ENHANCEMENT OF SINGLE RATE MULTICAST CONGESTION CONTROL POLICY

A dissertation submitted in the partial fulfillment of the requirements for the award of the degree of

MASTER OF ENGINEERING
IN
COMPUTER TECHNOLOGY & APPLICATIONS

by:

AMIT AGGARWAL
College Roll No. 11/CTA/04
(University Roll No. 8501)

Under the Guidance of

Dr. GOLDIE GABRANI
Head of Department (Computer Engineering)

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
UNIVERSITY OF DELHI
JUNE 2006

Certificate

This is to certify that dissertation entitled "**Enhancement of Single Rate Multicast Congestion Control Policy** " which is submitted by **Amit Aggarwal** in partial fulfillment of the requirement for the award of degree M.E. in Computer Technology & Applications to Delhi College of Engineering, Delhi is a record of the candidate work carried out by him.

(Project Guide)
Dr. Goldie Gabrani
Head of the Department
Department of Computer Engineering
Delhi College of Engineering
Bawana Road, Delhi

ACKNOWLEDGEMENTS

It is a great pleasure to have the opportunity to extent my heartiest felt gratitude to everybody who helped me throughout the course of this project.

It is distinct pleasure to express my deep sense of gratitude and indebtedness to my learned supervisor **Prof. Goldie Gabrani** Head of Department (Computer Science), Delhi College of Engineering, for her invaluable guidance, encouragement and patient review. Her continuous inspiration only has made me complete this dissertation. She is the one who kept on boosting me time and again for putting an extra ounce of effort to realize this work.

I would also like to take this opportunity to present my sincere regards to my teachers Prof. **D.Roy Choudhary, Mrs. Rajni Jindal, Dr S. K. Saxena** and **Mr. Rajeev Kumar** for their support and encouragement.

I am grateful to my parents, brother and sister for their moral support all the time, they have been always around on the phone to cheer me up in the odd times of this work.

Amit Aggarwal
M.E. (Computer Technology & Applications)
Department of Computer Engineering
Delhi College of Engineering, Delhi

Table of Contents

Certificate **2**

ACKNOWLEDGEMENTS **3**

List of Figures: **6**

List of Variables: **7**

List of abbreviations **8**

Abstract **9**

1) Introduction **10**

1.1)Brief Description of ESMCC 12

1.2 Key Contributions 15

2) Related Works **16**

2.1 Single-Rate Schemes 16

2.2 Multi-Rate Schemes 19

2.3 Additive increase and multiplicative decrease (AIMD) 20

2.3.1. Definitions 23

2.3.2 Slow Start 24

2.3.3 Fast Retransmit and Fast Recovery 29

3) ESMCC **31**

3.1 ESMCC Receiver 31

3.1.1 Throughput Rate On Congestion (TROC) - 31

3.1.2 Feedback Handler 34

3.1.3 Feedback Suppression 35

3.2 ESMCC Source 36

3.2.1 CR Selection: Tracking of the Slowest Receiver 36

3.2.2 CR Mode Control 41

3.2.3 Rate Adaptation 44

3.2.4 Update of Statistics 45

3.2.5 Data Packet Handler 47

3.3 Key Features of ESMCC 47

4) Simulations and Experiments **48**

5) Conclusion 58
6)References: 59
7) Appendix 61

List of Figures:

Figure 1: Example operation of ESMCC.
Figure 2: Packets in transit during additive increase: add one packet each RTT.
Figure 3: Packets in transit during slow start.
Figure 4: Fast retransmit based on duplicate ACKs.
Figure 5: Operations at ESMCC receiver.
Figure 6: Source Operations as a block diagram.
Figure 7: At every RTT, the source attempts to increase the transmission rate
and updates the operating mode of the source as either CR Active or CR inactive.
Figure 8: Operations that take place when a feedback packet from receiver i arrive at the source.
Figure 9. Sketch of updating congestion Representative (CR).
Figure 10: Handling of data packets at the source: Source keeps attaching
$\overline{\mu}\,\&\,\overline{\sigma}$ to every data packet.
Figure 11: Graph plotted between Rate vs Time for the three nodes.
Figure 12: Graph plotted between RTT update vs Time for the three nodes.
Figure 13: Graph plotted between RTT Deviation vs Time for the three nodes.
Figure 14: Graph plotted between Receiver vs CI suppressed.
Figure 15: Graph plotted between Receiver vs CI sent

List of Variables:

Symbol	***Meaning***
$\mu_i(t)$	Average TROC for receiver i at time *t*
$\Omega_i(t)$	TROC for receiver i at time *t*
$\omega_i(t)$	Instantaneous output rate at receiver i at time *t*
$\sigma_i(t)$	Deviation of TROC at receiver i at time *t*
$\overline{\mu}(t)$	Average TROC for the current CR at time *t*
$\overline{\Omega}(t)$	TROC for the current CR at time *t*
$\overline{\omega}(t)$	Instantaneous output rate at the current CR at time *t*
$\overline{\sigma}(t)$	Deviation of TROC at the current CR at time *t*
$\phi()$	EWMA averaging function
α	Exponential averaging factor
T	Response time of the CR, i.e. elapsed time until the first feedback is received from the current CR when the path to CR is fully loaded
T_{max}	Estimate of the maximum possible *T*
$\lambda(t)$	Source's sending rate at time *t*
s	Data packet size
Δt	Time period over which TROC is measured
RTT_{max}	Maximum RTT observed by the source among all receivers

List of abbreviations

1) ESMCC-----Enhancement of single rate multicast congestion control
2) TROC-------Throughput rate on congestion
3) EWMA----- Exponentially Weighted Moving Average
4) AIMD-------Additive increase multiplicative decrease
5) CR-----------Congestion Representative
6) RTT--------- Round trip time
7) CI------------Congestion Indication
8) MIMD-------Multiplicative increase and Multiplicative decrease
9) CC----------- Congestion Clear

Abstract

Multicast is the preferred transport mechanism for bulk data transfer to multiple receivers especially in multimedia applications and services on the internet. Applications like content distribution, streaming, multi-player games, multimedia multi-user chat/telephony, distance education etc could benefit from multicast and QoS. Multicast congestion control is the first step toward a multicast QoS architecture for the Internet. There are two categories of multicast congestion control. One of them is single-rate, in which the source controls the data transmission rate and all receivers receive data at the same rate. The other is multi-rate (layered multicast congestion control), in which receivers join just enough layers in the form of multicast groups to retrieve data as fast as they can.

This dissertation intended for the policy which is the enhancement of the single-rate multicast congestion control scheme (ESMCC) based on a new metric calculation, Throughput Rate on Congestion. It reduces a memory complexity to maintain state information at source and receivers; requires only simple computations. It addresses the pieces of the single-rate multicast congestion control problem including drop-to-zero issues, TCP friendliness and RTT estimation. It's rate-based on additive increase multiplicative-decrease (AIMD) module and does not necessitate measurement of RTTs from all receivers to the source. It is very effective with feedback suppression.

1) Introduction

The Internet relies on applications performing congestion control to react to network congestion and avoid congestion collapse. Most applications in use on the Internet employ TCP's congestion control algorithms [13].

The increasing popularity of group communication applications such as multi-party teleconferencing tools and information dissemination services had lead to a great deal of interest in the development of multicast transport protocols layered on top of IP multicast. Unlike TCP's point-to-point model, which treats multipoint data delivery as a collection of point-to-point flows thus sending duplicate data repeatedly over the same network links, multicast protocols can greatly improve the efficiency of multipoint data distribution by using a many-to-many delivery model. To allow multicast protocols to be deployed on the Internet, it is imperative that they incorporate mechanisms for handling network congestion. While many proposals have been forthcoming on reliable multicast protocols, few of them have focused on congestion control mechanisms to accompany these protocols.

There are two categories of multicast congestion control. One of them is single-rate, in which the source controls the data transmission rate and all receivers receive data at the same rate. The previous work includes, for example, DeLucia et. aI's work in [1], PGMCC[2], TFMCC[3], MDP-CC[4] and our prior work LESBCC [5]. The other is multi-rate (layered multicast congestion control), in which receivers join just enough layers in the form of multicast groups to retrieve data as fast as they can. The most noticeable among them are recently developed Fine-Grained Layered Multicast [6] and STAIR [7].

Multicast protocols face the feedback implosion problem [14, which becomes critical as multicast group size increases. Several existing reliable multicast transport protocols use probabilistic suppression to limit the amount of feedback received at the source.

The single-rate category is easy to implement and deploy, because it does not require support from intermediate nodes beyond standard multicast capabilities, also does not introduce high processing load to them. Although such schemes do not scale as well as multi-rate ones because they track the slowest receiver, they are suitable for such situations as the multicast in a not-so heterogeneous environment, or bulk data transfer without concerns over delay. With some network support [8], we can also emulate multi-rate schemes by deploying single-rate schemes on selected intermediate nodes.

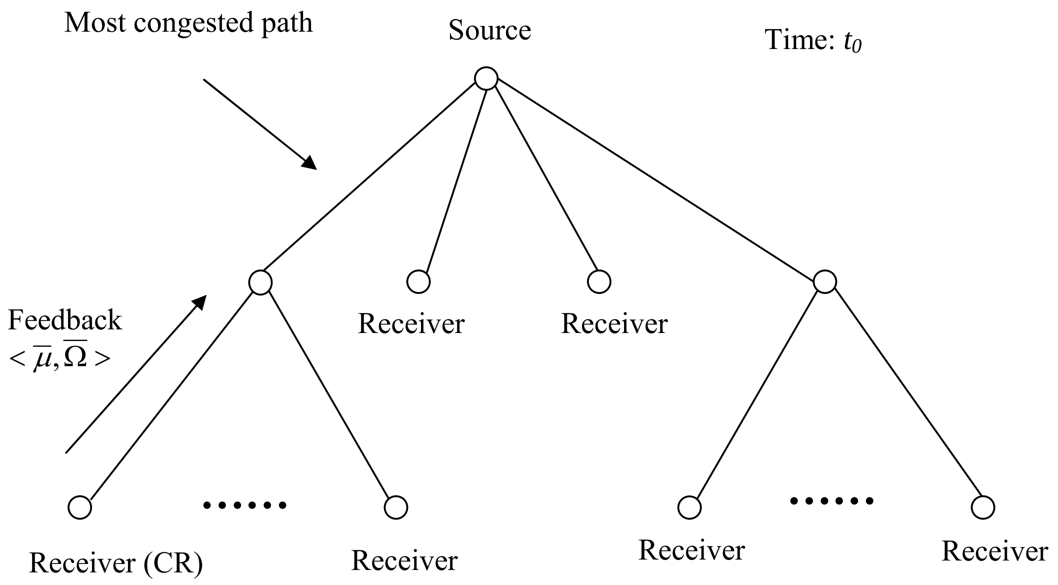
In this paper, we introduce an Enhancement of the Single-rate Multicast Congestion Control scheme (ESMCC) based on a new metric calculation, Throughput Rate on Congestion. We will first very briefly describe ESMCC below, and then in Section 2, we will briefly discuss some related work followed by the ESMCC details in Section 3. Then, we will present simulation and experiment results in Section 4. Finally, we will conclude the paper in Section 5.

1.1)Brief Description of ESMCC

The key idea of ESMCC is to base the scheme on a new metric, *TROC (Throughput Rate on Congestion),* which is the throughput rate measured by receivers when congestion is detected. At the source and receivers, 0(1) state is maintained, and only simple computations are required; there is no need to measure RTTs from all receivers to the source, which can be a tedious problem especially without external instrumentation (e.g. GPS, NTP server), and (iii) we do not make any assumption on network topology and intermediate nodes beyond standard multicast capabilities. It is also *effective* because (1) it successfully addresses the well-known problems of slowest receiver tracking, TCP-friendliness, and drop-to-zero; and (2) the feedback suppression mechanism works very effectively by suppressing over 95% feedback under normal situations.

The general concept of our scheme is as follows: The source dynamically selects one of the slowest receivers as *Congestion Representative (CR),* and only considers its feedback for rate adaptation. The slowest receivers are those with the lowest average TROCs. Each receiver keeps measuring its TROC when it detects congestion and updates its average TROC by means of a smoothing technique such as Exponentially Weighted Moving Average (EWMA). Receivers detect congestion, when they observe a loss in the data packets[1]. The source considers these average TROCs of the slowest receivers in its decision to select the CR. When there is no CR, all receivers may send feedbacks to the source. However, this no-CR situation will last at most one RTT, because the new CR will be chosen in one RTT. This limitation of one RTT time period on no-CR case also prevents any possible Ack implosion. Once a CR is selected, only the CR and those receivers with average TROC lower than that of the CR can send feedbacks so that feedbacks are efficiently suppressed. Also notice that our scheme is not concerned with reliability issue and only considers congestion control. Therefore, it is applicable to both reliable and unreliable multicast.

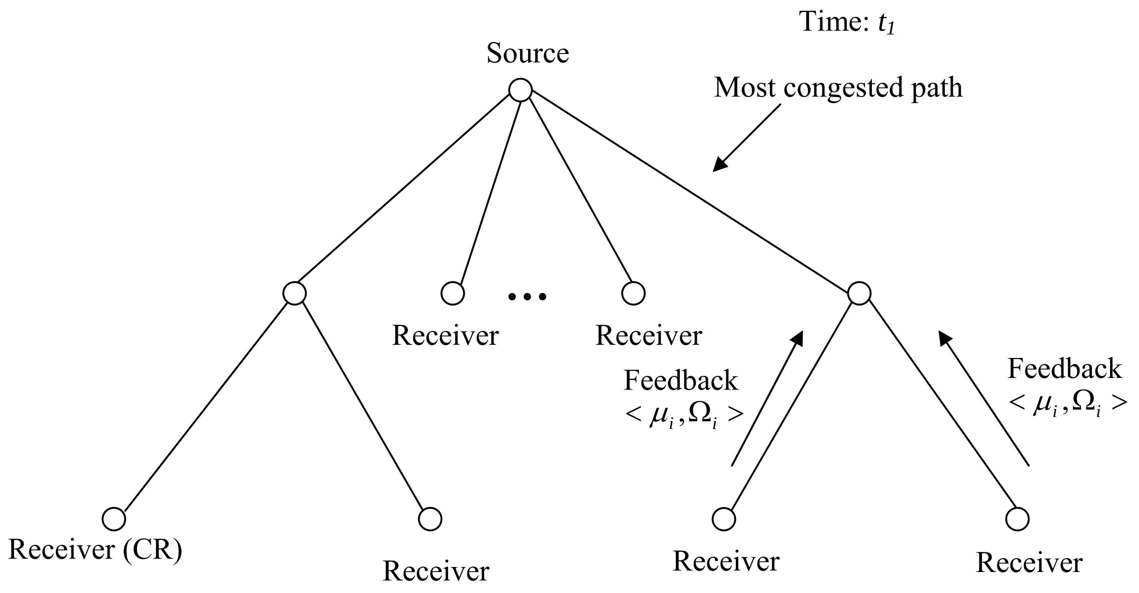
An example operation can illustrate how our scheme works more clearly. In Figure 1 (a), let's assume that at time to the source has chosen a receiver behind the most congested path as CR by comparing average TROCs of receivers. Only the CR will send feedback while other receivers suppress their feedback. These feedbacks are indeed congestion indications (CIs), because they are sent only when congestion is detected due to packet loss. As shown in Figure 1, the feedback from the current CR is the average TROC $\overline{\mu}(t_0) = \phi(\overline{\Omega}(t_0), \alpha)$, where $\overline{\Omega}(t_0)$ is the TROC of the current CR at time t_0 and $\Phi()$ is an EWMA averaging function with a being the exponential averaging factor, which will later be defined in detail. In addition, the TROC, $\overline{\Omega}(t_0)$, for the current CR is calculated by averaging *instantaneous output rate* $\overline{\omega}(t_0)$ of the current CR over a small period of time. [2]



