it should use Slow Start to again find its appropriate initial rate.

5.4 Late Joining Receivers

When a receiver joins a stream already in progress, the new receiver could be at the end of a much worse connection than any of the other receivers. A receiver measures its RTT before joining, and also initializes its own loss rate with the worst loss rate reported by any receiver. After joining, the receiver reports its new RTT and loss rate in the first HACK it generates. If the receiver's connection is so bad that it can not sustain the minimum throughput of the connection, it will shortly realize this, and disconnect. This could cause a short burst of overly high network congestion, but this is unavoidable without requiring the sender to slow-start every time a new receiver joins.

5.5 Control Node Failure

When any node in the tree detects a control node failure, it should send notification of this to the sender, by piggybacking the information on to a HACK. When the sender receives this notification, it must reduce its send rate to the minimum rate for the group. When the failure has been recovered from, the reliable multicast protocol must have a way of signaling this to the sender, at which point it enters the slow start algorithm, and increases up to equilibrium again.

5.5.1 Restricted Worst Edge Calculations

This document proposes the use of the Restricted Worst Edge mechanism [**?**] for selecting the values of RTT and P for the response function. This method takes advantage of intermediate nodes in a hierarchical tree, and works so long as the protocol is not being run on a fundamentally asymmetrical network, such as satellite down, terrestrial return. In those cases, Worst Path should be used instead.

In Restricted Worst Edge, the cumulative loss rate of the child is used (similar to TCP and Worst Path), combined with the local RTT for an edge. An edge is defined as the network path between any child node and its parent in the hierarchical tree. In addition, the Efficiency of the tree must be calculated, which is the amount of resources saved by using multicast instead of replicated unicast. The target throughput calculated for Restricted Worst Edge is bounded by the throughput calculated for Worst Edge, multiplied by Efficiency. This keeps small groups of just a few senders, with multiple DRs, from being unfair to TCP. In practice, this will likely be a rare occurrence, and may not need to be implemented.

When doing aggregation at a Designated Receiver or Top Node, it is important to deal with the local recovery problem that can occur if the amount of retransmissions is larger than called for by the value of P used in calculating the TargetRate. For sake of simplicity, we assume that this pathology is handled through the use of integrated FEC, as in RMTP-II.

The sender for a connection requires information about congestion on the network in order to determine the optimum transmission rate. The required information is collected from the receivers and intervening nodes, aggregated to the top node of the tree, and forwarded to the sender. The next section offers a scaleable way to implement both Worst Path and Restricted Worst Edge calculations, as well as windowed flow control. Because the calculations are aggregated at each interior node, they scale extremely well.

5.6 Hierarchical Calculations

This section shows how to compute flow control, Worst Path, and Restricted Worst Edge, in a scaleable way, by piggybacking the necessary variables on top of HACKs, and aggregating the variables as the packets are sent up the tree.

5.6.1 Flow Control

In order to compute flow control, the sender needs to get the highest flow control sequence number for any receiver. This indicates the highest sequence number all receivers are able to receive, given their current buffer status.

Receiver: A receiver sets *AvailableWindow* to the maximum sequence number that it has buffer space for.

Aggregation: Each intermediate node, i, takes the minimum of the values for each direct descendant, d, using sequence number arithmetic.

$AvailableWindow(i) = Min(AvailableWindow(d) : d$ is a child of $i)$

5.6.2 Worst Path

In order to compute Worst Path, the sender needs to get the computed throughput of the slowest receiver, based on that receiver's RTT and P measurements.

Receiver: A receiver computes its own*WorstReceiverThroughput*, by evaluating the response function with its own values for P and RTT.

Aggregation: Each intermediate node, i, takes the maximum of the values for each direct descendant, d.

$WorstReceiverThroughput(i) = Min(WorstReceiverThroughput(d) : d$ is a child of $i)$

5.6.3 Restricted Worst Edge

With Restricted Worst Edge, the sender needs to calculate the minimum throughput for the worst edge in the tree, using the RTT for that edge, and the worst loss rate of any child that uses that edge. This requires calculating the local RTT value for each edge, the worst loss rate for any receiver, WorstReceiverLossRate, and the WorstReceiverThroughput. We assume that the local RTT is already measured for each local edge between an interior node, i, and its child d. This is denoted as ChildRTT(d).