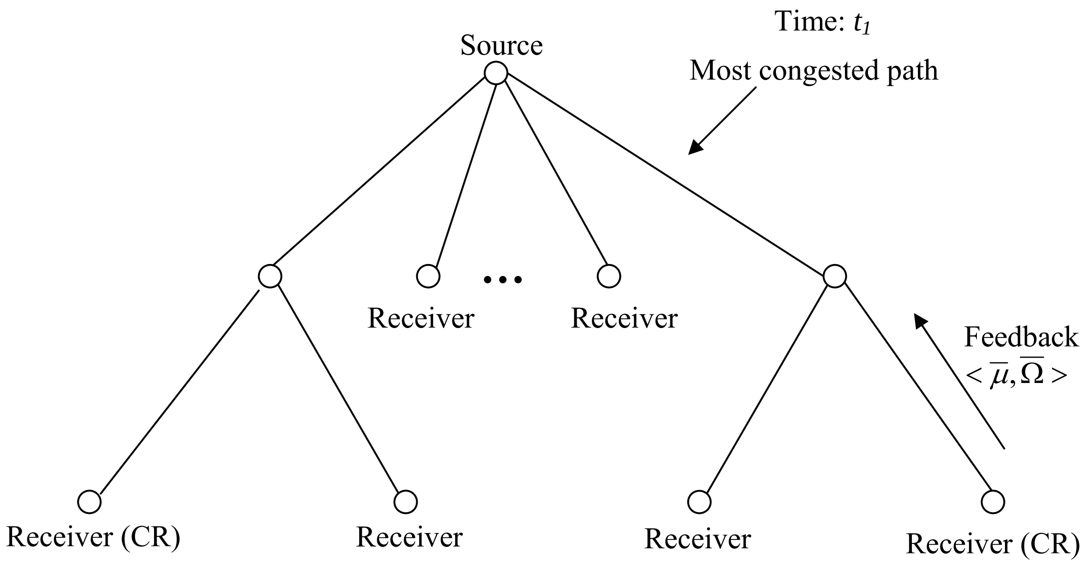
μ: *Avearage TROC*
Ω: *Throughput At Congestion*(*TROC*)

(a)

1 Note that it is also possible to use additional techniques to detect congestion. We do not focus on this to assure needed emphasis on the multicast congestion control rather than congestion detection.
2 These definitions of the different kinds of TROCs correspond to averaging at two different time-scales with two different methods, and they are calculated in the same manner for all receivers. will give more detailed explanation of these definitions later.



(b)



(c)

Fig.1. Example operation of ESMCC

Assume that, after some time another path becomes the new most congested path. After a while at time t_1, those receivers *1..K* behind that path will see average TROCs lower than that of the current CR $(\text{i.e.} \forall i = 1..k, \mu_i(t_1) < \overline{\mu}(t_0) - \overline{\sigma}(t_0))$, and will send feed backs as shown in Figure 1 (b). As the result, one of them will be chosen as the new CR. After that, again, other receivers will suppress their feedback as shown in Figure 1 (c)

1.2 Key Contributions

ESMCC introduces a novel method of using *explicit rate* feedback at the time of congestion (i.e. TROC) in such a way that several major multicast congestion control problems are remedied. By using *smoothing techniques* like EWMA, receivers in ESMCC successfully achieve efficient *feedback suppression.* Similarly, each receiver maintains two *statistical measures* (i.e. average TROC and deviation of TROC) which provides venue for *robust and effective tracking of the slowest receiver.* By using an AIMD-like rate adaptation technique, ESMCC also warrants *TCP-friendliness* and immunity to *the drop-to-zero problem.* In addition, only state of the slowest receiver (i.e. 0 (1) memory complexities) is needed and only estimation of the RTT to the CR is needed.

2) Related Works

2.1 Single-Rate Schemes

In single-rate schemes, all receivers get the same data rate, and the source adapts to the slowest receiver. These schemes are nice in that they do not require the source to transmit multiple streams or use special data coding. Furthermore, many single-rate multicast protocols have been proposed which try to implement a TCP-like service over multicast, so there is some interest in adding congestion control to such protocols especially when deployment in the Internet is desired.

Single-rate schemes have known limitations in presence of large or heterogeneous groups: a single slow receiver can drag down the data rate for the whole group. Furthermore, uncorrelated losses at receivers are not easy to handle, and an improper aggregation of feedback is likely to cause the so called "drop-to-zero" problem [16], where the sender's estimate of the loss rate is much higher than the actual loss rate experienced at every single receiver.

DeLucia et. aI's work in [1] is an early single-rate multicast congestion control scheme using representatives. It requires two types of feedback from receivers, *Congestion Clear (CC)* and *Congestion Indication (CI).* Note that their CIs are single bit and thus different from ours carrying the explicit output rateμ. A fixed number of receiver representatives are maintained at the source. Whenever a CI is received by the source, if the sender of this CI is in the representative set, the representative is refreshed; if not, the sender will replace the representative that has not been refreshed for the longest time. Feedback from representatives is echoed by the source to suppress feedback scheduled at non-representative receivers. The source uses only the feedback from representatives to do MIMD (multiplicative increase and multiplicative decrease) rate adaptation. The representative selection mechanism in that scheme is "simplistic" [1], but there is certain complexity involved in generating *CC*. The representative set is not guaranteed to include the slowest receiver, which means that the slowest receiver can be overloaded. Furthermore, it assumes that only a few bottlenecks cause most of the congestion. Based on this assumption, receiver suppression is the only mechanism for filtering feedback from receivers. In a heterogeneous network, where there may be many different bottlenecks and asynchronous congestion, the assumption may not be true. Consequently, the transmission rate may be reduced more than necessarily and stay very low or close to zero. This is known as the *drop-to-zero* problem.

PGMCC [2], TFMCC [3] and MDP-CC [4] are recent work also using representatives. Although they use different policies for rate adaptation, they all leverage the TCP throughput formula [10] [11] for allocating the slowest receiver, i.e. the receiver with the lowest estimate TCP throughput according to the formula. Therefore, it is necessary for them to measure packet loss rate and RTT for all receivers.

PGMCC [2] keeps one representative as *acker*. The acker sends ACKs to the source which mimics the behavior of TCP. At the same time, NAKs with loss rate are sent from all other receivers. This is different from our scheme because we do not require separate ACK streams. The PGMCC source measures RTT between itself and all receivers in terms of packet numbers, and compares the estimated throughput for updating acker. Due to the necessity of RTT measurement for all receivers, feedback suppression may have serious effect on PGMCC's performance. In fact, PGMCC does not provide a feedback suppression mechanism.

TFMCC [3] adjusts the rate according to the estimated rate calculated by the representative. RTTs are measured by receivers with a somewhat complex procedure. The sender needs to echo receiver's feedback according to some priority order, and there is one-way delay RTT adjustment plus sender-side RTT measurement. TFMCC comes with feedback suppression which is an enhanced version of [12] and is probabilistic timer-based. Therefore, the total number of feedbacks is the function of the estimated total number of receivers, and additional delay is introduced into feedback.

MDP-CC [4] increases/decreases the transmission rate exponentially toward the target rate. Similar to TFMCC, the target rate is also calculated by the representative. in contrast to PGMCC and TFMCC, MDP-CC maintains a pool of representative candidates for representative update. As shown in that paper, maintaining multiple representative candidates requires much effort. MDP-CC can use probabilistic timer-based feedback suppression which has the same properties as that of TFMCC.

2.2 Multi-Rate Schemes

Multi-rate schemes are based on the ability to generate the same data at different rates over multiple streams (generally organized as cumulative layers), either at the source, or as a result of a filtering/distillation process done by intermediate elements such as routers or transcoders. Receiver try to listen to one or more streams matching their capacity, thus effectively realizing a partitioning of the set of receivers into different groups. This approach is suitable to both audio and video streams, and to reliable data transfer by using proper coding techniques [15].

The advantage of multi-rate schemes is that receivers with different needs can be served at a rate closer to their needs, rather than having to match the speed of the slowest receiver in the group. This flexibility is paid in terms of coding costs, some bandwidth inefficiency, and possibly a more coarse match of source and receiver data rate. The most noticeable among them are recently developed Fine-Grained Layered Multicast [6] and STAIR [7]. However, the multi-rate schemes are *closely coupled* with routing and IGMP, which implies some potential problems. Aggregated multicast trees do not necessarily prune trees dynamically and hence break the assumptions of the multi-rate schemes. The slackness of response to congestion due to long leave latency continues to be an issue. Besides, frequent group joins and leaves can introduce significant load at routers.

2.3 Additive increase and multiplicative decrease (AIMD)

TCP maintains a new state variable for each connection-- CongestionWindow---which is used by the source to limit how much data it is allowed to have in transit at a given time. The congestion window is congestion control's counterpart to flow control's advertised window. TCP is modified to have no more than the minimum of the congestion window and the advertised window bytes of unacknowledged data. Thus, using the variables TCP's effective window is revised as follows:

MaxWindow=MIN(CongestionWindow,AdvertisedWindow)
EffectiveWindow = MaxWindow - (LastByteSent - LastByteAcked)

That is, MaxWindow replaces AdvertisedWindow in the calculation of EffectiveWindow. Thus, a TCP source is allowed to send no faster than the slowest component---the network or the destination host---can accommodate.

The problem, of course, is how TCP comes to learn an appropriate value for CongestionWindow. Unlike the AdvertisedWindow, which is sent by the receiving side of the connection, there is no one to send a suitable CongestionWindow to the sending side of TCP. The answer is that the TCP source sets the CongestionWindow based on the level of congestion it perceives to exist in the network. This involves decreasing the congestion window when the level of congestion goes up, and increasing the congestion window when the level of congestion goes down. Taken together, the mechanism is commonly called *additive increase/multiplicative decrease*.

The key question, then, is how does the source determine that the network is congested and it should decrease the congestion window? The answer is based on the observation that the main reason packets are not delivered, and a timeout results, is that a packet was dropped due to congestion. It is rare that a packet is dropped because of an error during transmission. Therefore, TCP interprets timeouts as a sign of congestion, and reduces the rate at which it is transmitting. Specifically, each time a timeout occurs, the source sets CongestionWindow to half of its previous value. This halving of the CongestionWindow for each timeout corresponds to the ``multiplicative decrease'' part of the mechanism.

Although CongestionWindow is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets. For example, suppose the CongestionWindow is currently set to 16 packets. If a loss is detected, CongestionWindow is set to 8. (Normally, a loss is detected when a timeout occurs, but as we see below, TCP has another mechanism to detect dropped packets.) Additional losses cause CongestionWindow to be reduced to 4, then 2, and finally 1 packet. CongestionWindow is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size* (MSS).

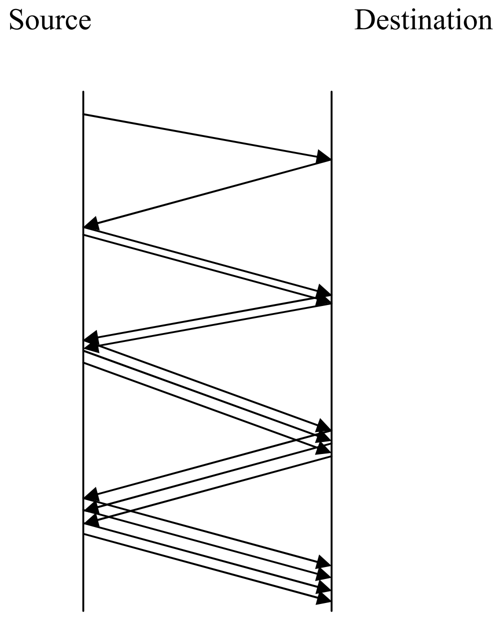


Figure 2: Packets in transit during additive increase: add one packet each RTT.

A congestion control strategy that only decreases the window size is obviously too conservative. We also need to be able to increase the congestion window to take advantage of newly available capacity in the network. This is the ``additive increase'' part of the mechanism, and it works as follows. Every time the source successfully sends a CongestionWindow's worth of packets---that is, each packet sent out during the last RTT has been ACK'ed---it adds the equivalent of one packet to CongestionWindow. This linear increase is illustrated in Figure 2. Note that in practice, TCP does not wait for an entire window's worth of ACKs to add one packet's worth to the congestion window, but instead increments CongestionWindow by a little for each ACK that arrives. Specifically, the congestion window is incremented as follows each time an ACK arrives:

```
Increment=(MSS*MSS)/CongestionWindow
CongestionWindow += Increment
```

That is, rather than incrementing CongestionWindow by an entire MSS each RTT, we increment it by a fraction of MSS every time an ACK is received. Assuming each ACK acknowledges the receipt of MSS bytes, then that fraction is MSS/CongestionWindow.

This pattern of continually increasing and decreasing the congestion window continues throughout the lifetime of the connection. In fact, if you plot the current value of CongestionWindow as a function of time, you get a ``sawtooth" pattern. The important thing to understand about additive increase/multiplicative decrease is that the source is willing to reduce its congestion window at a much faster rate than it is willing to increase its congestion window. This is in contrast to an additive increase/additive decrease strategy in which the window in incremented by 1 packet when an ACK arrives and decreased by 1 when a timeout occurs. It is has been shown that additive increase/multiplicative decrease is a necessary condition for a congestion control mechanism to be stable.

Finally, since a timeout is an indication of congestion, triggering multiplicative decrease, TCP needs the most accurate timeout mechanism it can afford. The two main things to remember about that mechanism are that (1) timeouts are set as a function of both the average RTT and the standard deviation in that average, and (2) due to the cost of measuring each transmission with an accurate clock, TCP only samples the round trip time once per RTT (rather than once per packet) using a coarse-grain (500ms) clock.

2.3.1. Definitions

This section provides the definition of several terms that will be used throughout the remainder of this document.

SEGMENT:

A segment is ANY TCP/IP data or acknowledgment packet (or both).

SENDER MAXIMUM SEGMENT SIZE (SMSS):

The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.

RECEIVER MAXIMUM SEGMENT SIZE (RMSS):

The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup. Or, if the MSS option is not used, 536 bytes. The size does not include the TCP/IP headers and options.

FULL-SIZED SEGMENT:

A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).

RECEIVER WINDOW (rwnd):

The most recently advertised receiver window.

CONGESTION WINDOW (cwnd):

A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

INITIAL WINDOW (IW):

The initial window is the size of the sender's congestion window after the three-way handshake is completed.

LOSS WINDOW (LW):

The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission timer.

RESTART WINDOW (RW):

The restart window is the size of the congestion window after a TCP restarts transmission after an idle period

FLIGHT SIZE:

The amount of data that has been sent but not yet acknowledged.

2.3.2 Slow Start

The additive increase mechanism just described is the right thing to do when the source is operating close to the available capacity of the network, but it takes too long to ramp up a connection when it is starting from scratch. TCP therefore provides a second mechanism, ironically called slow start that is used to increase the congestion window rapidly from a cold start. Slow start effectively increases the congestion window exponentially, rather than linearly.
Specifically, the source starts out by setting CongestionWindow to one packet. When the ACK for this packet arrives, TCP adds one packet to CongestionWindow and then sends two packets. Upon receiving the corresponding two ACKs, TCP increments CongestionWindow by two---one for each ACK---and next sends four packets. The end result is that TCP effectively doubles the number of packets it has in transit every RTT. Figure 3 shows the growth in the number of packets in transit during slow start. Compare this to the linear growth of additive increase illustrated in Figure 2.

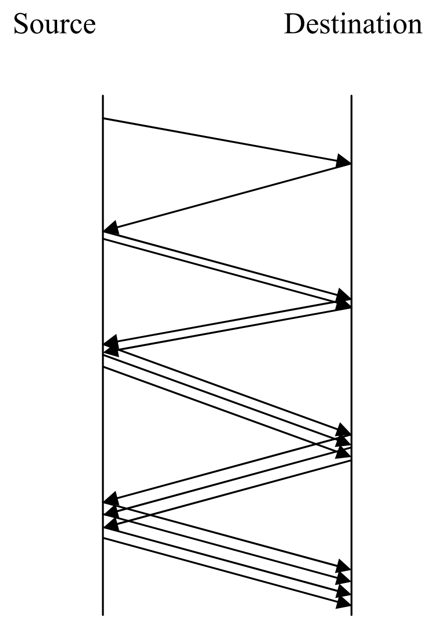


Figure 3: Packets in transit during slow start.

Why any exponential mechanism would be called ``slow'' is puzzling at first, but can be explained if put in the proper historical context. We need to compare slow start not against the linear mechanism of the previous subsection, but against the original behavior of TCP. Consider what happens when a connection is established and the source first starts to send packets; i.e., it currently has no packets in transit. If the source sends as many packets as the advertised window allows---which is exactly what TCP did before slow start was developed---then even if there is a fairly large amount of bandwidth available in the network, the routers may not be able to consume this burst of packets. It all depends on how much buffer space is available at the routers. Slow start was therefore designed to space packets out so that this burst does not occur. In other words, even though its exponential growth is faster than linear growth, slow start is much ``slower'' than sending an entire advertised window's worth of data all at once.

There are actually two different situations in which slow start runs. The first is at the very beginning of a connection, at which time the source has no idea how many packets it is going to be able to have in transit at a given time. (Keep in mind that TCP runs over everything from 9600bps links to 2.4Gbps links, so there is no way for the source to know the network's capacity.) In this situation, slow start continues to double CongestionWindow each RTT until there is a loss, at which time a timeout causes multiplicative decrease to divide CongestionWindow by two.

The second situation where slow start is used is a bit more subtle; it occurs when the connection goes dead waiting for a timeout to occur. Recall how TCP's sliding window algorithm works---when a packet is lost, the source eventually reaches a point where it has sent as much data as the advertised window allows, and so it blocks waiting for an ACK that will not arrive. Eventually, a timeout happens, but by this time there are no packets in transit, meaning that the source will receive no ACKs to ``clock'' the transmission of new packets. The source will instead receive one big cumulative ACK that reopens the entire advertised window, but as explained above, the source uses slow start to restart the flow of data rather than dumping a window's worth of data on the network all at once.

Although the source is using slow start again, it now knows more information than it did at the beginning of a connection. Specifically, the source has the current value of CongestionWindow, which because of the timeout, has already been divided by two. Slow start is used to rapidly increase the sending rate up to this value, and then additive increase is used beyond this point. Notice that we have a tiny bookkeeping problem to take care of, in that we want to remember the ``target'' congestion window resulting from multiplicative decrease, as well as the ``actual'' congestion window being used by slow start. To address this problem, TCP introduces a temporary variable, typically called CongestionThreshold that is set equal to the CongestionWindow resulting from multiplicative decrease. Variable CongestionWindow is then reset to one packet, and it is incremented by one packet for every ACK that is received until it reaches CongestionThreshold, at which point is incremented by one packet per RTT.

In other words, TCP increases the congestion window as defined by the following code fragment:

```
{
    u_int    cw = state->CongestionWindow;
    u_int    incr = state->maxseg;

    if (cw > state->CongestionThreshold)
        incr = incr * incr / cw;
    state->CongestionWindow = MIN(cw + incr, TCP_MAXWIN);
}
```

where state represents the state of a particular TCP connection and TCP_MAXWIN defines an upper bound on how large the congestion window is allowed to grow.

Ttraces how TCP's CongestionWindow increases and decreases over time, and serves to illustrate the interplay of slow start and additive increase/multiplicative decrease. This trace was taken from an actual TCP connection, and traces the current value of CongestionWindow---the thick grey line---over time. The graph also depicts other information about the connection:

- the vertical bars show when a packet that was eventually retransmitted was first transmitted,
- the small hash marks at the top of the graph show the time when each packet is transmitted, and
- the circles at the top of the graph show when a timeout occurs.

There are several things to notice about this trace. The first is the rapid increase in the congestion window at the beginning of the connection. This corresponds to the initial slow start phase. The slow start phase continues until several packets are lost at about 0.4 seconds into the connection, at which time CongestionWindow flattens out at about 34KB. (Why so many packets are lost during slow start is discussed below.) The reason the congestion window flattens is that there are no ACKs arriving, due to the fact that several packets were lost. In fact, no new packets are sent during this time, as denoted by the lack of tick marks at the top of the graph. A timeout eventually happens at approximately 2 seconds, at which time the congestion window is divided by two (i.e., cut from approximately 34KB to around 17KB), and CongestionThreshold is set to this value. Slow start then causes CongestionWindow to be reset to one packet, and start ramping up.

There is not enough detail in the trace to see exactly what happens when a couple of packets are lost just after 2 seconds, so we jump ahead to the linear increase in the congestion window that occurs between 2 and 4 seconds. This corresponds to additive increase. At about 4 seconds, CongestionWindow flattens out, again due to a lost packet. Now, at about 5.5 seconds

- a timeout happens, causing the congestion window to be divided by two, dropping it from approximately 22KB to 11KB, and CongestionThreshold is set to this amount;
- CongestionWindow is reset to one packet, as the sender enters slow start;
- slow start causes CongestionWindow to grow exponentially until it reaches CongestionThreshold;
- CongestionWindow then grows linearly.

The same pattern is repeated at around 8 seconds when another timeout occurs.

We now return to the question of why so many packets are lost during the initial slow start period. What TCP is attempting to do here is learn how much bandwidth is available on the network. This is a very difficult task. If the source is not aggressive at this stage, for example only increasing the congestion window linearly, then it takes a long time for it to discover how much bandwidth is available. This can have a dramatic impact on the throughput achieved for this connection. On the other hand, if the source is aggressive at this stage, as is TCP during exponential growth, then the source runs the risk of having half a window's worth of packets dropped by the network.

To see what can happen during exponential growth, consider the situation where the source was just able to successfully send 16 packets through the network, and then doubles its congestion window to 32. Suppose, however, that the network just happens to have enough capacity to support only 16 packets from this source. The likely result is that 16 of the 32 packets sent under the new congestion window will be dropped by the network; actually, this is the worst case outcome, since some of the packets will be buffered in some router. This problem will become increasing severe as the *delay* bandwidth* product of networks increases. For example, a *delay*bandwidth* product of 500KB means that each connection has the potential to lose up to 500KB of data at the beginning of each connection. Of course, this assumes both the source and destination implement the ``big windows'' extension.

Some have proposed alternatives to slow start whereby the source tries to estimate the available bandwidth through more clever means of sending out a bunch of packets and seeing how many make it through. A technique called *packet-pair* is representative of this general strategy. In simple terms, the idea is to send a pair of packets with no spacing between them. Then, the source sees how far apart the ACKs for those two packets are. The gap between the ACKs is taken as a measure of how much congestion there is in the network, and therefore, how much increase in the congestion window is possible. The jury is still out on the effectiveness of approaches such as this, although the results are promising.

2.3.3 Fast Retransmit and Fast Recovery

The mechanisms described so far were part of the original proposal to add congestion control to TCP. It was soon discovered, however, that the coarse-grain implementation of TCP timeouts led to long periods of time during which the connection went dead waiting for a timer to expire. Because of this, a new mechanism, called *fast retransmit*, was added to TCP. Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism. The fast retransmit mechanism does not replace regular timeouts, it just enhances that facility.

The idea of fast retransmits it straightforward. Every time a data packet arrives at the receiving side, the receiver responds with an acknowledgement, even if this sequence number has already been acknowledged. Thus, when a packet arrives out of order---that is, TCP cannot yet acknowledge the data it contains because earlier data has not yet arrived---TCP resends the same acknowledgement it sent last time. This second transmission of the same acknowledgement is called a *duplicate ACK*. When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost. Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees some number of duplicate ACKs, and then retransmits the missing packet. In practice, TCP waits until it has seen three duplicate ACKs before retransmitting the packet.

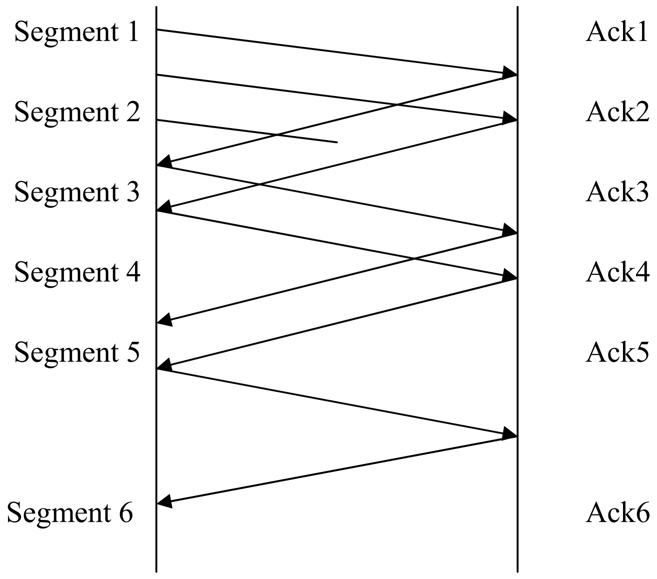


Figure 4: Fast retransmit based on duplicate ACKs.

Figure 4, illustrates how duplicate ACKs leads to a fast retransmit. In this example, the destination receives packets 1 and 2, but packet 3 is lost in the network. Thus, the destination will send a duplicate ACK for packet 2 when packet 4 arrives, again when packet 5 arrives, and so on. (To simplify this example, we think in terms of packet 1, 2, 3, and so on, rather than worrying about the sequence numbers for each byte.) When the sender sees the third duplicate ACK for packet 2---the one sent because the receiver had gotten packet 6---it retransmits packet 3. Note that when the retransmitted copy of packet 3 arrives at the destination, it then sends a cumulative ACK for everything up to and including packet 6 back to the source.

The behavior of a version of TCP with the fast retransmit mechanism. It is interesting to compare the trace , where fast retransmit was not implemented---the long periods during which the congestion window stays flat and no packets are sent has been eliminated. In general, this technique is able to eliminate about half of the coarse-grain timeouts on a typical TCP connection, which results in roughly a 20% improvement in the throughput over what could have otherwise been achieved. Notice, however, that the fast retransmit strategy does not eliminate all coarse-grained timeouts. This is because for a small window size, there will not be enough packets in transit to cause enough duplicate ACKs to be delivered. Given enough lost packets---for example, as happens during the initial slow start phase---the sliding window algorithm eventually blocks the sender until a timeout occurs. Given the current 64KB maximum advertised window size, TCP's fast retransmit mechanism is able to detect up to three dropped packets per window in practice.

Finally, there is one last improvement we can make. When the fast retransmit mechanism signals congestion, rather than drop the congestion window all the way back to one packet and run slow start, it is possible to use the ACKs that are still in the pipe to clock the sending of packets. This mechanism, which is called fast recovery, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins. In other words, slow start is only used at the beginning of a connection and whenever a coarse-grain timeout occurs. At all other times, the congestion window is following a pure additive increase/multiplicative decrease pattern.

3) ESMCC

As we have mentioned in the introduction, in ESMCC, receivers send their average TROCs back to the sender whenever necessary, and the sender dynamically chooses a representative (CR) out of them and use only its TROCs to adjust the sending rate. In this section, we will present the details of how the whole scheme works. We will first present operations at an ESMCC receiver and at the source, followed by a list of the key features of ESMCC.

3.1 ESMCC Receiver

Receivers in ESMCC perform two major functions: (i) calculation and maintenance of TROC and average TROC, and (ii) proper generation and suppression of feedbacks to the source. The former function is crucial since TROC is used to help the source in rate adaptation as well as in deciding which receiver will be the CR. The latter function is also important in that it determines scalability of ESMCC in terms of two well-known single-rate multicast problems: feedback-implosion, and slowest receiver tracking.

3.1.1 Throughput Rate On Congestion (TROC) - $\mu(t), \Omega(t), \omega(t)$

Upon detection of a packet loss at a receiver in ESMCC, that receiver measures explicit output rate TROC $\Omega(t)$ and updates the average TROC $\mu(t)$. We represent TROC measured at time *t* at receiver ***i*** as $\Omega_i(t)$. Measurement of TROC is done over a small time period Δt which we take as 1 second for all cases.