In addition, the sender needs to find the Efficiency of the tree, which requires computing the Receiver-Count, UnicastCost and the MulticastCost of the tree. Unlike the other variables, these two only need to be computed when the membership of a group changes.

WorstReceiverLossRate This is the worst loss rate of any receiver in a subtree.

Receiver: A receiver sets this to its most recent loss rate, measured using the algorithms specified in the second section.

Aggregation: Each intermediate node, i, takes the maximum of the values for each direct descendant, d.

$WorstReceiverLossRate(i) = Max(WorstReceiverLossRate(d) : d \text{ is a child of } i)$

WorstEdgeThroughput This is the value computed for Restricted Worst Edge, without the Efficiency bound.

Receiver: A receiver initializes WorstEdgeThroughput to infinity.

Aggregation: Each interior node, i, calculates the ChildThroughput to each of its children, d, by evaluating the response function, T_{RM} with the local ChildRTT to that child, and the WorstReceiverLossRate reported from that child. It then reports the smallest throughput, over any of the calculated ChildThroughput values and reported WorstEdgeThroughput values.

$ChildThroughput(i, d) = min(WorstEdgeThroughPut(d), T_{RM}(PacketSize(c), ChildRTT(d) - RTT(c, i), T', WorstReceiverLossRate(d)))$

$WorstEdgeThroughput(i) = Max(ChildThroughput(i, d) : d \text{ is a child of } i)$

ReceiverCount This is the number of receivers that are descendants of a given node.

Receiver: A receiver initializes this to one.

Aggregation: Each intermediate node, i, sums the value of ReceiverCount for all its direct descendants, d.

$ReceiverCount(i) = \sum_{d \text{ is a child of } i} ReceiverCount(d)$

UnicastCost This is the cost to send a packet using replicated unicast, where the cost is measured by the RTT to each receiver.

Receiver: A receiver initializes this to 0.

Aggregation: Each intermediate node, i, sums the value received from each of its direct descendants, d, with the values it calculates locally for that descendant. The value to each descendant is the RTT of the edge to that descendant times the value of ReceiverCount for that descendant.

$ChildUnicastCost(i, d) = ReceiverCount(d) * (ChildRTT(d) - RTT(c, i)) + UnicastCost(d)$

$UnicastCost(i) = \sum_{d \text{ is a child of } i} UnicastCost(i, d)$

MulticastCost This is the cost to send a packet to all receivers using multicast, where the cost is measured by the RTT of each link in the tree.

Receiver: A receiver initializes this to 0.

Aggregation: Each intermediate node, i, sums the value received from each of its direct descendants, d, with the values it calculates locally as the RTT to each descendant.

$ChildMulticastCost(i, d) = ChildRTT(d) - RTT(c, i) + MulticastCost$

$MulticastCost(i) = \sum_{d \text{ is a child of } i} MulticastCost(i, d)$

Sender Calculations When the sender, s, receives the above variables, it acts as the last intermediate node, and performs the above calculations one final time. The sender then computes the Restricted Worst Edge target throughput using the below equations, and applies the dynamic algorithms from the rest of the paper, to compute the final throughput for the sender.

For Restricted Worst Edge calculations, the sender uses WorstEdgeThroughput, unless it does not pass the efficiency test [WC98]. The target rate is bounded by the WorstReceiverThroughput (from the Worst Edge calculations above) times the Efficiency of the connection.

$Efficiency = \frac{UnicastCost(s)}{MulticastCost(s)}$
$TargetRate = Min(WorstReceiverThoughput(s) * Efficiency, WorstEdgeThroughput(s))$

6 Discontinuous Behavior

When a sender starts up, a sender temporarily goes quiescent, or a new receiver joins the group, the algorithm should adapt to this in a way that is both responsive, fair, safe, and stable. This is a primary challenge for the mechanisms chosen to implement the algorithms, and requires additional analysis and simulations. The key design decisions that affect this are:

Slow Start Mechanism. When a sender starts up, or goes quiescent for an extended period of time, it must probe the network to find the correct sending rate. Ideally, this is done at a faster rate of increase than the normal increase/decrease policy. However, without positive acknowledgements from each receiver, it is much more difficult for the algorithm to do so and still be safe and stable.