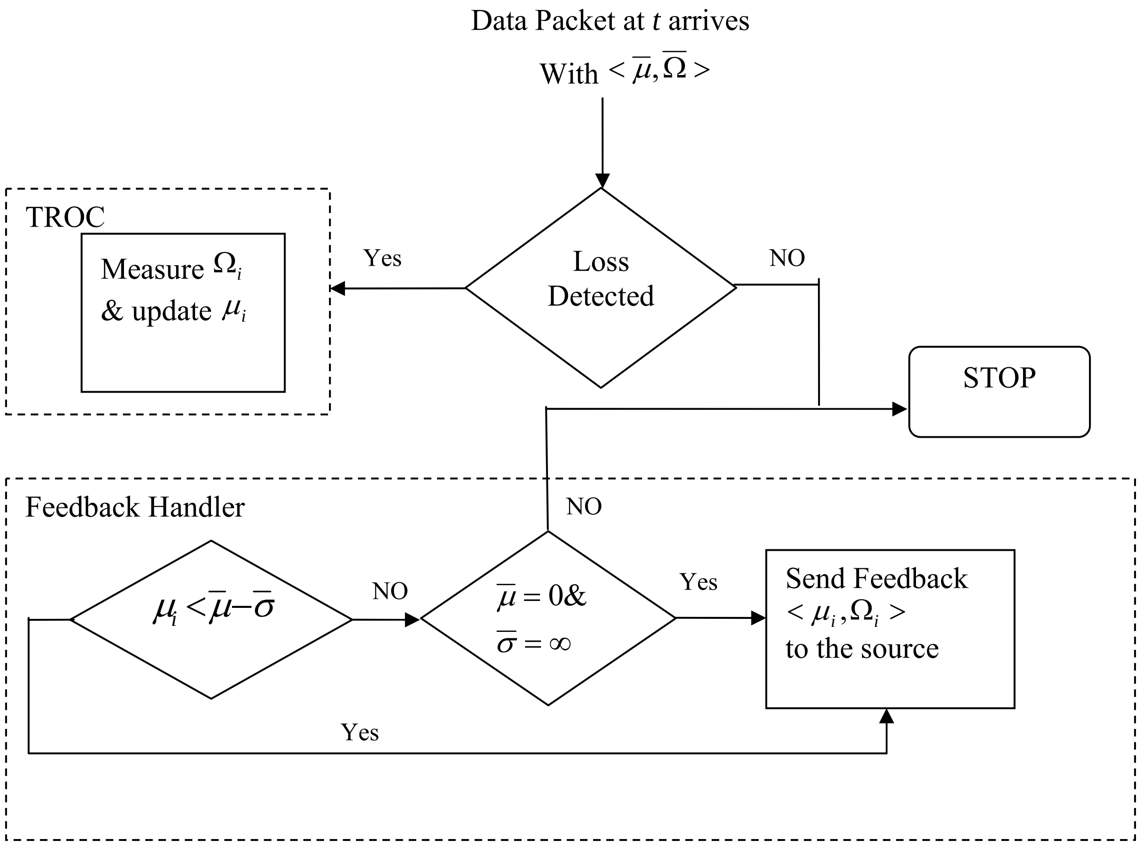
Thus, measurement of average TROC $\mu(t)$ includes two levels of averaging. The first averaging is done to measure the TROC, which can be expressed as averaging of instantaneous output rate $\omega(t)$. So, TROC at receiver ***i*** at time ***t*** is calculated as:

$$\Omega_i(t) = E[\omega_i(t)] = \frac{1}{\Delta t}\int_{t}^{t+\Delta t} \omega_i(t)\,dt$$

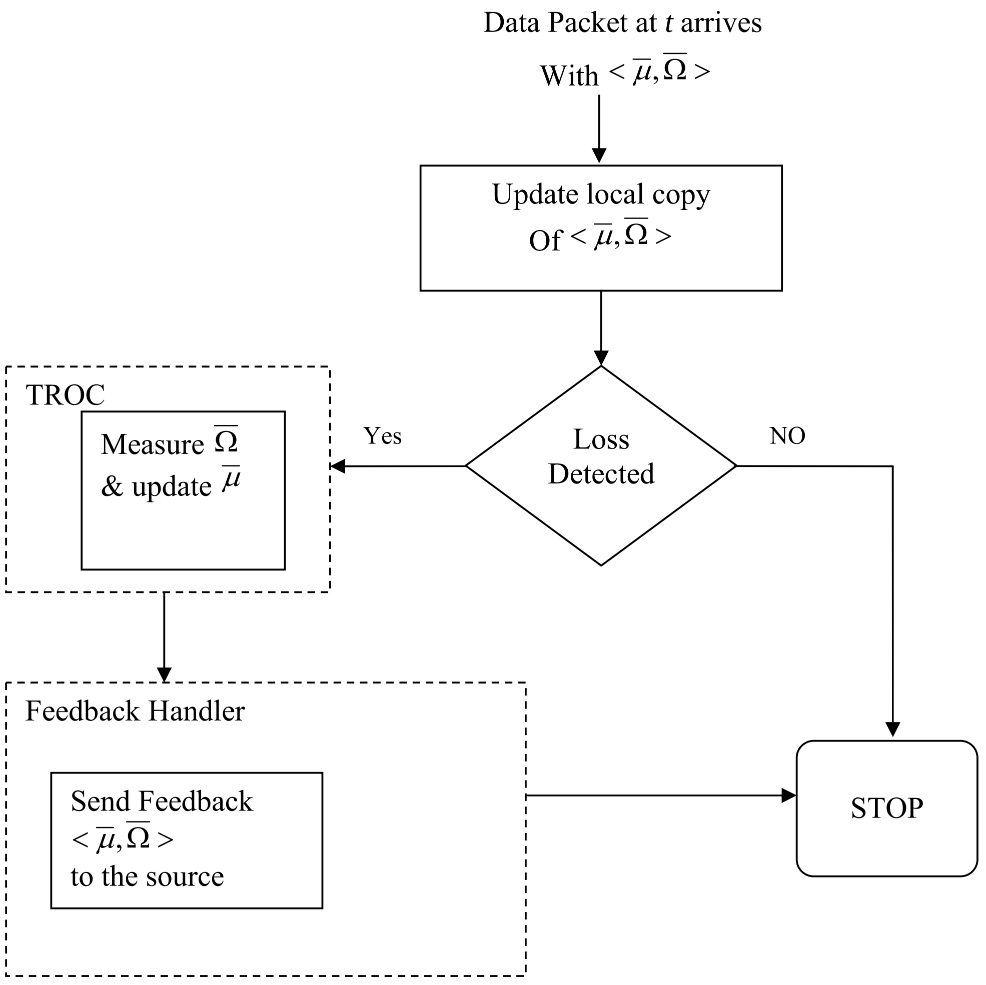
The second level of averaging is done by a moving average function $\Phi()$ (i.e. EWMA) with an exponential weighting factor of α. The value of α determines importance of the previous TROC values in the resulting average. So, given that the previous packet loss happened at time $\boldsymbol{t_o}$, average TROC at receiver ***i*** at time t_1 is calculated by a recursive relationship:

$$\mu_i(t_1) = \phi(\mu_i(t_0), \Omega_i(t_1), \alpha)$$
$$\mu_i(t_1) = (1-\alpha)\mu_i(t_0) + \alpha\Omega_i(t_1)$$

To distinguish these measures for CR we will use a hat on the notation for the rest of the paper. So, $\overline{\mu}(t), \overline{\Omega}(t)$, and $\overline{\omega}(t)$ represents the average TROC, the TROC and the instantaneous output rate for the current CR of the multicast session.


Data Packet at t arrives
With $<\overline{\mu}, \overline{\Omega}>$
TROC
Measure Ω_i & update μ_i
Yes
Loss Detected
NO
STOP
Feedback Handler
NO
$\mu_i < \overline{\mu} - \overline{\sigma}$
NO
$\overline{\mu} = 0\&$ $\overline{\sigma} = \infty$
Yes
Send Feedback $<\mu_i, \Omega_i>$ to the source
Yes

(a) a non-CR Receiver i


Data Packet at *t* arrives
With $<\overline{\mu},\overline{\Omega}>$
Update local copy
Of $<\overline{\mu},\overline{\Omega}>$
Loss Detected
Yes
NO
TROC
Measure $\overline{\Omega}$
& update $\overline{\mu}$
Feedback Handler
Send Feedback
$<\overline{\mu},\overline{\Omega}>$
to the source
STOP

(b)the current CR receiver

Fig.5:Operations at ESMCC receiver

Similar to average TROC, another important metric to keep track of is the deviation of TROC, because it plays a crucial role in feedback suppression as well as selection of CR which will be detailed in Sections 3.2.1 and 3.1.3 respectively. We represent the deviation of TROC as $\sigma_i(t)$, and calculate it again by means of the EWMA function $\Phi()$:

$$\sigma_i(t_1) = \phi(\sigma_i(t_0), \mu_i(t_1), \Omega_i(t_1), \alpha)$$
$$\sigma_i(t_1) = (1-\alpha)\sigma_i(t_0) + \alpha\left|\mu_i(t_1) - \Omega_i(t_1)\right|$$

3.1.2 Feedback Handler

In ESMCC, as shown in Figure 5-a, feedbacks are generated only when a packet loss is detected. Consider a data packet A at the arrival of which, receiver i detects that some data packets have been lost. The feedback generated by this receiver will contain: (i) the sequence number of the lastly received data packet A, (ii) the TROC,$\Omega_i(t)$, measured at the arrival of A, and (iii) the average TROC, $\mu_i(t)$ So, the feedback will be a tuple of three items. When the feedback arrives at the source, the first item will be used for making RTT estimation for CR, the second item will be used for adjusting the transmission rate, and the last item will be used in the decision-making process of CR selection.

Regarding the meaning of feedbacks in ESMCC, there are two different situations for two different purposes:

- **Required Feedback from the CR**: As shown in Figure 5-b, when the CR detects a packet loss, it needs to send congestion indication as a feedback to the source; so that the source can adjust the transmission rate. Since the feedback includes *TROC*$\overline{\Omega}(t)$, it also serves as a congestion indication since it is measured up on detection of congestion .

- **Optional Feedback**: As shown in Figure 5-a, a non-CR receiver detects a packet loss and generates a feedback only when it thinks that it is slower than the current CR. For receiver ***i***, the necessary condition for sending a feedback is$\mu_i(t) < \overline{\mu}(t) - \overline{\sigma}(t)$. Each non-CR receiver performs this comparison to make effective suppression of unnecessary feedbacks, which we will discuss next.

3.1.3 Feedback Suppression

Effective feedback suppression can reduce the risk of feedback implosion, and allow a multicast congestion control scheme to be used for large groups. In ESMCC, the source conveys the average TROC $\overline{\mu}(t)$ and the deviation $\overline{\sigma}(t)$ of the CR's TROC to receivers whenever the CR is updated or $\overline{\mu}(t)$ and $\overline{\sigma}(t)$ are changed. The source conveys these statistics about the current CR by attaching them to the data packets. A receiver will send feedbacks, only if its own average TROC is less than the current average TROC of the CR by an amount at least the standard deviation of the CR's TROC. That is, for receiver ***i*** the necessary condition for sending a feedback is $\mu_i(t) < \overline{\mu}(t) - \overline{\sigma}(t)$. Note that we do not use a weaker condition of $\mu_i(t) \leq \overline{\mu}(t)$ to be conservative and keep CR stable.

If needed, the source can use this behavior of the receivers to obtain feedbacks from all receivers. $\overline{\mu}(t)$ and $\overline{\sigma}(t)$ conveyed by the source can be changed to large or smaller values so that receivers can send feedbacks. This is needed when the current CR is inactive and the source needs to trigger feedbacks from all receivers for new CR selection (Figure 9). To remedy the possibility of feedback implosion, the source can change these $\overline{\mu}(t)$ and $\overline{\sigma}(t)$ thresholds to obtain feedback from a portion of receivers at a time.

Clearly, no timer is involved in our feedback suppression; no knowledge of the whole group is needed. Unlike other probabilistic timer-based feedback suppression schemes, feedbacks are not scheduled at all before being suppressed. Yet, it is effective since the amount of feedbacks sent to the source is independent of the total number of receivers.

There is one situation which might be of concern. When the current CR is absent and the source needs to choose a new CR, all receivers seeing congestion of similar degree may send feedback at the same time. However, this situation will last at most one RTT, because the new CR will be chosen in one RTT. Besides, in reality, due to the heterogeneity of the network, many receivers will get the information of the new CR before they can send out feedbacks for CR re-selection. Therefore, the total number of feedbacks sent under this situation is limited, and we do not deem it as a problem.

3.2 ESMCC Source

In a single-rate multicast congestion control protocol, the source is responsible for several major functions. These include: (i) proper and scalable selection of the CR that represents the slowest receiver(s) in the multicast session, (ii) proper adaptation of the transmission rate so that available bandwidth utilized as much as possible while assuring that the slowest receiver(s) is not overloaded, and (iii) estimation and maintenance of necessary statistics such as RTT. In order to perform the first function, ESMCC employs a set of *CR Selection* criteria as well as a *CR Mode Control* module that operates at every RTT. Similarly, to perform the second function, ESMCC has a *Rate Increase* module that operates at every RTT and a *Rate Decrease* module that operates at every congestion indication from the receivers.

As it is shown in Figure 6, an ESMCC source has six major functions and modules, each of which has a specific purpose. In the following subsections, we will describe each of these functions and modules in detail.

3.2.1 CR Selection: Tracking of the Slowest Receiver

ESMCC compares average TROC of all receivers to locate the slowest ones, and chooses one of them as the *Congestion Representative (CR).* By using a metric like TROC (which is based on explicit output rate), it avoids computing TCP throughput formula [11] [10] which requires per receiver RTT and packet loss rate.

ESMCC receivers help the source to select a receiver with the lowest average TROC by sending in feedbacks *only if* their average TROCs is low enough to qualify them as CR. It is imperative that the receivers do not send more than necessary or less than enough feedbacks, which necessitates proper and effective suppression of feedbacks. Details of how receivers suppress the feedbacks were covered in Section 3.1.3.

Thus, to make selection of the slowest receiver as the CR, two types of comparisons take place in the system:

1) *Comparison at receivers:* Each receiver checks whether it thinks itself as a potential CR. If so, it sends feedback to compete for being the CR.
2) *Comparison at the source:* The source compares the feedbacks from those receivers who think they are qualified, and makes the final decision of which should be the CR.

These comparisons are shown in detail in Feedback Handler part of Figure 5-a and CR Selection part of Figure 8.

Network conditions always keep changing, and we need to continuously keep our choice of CR up-to-date. There are mainly two situations under which CR needs to be updated:

❖ Case 1: *A non-CR receiver worsens.* The situations of some non-CR receivers change so that one of them sees more severe congestion than the current CR does.
❖ Case 2: *CR improves or leaves.* While the situations of all non-CR receivers remain unchanged, the previously most congested path is improved so that the current CR sees less congestion than other receivers, or it leaves the multicast session.

Tracking the slowest receiver by examining average TROCs can deal with *Case* 1, but to cope with *Case* 2 needs more effort. Under this situation, there can be no feedbacks from the current CR. Recall that the source only considers the feedbacks from the CR for rate adaptation and ignores all other feedbacks.

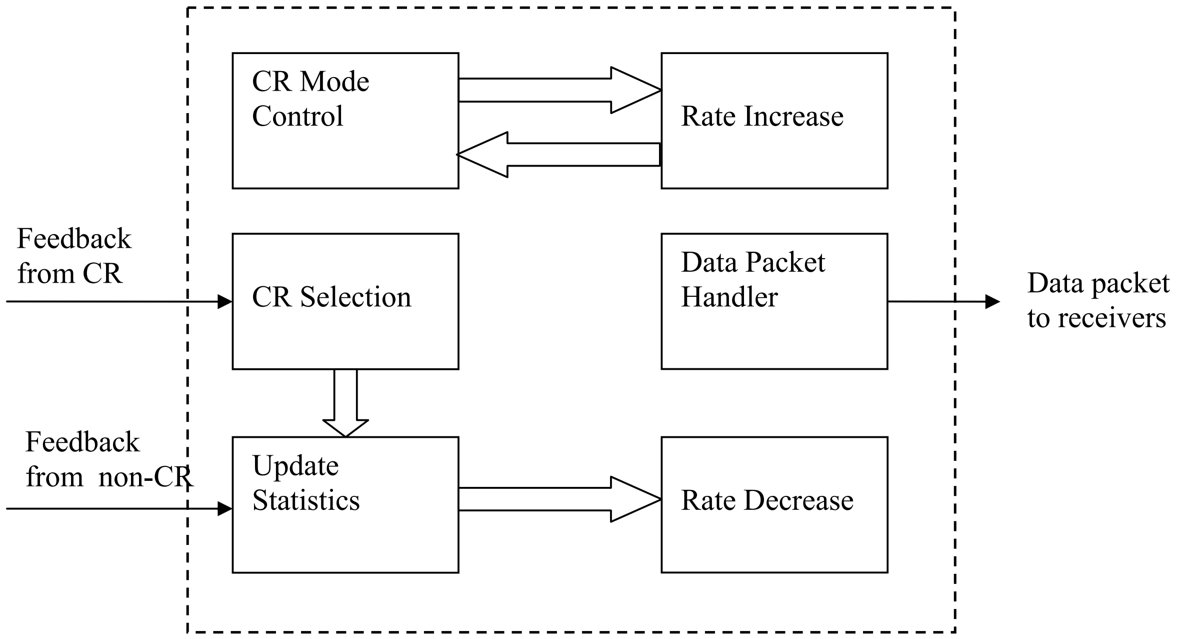


Fig.6:Source Operations as a block diagram

If the source does not change CR in time, the transmission rate will be out of control. To detect this situation, we estimate an upper bound (denoted as *T max)* of the idle time (denoted as *T)* before the source receives the first feedback from the CR when the bottleneck is fully loaded. Notice that *T* is indeed response time of CR during a congestion epoch, so we named it *CR Response Time.* We will give a detailed description of measurement of *T* later in Section 3.2.4.

As shown in CR Selection part of Figure 8, the source in ESMCC defines two modes for the CR, Active or Inactive, which reflect validity of the CR. At every RTT, the source updates the mode of CR. We will detail the update of CR's operation mode in the next Section 3.2.2.

There is one small trick we use to bias the choice of CR towards those receivers with higher RTTs. As shown in CR Inactive Mode part of the CR Selection in Figure 8, right after a new CR is chosen, we start a *longer-RTT* period of *2RTTmax,* where *RTTmax* is the maximum RTT the source has ever seen. Later within this period, as shown in CR Active Mode part of the CR

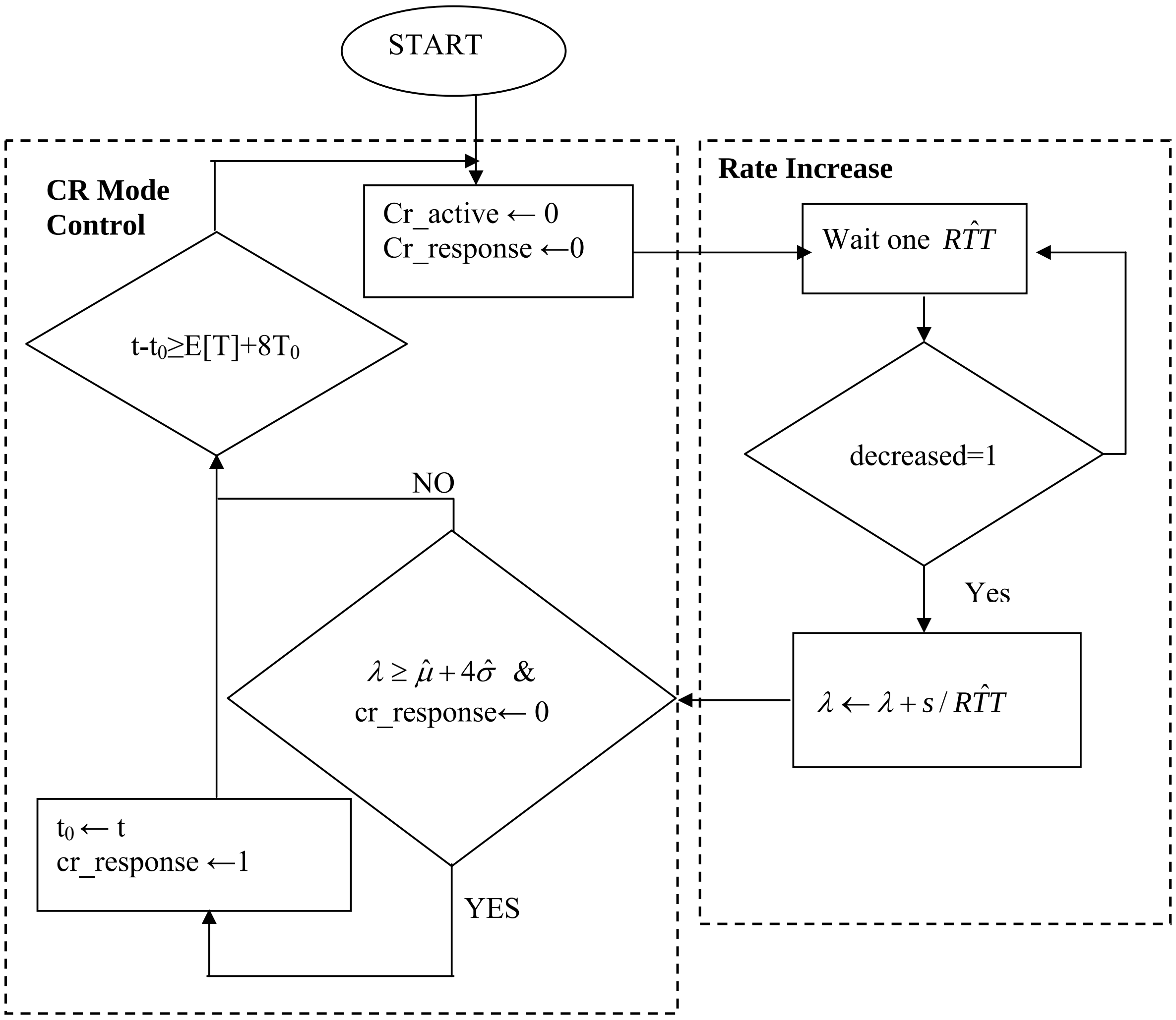


Figure7 :At every RTT , the source attempts to increase the transmission rate and updates the operating mode of the source as either CR active or CR inactive

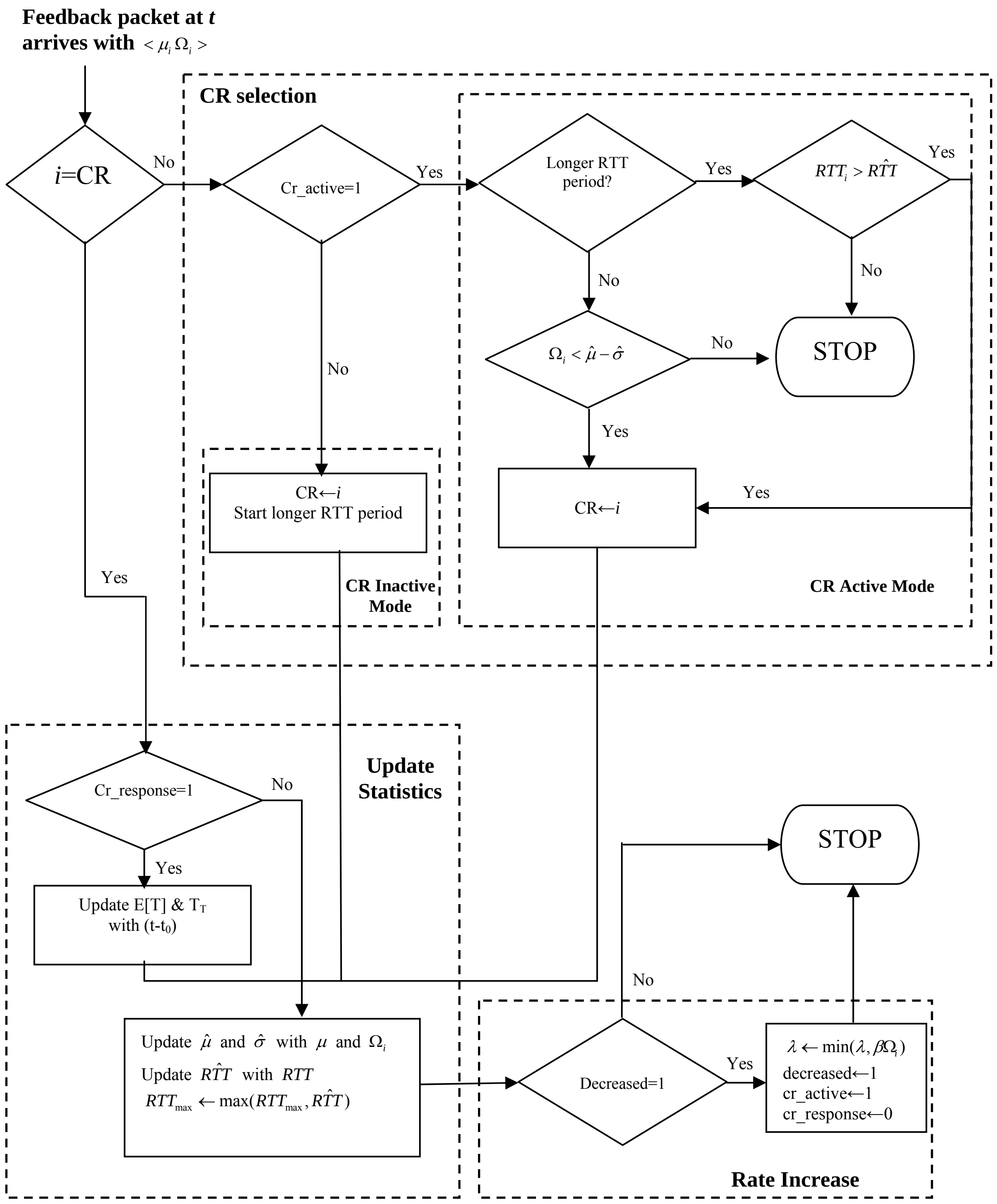

Feedback packet at t arrives with $<\mu_i \Omega_i>$
i=CR
No
Yes
CR selection
Cr_active=1
Yes
No
Longer RTT period?
Yes
No
$RTT_i > \hat{RTT}$
Yes
No
$\Omega_i < \hat{\mu} - \hat{\sigma}$
No
Yes
STOP
CR←i Start longer RTT period
CR Inactive Mode
CR←i
Yes
CR Active Mode
Update Statistics
Cr_response=1
No
Yes
Update E[T] & T_T with (t-t_0)
Update $\hat{\mu}$ and $\hat{\sigma}$ with μ and Ω_i
Update $\hat{RTT}$ with RTT
$RTT_{max} \leftarrow \max(RTT_{max}, \hat{RTT})$
STOP
No
Decreased=1
Yes
$\lambda \leftarrow \min(\lambda, \beta\Omega_i)$
decreased←1
cr_active←1
cr_response←0
Rate Increase

Figure 8: Operations that take place when a feedback packet from receiver ***i*** arrives at the source

Selection in Figure 8, if the source receives a feedback from another receiver with similar average TROC as that of the CR, it will update CR to this receiver, since this one tends to have longer RTT. Notice that the longer-RTT period is not reset after CR switches within the longer-RTT period.

3.2.2 CR Mode Control

To determine whether or not the selected CR is active, the source uses two measures: (i) an estimate of the time when the bottleneck becomes fully loaded, and (ii) *Tmax,* an estimate of the time it would maximally take the current CR to respond during congestion. Basically, the source starts to count when it detects the time corresponding to the first estimate above. And then, it identifies the CR as Inactive when the count reaches the second time estimate above. In other words, suppose we somehow detect that the bottleneck is fully loaded at time ***t.*** If there has been no feedback from the current CR until ***t*** + *Tmax,* we can say that the current CR is now inactive and needs to be changed. Indeed, this is sort of a timeout on TROC of the current CR. This process of mode determination can be seen in the flowchart shown in CR Mode Control part of Figure 7.

To see this mode control process on a timeline, let's look at Figure 9. When the CR is still active, we measure samples of T at the source, using feedback packets only from CR. When the transmission rate reaches $\overline{\mu}(t)+4\overline{\sigma}(t)$ [3], we assume that bottleneck becomes fully loaded and start to count. Let the current time be t_o. At a later time t_l, suppose the first feedback from the CR arrives at the source. Then, t_l **-** ***to*** is a sample of T and we update the average and deviation of T again with EWMA just like we did for the TROC in (2) and (3).T_{max} is the average value of T plus eight times its deviation [4], i.e.

$$T_{\max} = E[T] + 8T_{\sigma}$$

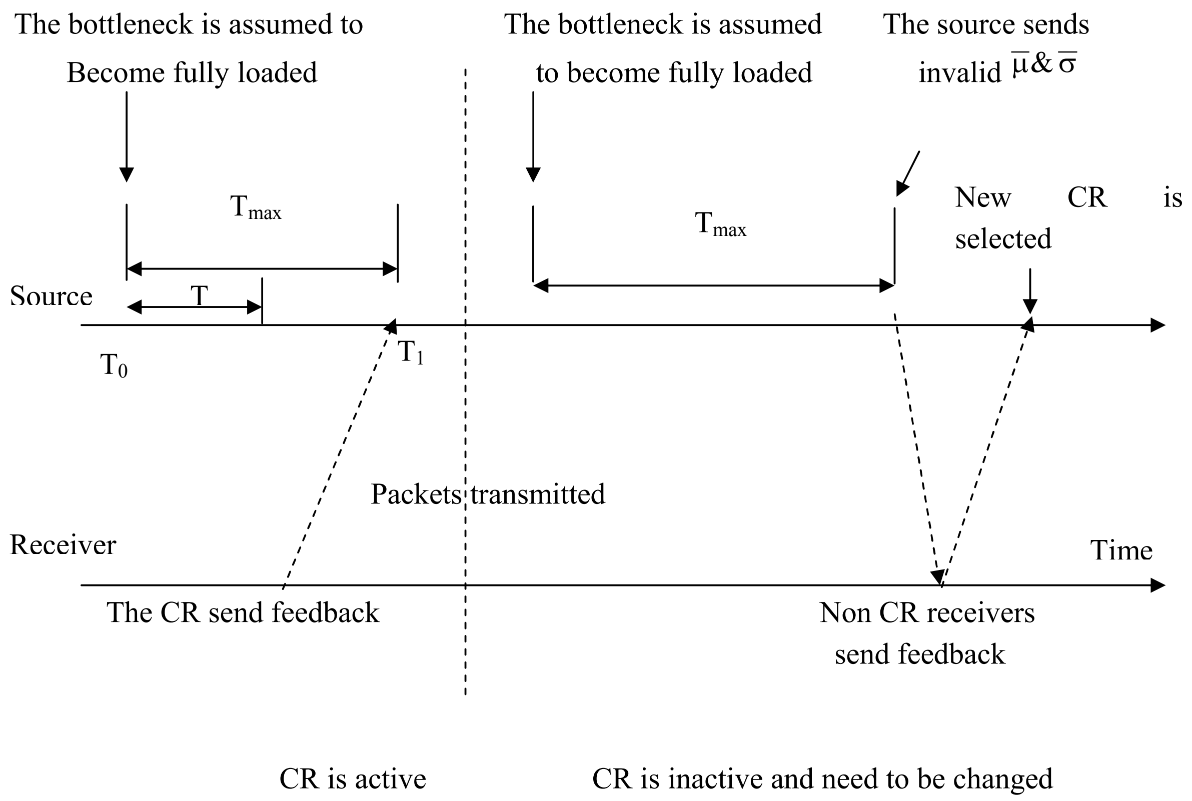

The bottleneck is assumed to Become fully loaded
The bottleneck is assumed to become fully loaded
The source sends invalid $\overline{\mu}$ & $\overline{\sigma}$
T_{max}
T_{max}
New CR is selected
Source
T
T_0
T_1
Packets transmitted
Receiver
Time
The CR send feedback
Non CR receivers send feedback
CR is active
CR is inactive and need to be changed

Fig.9. Sketch of updating congestion Representative(CR).