Sender Quiescence. When a sender stops sending for a period of time, it loses confidence that it knows the appropriate rate at which to send data in to the network. Consequently, its allowed transmission rate should decay over time, if it does not have data to send. If this decay is larger than a given threshold, then on resumption of data transmission, it should use the slow start mechanism, if any.

Late Joining Receivers.

When a receiver joins a stream already in progress, it may be the new bottleneck for the group. In order to be safe, the mechanisms should attempt to discover if the receiver's target throughput rate is dramatically worse than that of the other receivers, and adapt to it if so.

Control Node Failures. Some protocols, such as RMTP-II [WBPM98], have interior control nodes that are used for local recovery and ACK-aggregation. If one of these nodes fails, care needs to be taken that the sender throttles its send rate while the receivers are recovering from this disruption. After recovery is complete, slow start can be used to increase the rate again.

7 Messages

We cannot give packet formats in this specification because TFMCC is intended to be integrated into a variety of multicast protocols. However, most protocols are likely to require a similar set of information.

7.1 Sender Messages

Piggy-backed on all (or many) of the senders data packets will be the following information:

Report Flag This indicates that receivers are to set a response timer to make a congestion control feedback report. It is used to synchronize reports so that feedback suppression happens correctly.

NTP Clock Sync Flag This indicates that the sender's clock is synchronized using NTP.

External Clock Sync Flag This indicates that the sender's clock is synchronized using an external clock source accurate to within 1ms.

Send timestamp The absolute time the packet was sent measured in units of milliseconds.

Worst Case RTT The senders current estimate of near-worst case RTT measured in units of milliseconds.

RTT Measurements A list of pairs of IP addresses and the RTTs measured to those hosts using receiver feedback.

Not all of this information is required in every data packet, but if the report flag is set, the send timestamp must be present. Also the worst case RTT should be sent at least once per worst-case RTT.

All data packets must contain a strictly monotonically increasing message sequence number (MSEQ) which must not wrap for significantly longer than the longest message history that will be needed.

7.2 Receiver Feedback Messages

Receiver feedback may be piggy-backed on other control messages so long as this does not adversely affect the timeliness of the reporting or feedback suppression.

Each feedback message contains the following information:

B(1) The value of bandwidth calculated by the receiver from its loss record and RTT measurements.

Sender Timestamp The sender timestamp of the data packet triggering this feedback message.

Receive Timestamp The time, measured at the receiver, that the data packet triggering this message arrived.

Feedback Timestamp The time, measured at the receiver, that this feedback message was sent.

NTP Clock Sync Flag This indicates that the receiver's clock is synchronized using NTP.

External Clock Sync Flag This indicates that the receiver's clock is synchronized using an external clock source accurate to within 1ms.

See section 4.1.1 for an explanation of the purpose of these flags and why we prefer two absolute timestamps to a single difference of the two.

8 Simulation results

Preliminary simulation results of the use of TCP throughput equations for *unicast* congestion control can be found at: http://north.east.isi.edu/ mjh/results.ps

9 Acknowledgments

This specification is based on the ideas of many people, including Allison Mankin, Jim Conlan, Steve Deering, Jon Crowcroft, Jean Bolot, J. Padhye, V. Firoiu, Don Towsley, and Jim Kurose and other members of the Reliable Multicast Research Group. Our apologies to those we've accidentally omitted.

References

[1] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation", Proc. ACM Sigcomm, 1998, Vancouver, Canada.

[2] J. Mahdavi, S. Floyd, "TCP-Friendly Unicast Rate-Based Flow Control", Technical note sent to the end2end-interest mailing list, 1997. http://www.psc.edu/networking/papers

[3] M. Mathis, J. Semke, J. Mahdavi, T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm", ACM Computer Communication Review, 27(3):67-82, Jul. 1997.

[4] P. Sharma, D. Estrin, S. Floyd, "Scalable Session Messages in SRM", http://catarina.usc.edu/estrin/papers/infocom98/ssession.ps

[5] V. Ozdemir, S. Muthukrishnon, I. Rhee, "Scalable, Low-Overhead Network Delay Estimation", presented at the RMRG, December 1998.

[6] Basu, J. Golestani, "Hierarchical Round Trip Time Estimation", presented at the RMRG, December 1998.