When the CR is not active, for the duration of T_{max} since we start to count, no feedback will be received by the source. The source then requests feedback from other receivers for new CR selection, as described in Section 3.1.3.

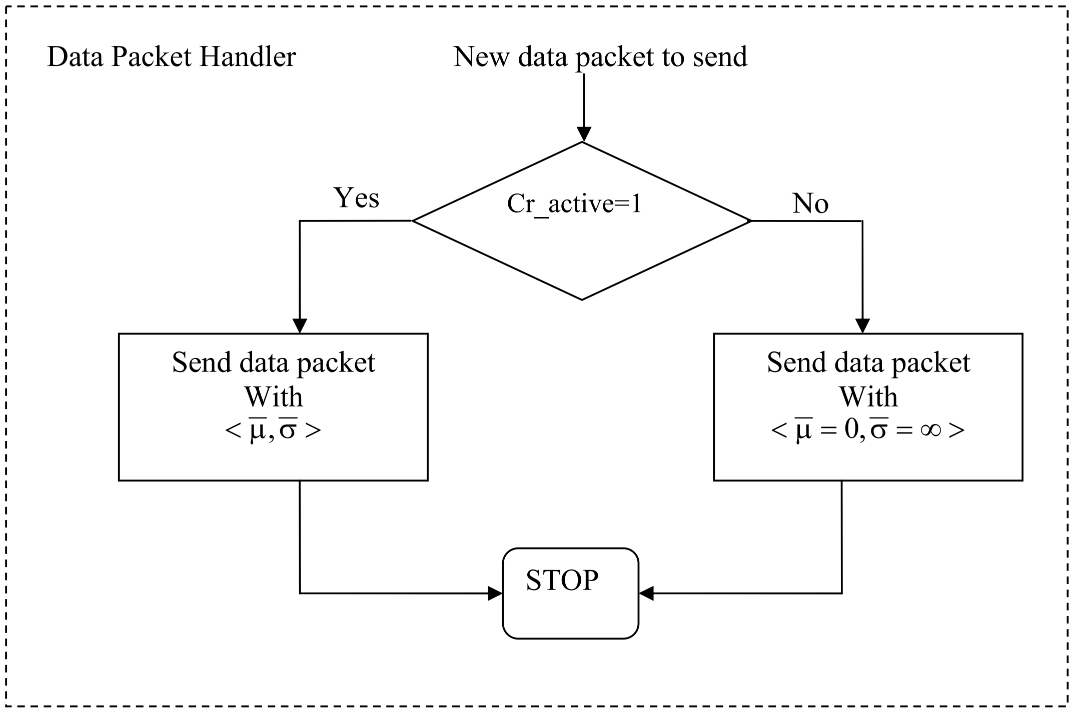

Data Packet Handler
New data packet to send
Cr_active=1
Yes
No
Send data packet With $<\overline{\mu},\overline{\sigma}>$
Send data packet With $<\overline{\mu}=0,\overline{\sigma}=\infty>$
STOP

Figure 10: Handling of data packets at the source: Source keeps attaching $\overline{\mu}\&\overline{\sigma}$ to every data packet.

3 According to Chebychev's Inequality, about 94% of the random samples are less than this value.
4 We choose the value of 8 to be conservative.

3.2.3 Rate Adaptation

Since TROCs are measured at receivers upon packet losses, they indicate how much bandwidth a flow can get out of the fully loaded bottleneck, assuming congestion is the only reason for packet losses. The less it can get, the more congested the bottleneck is. Therefore, we choose one receiver with the lowest average TROC as the CR, and let the source only consider the feedbacks from that receiver for rate adaptation.

ESMCC is a *rate-based* scheme, using the policy of additive increase and multiplicative decrease (AIMD). As shown in Rate Increase part of Figure 7, if there are no feedbacks from the CR, the transmission rate is increased by $s\ /\ RTT$ per RTT, where s is the packet size; RTT is that between the source and the CR. If a feedback is received from the CR at time t_l, let the TROC in this feedback be $\overline{\mu}(t)$, we adjust the transmission rate to the minimum of *;* $\beta\overline{\mu}(t)$ and the current rate. Feedbacks from other non-CR receivers will be ignored, and at most one rate cut is allowed per RTT. This is shown in Rate Decrease part of Figure 8.

Thus, adaptation of the source rate $\lambda(t)$ is done according to the following AIMD-like method:

$$\lambda(t_1) = \begin{cases} \lambda(t_0) + s/RTT, & no \quad feedback \\ \min(\lambda(t_0), \beta\hat{\mu}(t_0)) & feeback \quad with \quad \hat{\mu}(t) \end{cases}$$

Where the feedback $\overline{\mu}(t)$ arrives at source between to and t_l, i.e. $t_0 < t \le t_1$.

The rate reduction factor β is an important parameter of ESMCC. The larger the β, the more aggressive is ESMCC. To keep ESMCC TCP-friendly, we will see that β must be at least 0.5. Moreover, the exact value of β depends on how ESMCC is implemented. According to the simulation and experiment results, we suggest $\beta =$ 0.65 for implementation on user level, and $\beta =$ 0.75 for implementation in system kernel. The reason is that, if ESMCC is implemented on user level, due to the coarseness of timers, its traffic is more bursty than that of TCP running in kernel. To cancel that effect, β should be set lower.

3.2.4 Update of Statistics

ESMCC source needs to maintain sets of statistics for the purposes of (i) estimating the RTT between the source and the CR, (ii) estimating response time of the CR during congestion epochs, and (iii) keeping track of the TROC of the CR. Flowchart of how these statistics are updated is shown in Update Statistics part of Figure 8. We now briefly describe how each of these sets of statistics is updated:

RTT Estimation: Unlike a NAK, which includes the sequence number of a lost packet, a feedback in ESMCC includes the sequence number of a packet upon the arrival of which packet losses are detected. The source calculates the difference between the sending time of this packet and the arriving time of this feedback to get a sample of RTT. By doing this, we avoid the unnecessary delay between the supposed arriving time of a lost packet and the time of its loss being detected. Nevertheless, since feedbacks are sent only when packet losses occur, RTT estimated by these feedbacks includes the maximum bottleneck queuing delay and thus is still the upper bound. On the other hand, ACKs as those in TCP mayor may not include bottleneck queuing delay. Therefore, on average, RTT estimated by ESMCC's feedbacks is larger than that by ACKs under the same situation. In fact, this is the reason why we set β to some value higher than 0.5.

ESMCC maintain the following two values regarding RTT: (i) $R\overline{T}T$, estimate of the RTT between the source and the CR, and (ii) $RTT_{max,}$ the maximum RTT estimate $R\overline{T}T$ that was ever seen by the source. As shown in Figure 8, *upon receipt of a feedback* from receiver ***i***, the source updates $R\overline{T}T$ and RTT_{max} when either (i) the receiver ***i*** is the CR or (ii) the feedback caused the CR to be changed. Notice that this method calculates the RTT only from the samples when congestion exists.

CR Response Time: Another statistic that ESMCC source needs is the time, *Tmax,* it would *maximally* take the current CR to respond during a congestion epoch. This is a crucial measure since it is used to determine whether or not the current CR is still active or not, as it can so happen that the CR may leave the system. The value of *Tmax* is composed of *E[T]* and T_σ which are average value of *T* and its deviation respectively. The composition we use is $T_{max} = E[t] + 8T_\sigma$, which means the source needs to measure and maintain the values of *E[T]* and T_σ. As it can be seen from Update Statistics part of Figure 8, the source updates *E[T]* and $T_\sigma$only *upon receipt of a feedback* from the current CR within the time period that started when the bottleneck is estimated to be fully loaded after a rate increase.

CR's TROC: As described in (2), average TROC is calculated by means of an *EWMA* function, which we represent as $\phi()$. In addition to average TROC, the source also maintains the deviation of TROC, $\overline{\sigma}$, for the current CR. CR's average TROC, $\overline{\mu}$, and deviation of CR's TROC, $\overline{\sigma}$, are crucial statistics since they represents the maximum possible transmission rate for the current session and are directly used for the process of CR selection. As shown in Figure 8, *upon receipt of a feedback* from receiver ***i***, the source updates $\overline{\mu}$ and $\overline{\sigma}$ when either (i) the receiver ***i*** is the CR or (ii) the feedback caused the CR to be changed.

3.2.5 Data Packet Handler

Receivers in ESMCC must be informed about the current value of CR's TROC, $\overline{\mu}$, and its deviation, $\overline{\sigma}$. In order to convey $\overline{\mu}$ and $\overline{\sigma}$ to the receivers, ESMCC source attaches them to the data packets. As shown in Figure 8, the source specifically sets $\overline{\mu} = 0$ and $\overline{\sigma} = \infty$ when the CR is Inactive mode. The purpose of this is to make the receivers send their current TROC values, so that a new CR can be elected.

Even though we have not implemented in the simulations of this dissertation, it is also possible to set $\overline{\mu}$ and $\overline{\sigma}$, so that only those receivers with TROC very close to the latest CR's TROC will send feedback. Such a strategy is particularly needed when the total number of receivers is too large.

3.3 Key Features of ESMCC

As we can see from the details above, ESMCC has the following features:

- 0(1) **Memory Complexity:** The amount of memory needed to maintain the state information at source and receivers is 0(1). That is, the number of states is constant and independent of the number of receivers in a multicast session.
- **Practical Operations:** Operations of source and receivers are all simple, without requiring intense computation. In particular, there is no need to do per-receiver RTT estimation.
- **Effective Feedback Suppression:** With our non-probabilistic-timer-based feedback suppression mechanism in place, the amount of feedbacks is independent othe total number of receivers.

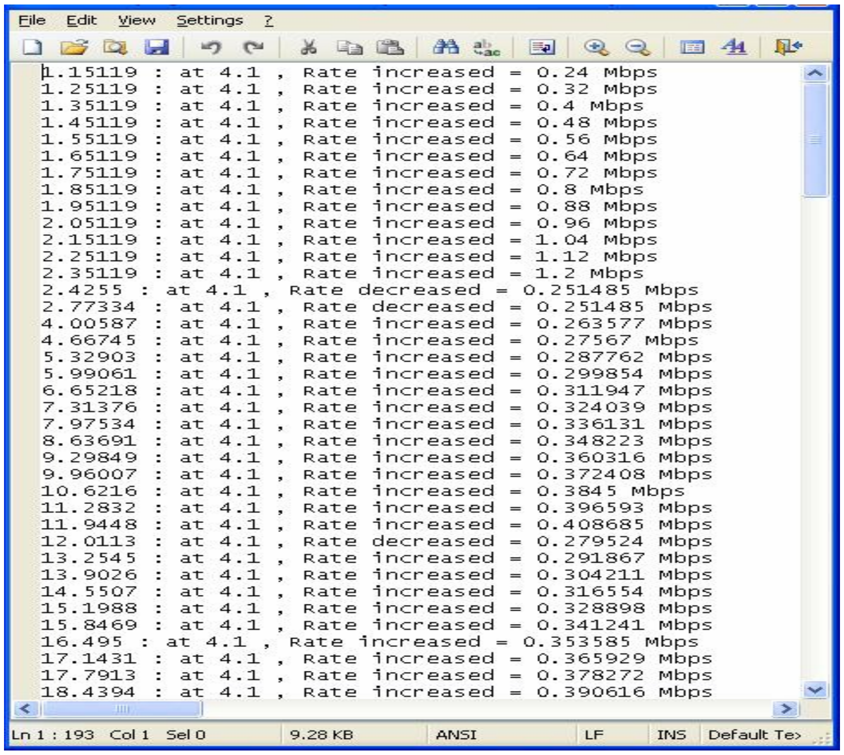
4) Simulations and Experiments

We have run simulations on *ns-2.28* to validate the performance of ERMCC. The *ns-2* simulations checked the TCP -Friendliness, Drop-to-zero avoidance, Effective feedback suppression.

We used a star topology to generate asynchronous and independent congestion on different paths. There are 33 ends nodes in the topology. Between each pair of source i and receiver i (i = 1 ... 16), there are one TCP flow and one single-receiver ERMCC flow. Furthermore, there is a multireceiver ERMCC flow from source 17 to all upto 33 receivers. Therefore, on a path between the router and any receiver, the multi-receiver ERMCC flow competes with a TCP flow and a single-receiver ERMCC flow.

We have randomly chosen a three node 4.1, 7.1 and 14.1 and draw a graph from the data sheet which is generated from the simulation. First of we draw a graph for the rate vs. time.

Snapshot of the Data Sheet for the Generation of Rate vs Time graph



```
1.15119 : at 4.1 , Rate increased = 0.24 Mbps
1.25119 : at 4.1 , Rate increased = 0.32 Mbps
1.35119 : at 4.1 , Rate increased = 0.4 Mbps
1.45119 : at 4.1 , Rate increased = 0.48 Mbps
1.55119 : at 4.1 , Rate increased = 0.56 Mbps
1.65119 : at 4.1 , Rate increased = 0.64 Mbps
1.75119 : at 4.1 , Rate increased = 0.72 Mbps
1.85119 : at 4.1 , Rate increased = 0.8 Mbps
1.95119 : at 4.1 , Rate increased = 0.88 Mbps
2.05119 : at 4.1 , Rate increased = 0.96 Mbps
2.15119 : at 4.1 , Rate increased = 1.04 Mbps
2.25119 : at 4.1 , Rate increased = 1.12 Mbps
2.35119 : at 4.1 , Rate increased = 1.2 Mbps
2.4255 : at 4.1 , Rate decreased = 0.251485 Mbps
2.77334 : at 4.1 , Rate decreased = 0.251485 Mbps
4.00587 : at 4.1 , Rate increased = 0.263577 Mbps
4.66745 : at 4.1 , Rate increased = 0.27567 Mbps
5.32903 : at 4.1 , Rate increased = 0.287762 Mbps
5.99061 : at 4.1 , Rate increased = 0.299854 Mbps
6.65218 : at 4.1 , Rate increased = 0.311947 Mbps
7.31376 : at 4.1 , Rate increased = 0.324039 Mbps
7.97534 : at 4.1 , Rate increased = 0.336131 Mbps
8.63691 : at 4.1 , Rate increased = 0.348223 Mbps
9.29849 : at 4.1 , Rate increased = 0.360316 Mbps
9.96007 : at 4.1 , Rate increased = 0.372408 Mbps
10.6216 : at 4.1 , Rate increased = 0.3845 Mbps
11.2832 : at 4.1 , Rate increased = 0.396593 Mbps
11.9448 : at 4.1 , Rate increased = 0.408685 Mbps
12.0113 : at 4.1 , Rate decreased = 0.279524 Mbps
13.2545 : at 4.1 , Rate increased = 0.291867 Mbps
13.9026 : at 4.1 , Rate increased = 0.304211 Mbps
14.5507 : at 4.1 , Rate increased = 0.316554 Mbps
15.1988 : at 4.1 , Rate increased = 0.328898 Mbps
15.8469 : at 4.1 , Rate increased = 0.341241 Mbps
16.495 : at 4.1 , Rate increased = 0.353585 Mbps
17.1431 : at 4.1 , Rate increased = 0.365929 Mbps
17.7913 : at 4.1 , Rate increased = 0.378272 Mbps
18.4394 : at 4.1 , Rate increased = 0.390616 Mbps
```

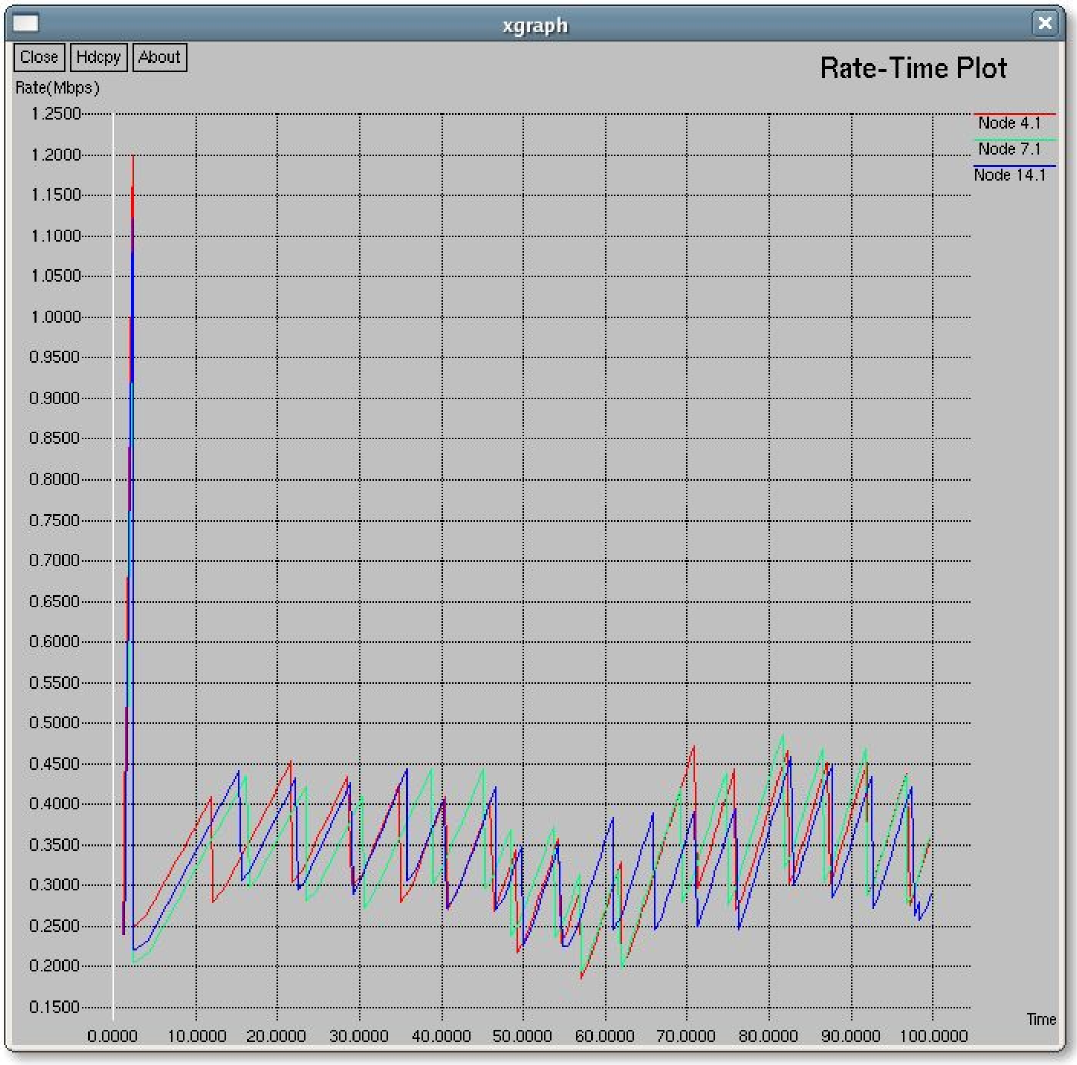
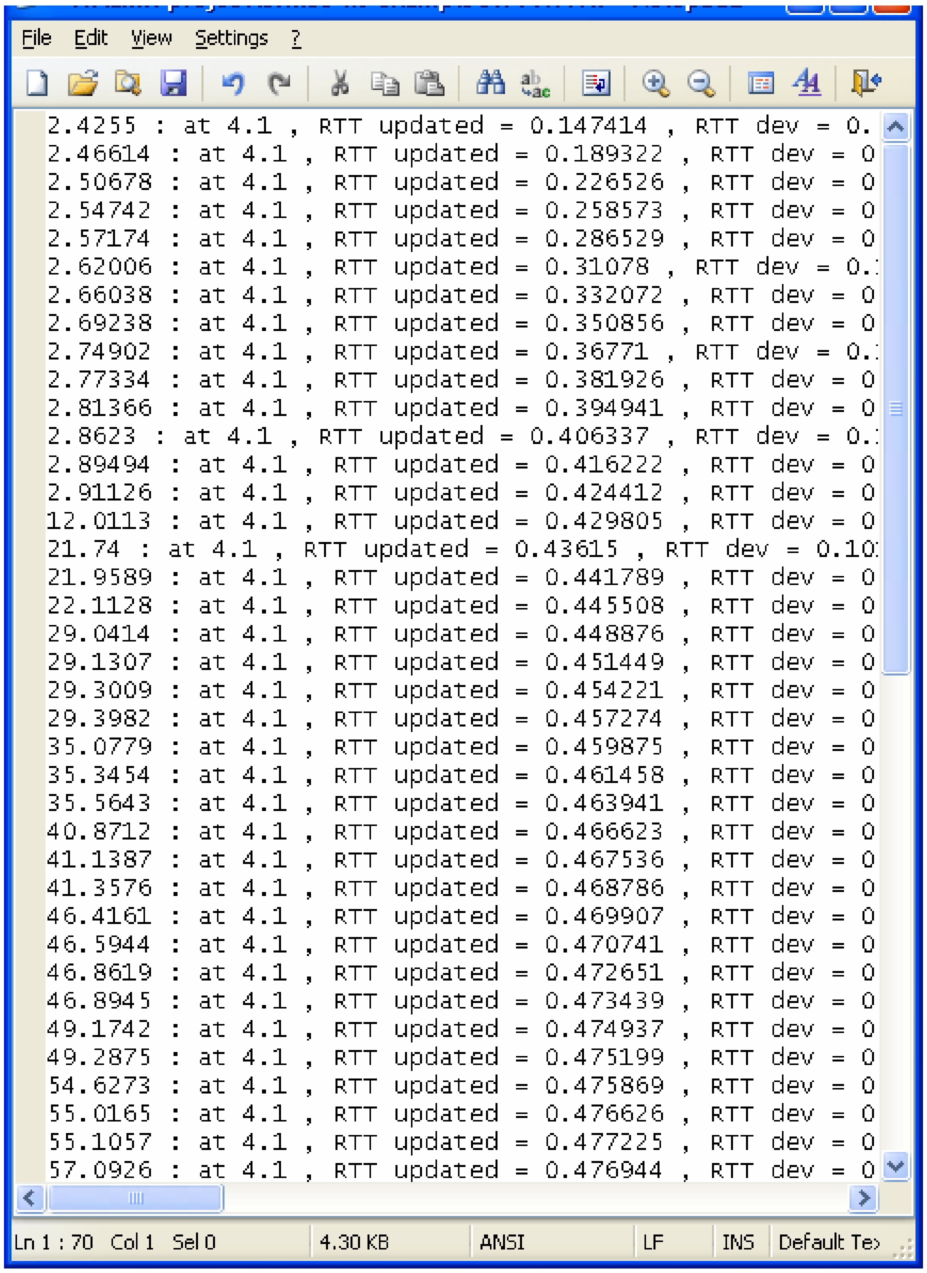


Figure 11: Graph plotted between Rate vs Time for the three nodes

This graph is constructed between the Rate vs Time for the three nodes. The rate is measured in Mbps and time in second. The rate is increased from the above given formula in the Rate increased module($\lambda \leftarrow \lambda + s / R\overline{T}T$) whenever the rate is increased from the $\lambda \geq \overline{\mu} + 4\overline{\sigma}$ the rate decresed occur. This graph shows that the TCP-friendliness maintained and avoids Drop-to-zero.

Snapshot of the Data Sheet for the Generation of RTT update vs Time graph



```
2.4255 : at 4.1 , RTT updated = 0.147414 , RTT dev = 0.
2.46614 : at 4.1 , RTT updated = 0.189322 , RTT dev = 0
2.50678 : at 4.1 , RTT updated = 0.226526 , RTT dev = 0
2.54742 : at 4.1 , RTT updated = 0.258573 , RTT dev = 0
2.57174 : at 4.1 , RTT updated = 0.286529 , RTT dev = 0
2.62006 : at 4.1 , RTT updated = 0.31078 , RTT dev = 0.
2.66038 : at 4.1 , RTT updated = 0.332072 , RTT dev = 0
2.69238 : at 4.1 , RTT updated = 0.350856 , RTT dev = 0
2.74902 : at 4.1 , RTT updated = 0.36771 , RTT dev = 0.
2.77334 : at 4.1 , RTT updated = 0.381926 , RTT dev = 0
2.81366 : at 4.1 , RTT updated = 0.394941 , RTT dev = 0
2.8623 : at 4.1 , RTT updated = 0.406337 , RTT dev = 0.
2.89494 : at 4.1 , RTT updated = 0.416222 , RTT dev = 0
2.91126 : at 4.1 , RTT updated = 0.424412 , RTT dev = 0
12.0113 : at 4.1 , RTT updated = 0.429805 , RTT dev = 0
21.74 : at 4.1 , RTT updated = 0.43615 , RTT dev = 0.10
21.9589 : at 4.1 , RTT updated = 0.441789 , RTT dev = 0
22.1128 : at 4.1 , RTT updated = 0.445508 , RTT dev = 0
29.0414 : at 4.1 , RTT updated = 0.448876 , RTT dev = 0
29.1307 : at 4.1 , RTT updated = 0.451449 , RTT dev = 0
29.3009 : at 4.1 , RTT updated = 0.454221 , RTT dev = 0
29.3982 : at 4.1 , RTT updated = 0.457274 , RTT dev = 0
35.0779 : at 4.1 , RTT updated = 0.459875 , RTT dev = 0
35.3454 : at 4.1 , RTT updated = 0.461458 , RTT dev = 0
35.5643 : at 4.1 , RTT updated = 0.463941 , RTT dev = 0
40.8712 : at 4.1 , RTT updated = 0.466623 , RTT dev = 0
41.1387 : at 4.1 , RTT updated = 0.467536 , RTT dev = 0
41.3576 : at 4.1 , RTT updated = 0.468786 , RTT dev = 0
46.4161 : at 4.1 , RTT updated = 0.469907 , RTT dev = 0
46.5944 : at 4.1 , RTT updated = 0.470741 , RTT dev = 0
46.8619 : at 4.1 , RTT updated = 0.472651 , RTT dev = 0
46.8945 : at 4.1 , RTT updated = 0.473439 , RTT dev = 0
49.1742 : at 4.1 , RTT updated = 0.474937 , RTT dev = 0
49.2875 : at 4.1 , RTT updated = 0.475199 , RTT dev = 0
54.6273 : at 4.1 , RTT updated = 0.475869 , RTT dev = 0
55.0165 : at 4.1 , RTT updated = 0.476626 , RTT dev = 0
55.1057 : at 4.1 , RTT updated = 0.477225 , RTT dev = 0
57.0926 : at 4.1 , RTT updated = 0.476944 , RTT dev = 0
```

This is the snapshot data sheet of the RTT update of the one of the node . With the help of these datasheet for the three different nodes we have created the graph between the RTT update and time .

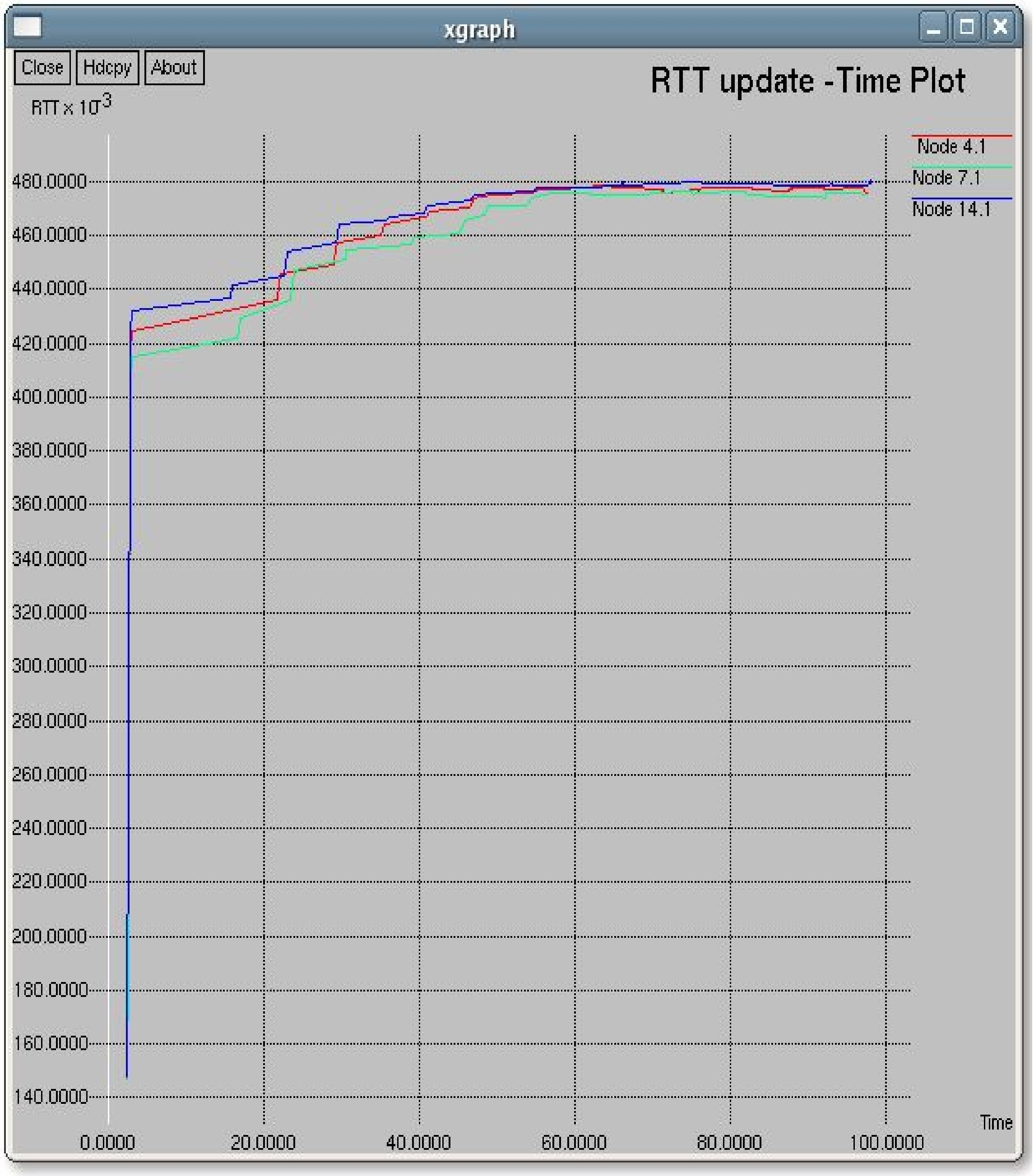
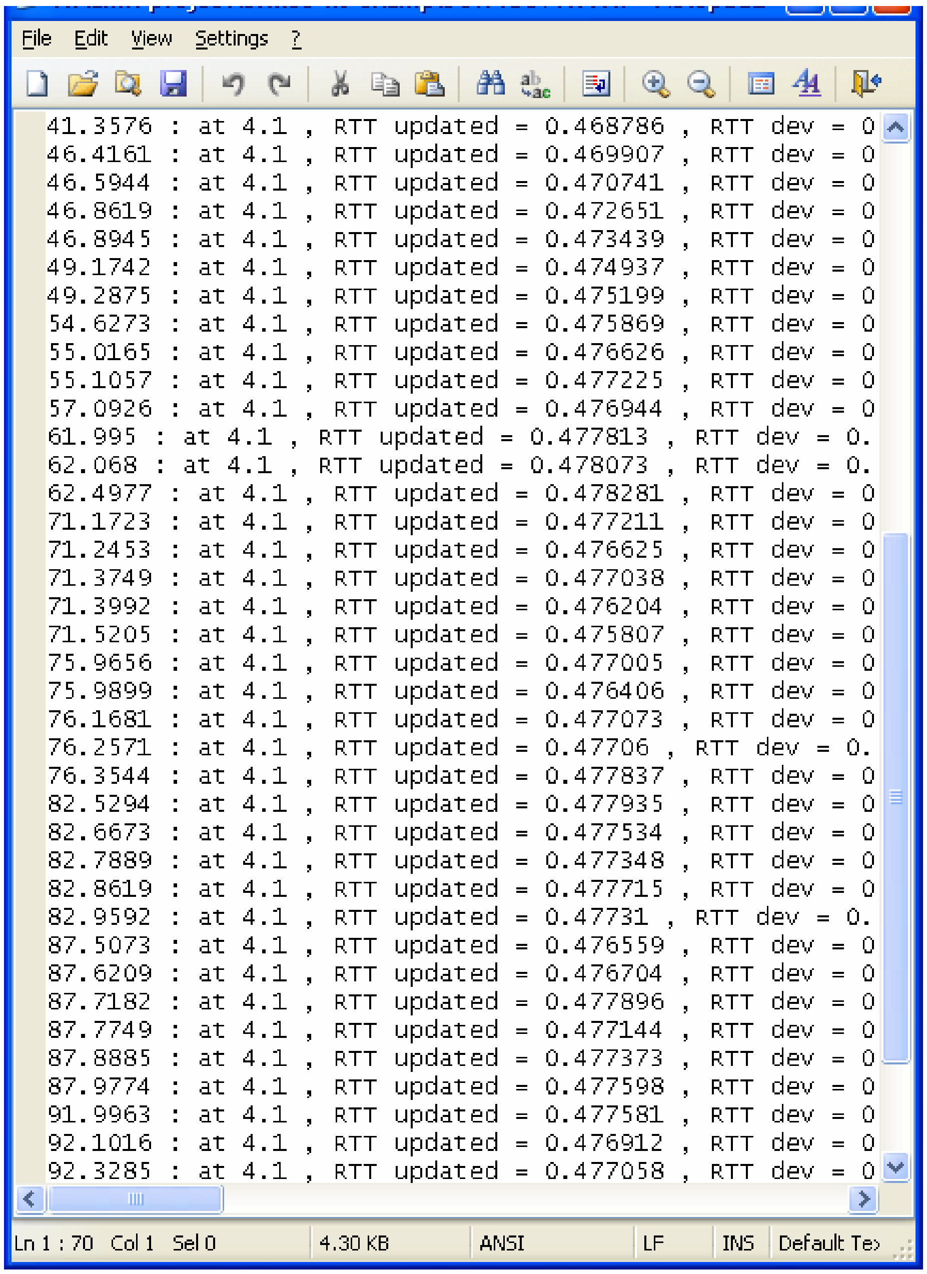


Figure 12: Graph plotted between RTT update vs Time for the three nodes

Snapshot of the Data Sheet for the Generation of RTT Deviation vs Time graph



```
41.3576 : at 4.1 , RTT updated = 0.468786 , RTT dev = 0
46.4161 : at 4.1 , RTT updated = 0.469907 , RTT dev = 0
46.5944 : at 4.1 , RTT updated = 0.470741 , RTT dev = 0
46.8619 : at 4.1 , RTT updated = 0.472651 , RTT dev = 0
46.8945 : at 4.1 , RTT updated = 0.473439 , RTT dev = 0
49.1742 : at 4.1 , RTT updated = 0.474937 , RTT dev = 0
49.2875 : at 4.1 , RTT updated = 0.475199 , RTT dev = 0
54.6273 : at 4.1 , RTT updated = 0.475869 , RTT dev = 0
55.0165 : at 4.1 , RTT updated = 0.476626 , RTT dev = 0
55.1057 : at 4.1 , RTT updated = 0.477225 , RTT dev = 0
57.0926 : at 4.1 , RTT updated = 0.476944 , RTT dev = 0
61.995 : at 4.1 , RTT updated = 0.477813 , RTT dev = 0.
62.068 : at 4.1 , RTT updated = 0.478073 , RTT dev = 0.
62.4977 : at 4.1 , RTT updated = 0.478281 , RTT dev = 0
71.1723 : at 4.1 , RTT updated = 0.477211 , RTT dev = 0
71.2453 : at 4.1 , RTT updated = 0.476625 , RTT dev = 0
71.3749 : at 4.1 , RTT updated = 0.477038 , RTT dev = 0
71.3992 : at 4.1 , RTT updated = 0.476204 , RTT dev = 0
71.5205 : at 4.1 , RTT updated = 0.475807 , RTT dev = 0
75.9656 : at 4.1 , RTT updated = 0.477005 , RTT dev = 0
75.9899 : at 4.1 , RTT updated = 0.476406 , RTT dev = 0
76.1681 : at 4.1 , RTT updated = 0.477073 , RTT dev = 0
76.2571 : at 4.1 , RTT updated = 0.47706 , RTT dev = 0.
76.3544 : at 4.1 , RTT updated = 0.477837 , RTT dev = 0
82.5294 : at 4.1 , RTT updated = 0.477935 , RTT dev = 0
82.6673 : at 4.1 , RTT updated = 0.477534 , RTT dev = 0
82.7889 : at 4.1 , RTT updated = 0.477348 , RTT dev = 0
82.8619 : at 4.1 , RTT updated = 0.477715 , RTT dev = 0
82.9592 : at 4.1 , RTT updated = 0.47731 , RTT dev = 0.
87.5073 : at 4.1 , RTT updated = 0.476559 , RTT dev = 0
87.6209 : at 4.1 , RTT updated = 0.476704 , RTT dev = 0
87.7182 : at 4.1 , RTT updated = 0.477896 , RTT dev = 0
87.7749 : at 4.1 , RTT updated = 0.477144 , RTT dev = 0
87.8885 : at 4.1 , RTT updated = 0.477373 , RTT dev = 0
87.9774 : at 4.1 , RTT updated = 0.477598 , RTT dev = 0
91.9963 : at 4.1 , RTT updated = 0.477581 , RTT dev = 0
92.1016 : at 4.1 , RTT updated = 0.476912 , RTT dev = 0
92.3285 : at 4.1 , RTT updated = 0.477058 , RTT dev = 0
```

This is the snapshot data sheet of the RTT deviation of the one of the node . With the help of these datasheet for the three different nodes we have created the graph between the RTT deviation and time .

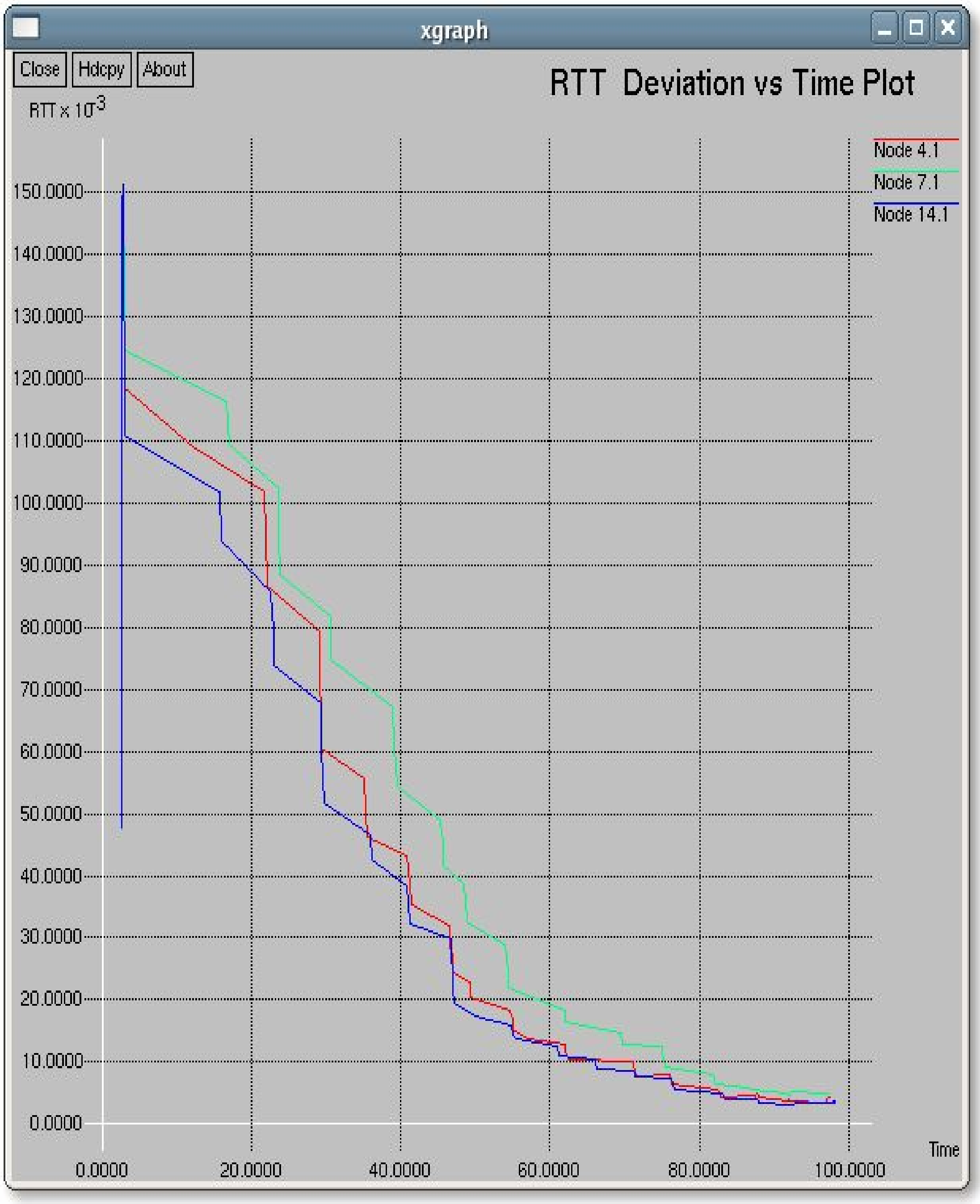
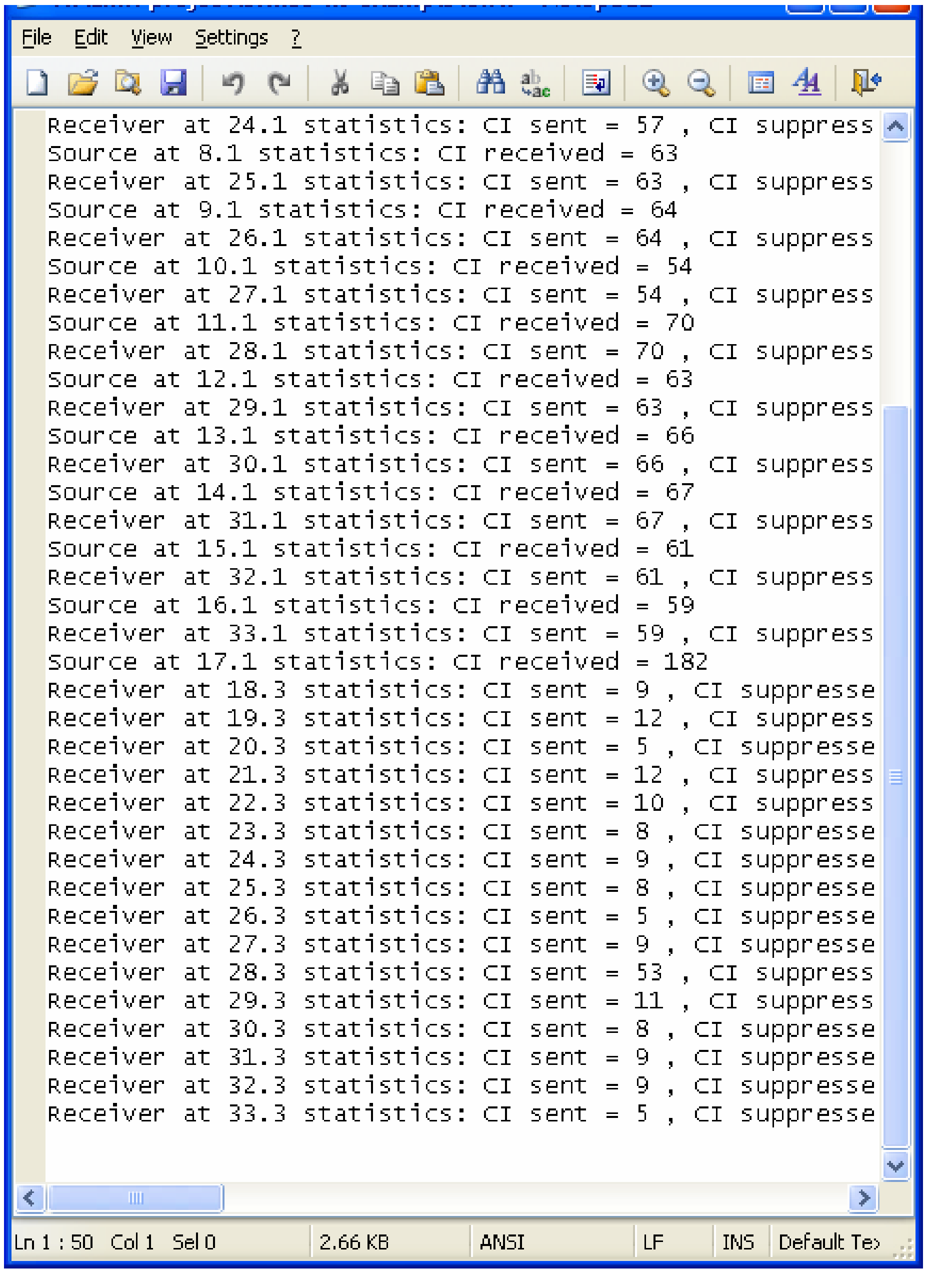


Figure 13: Graph plotted between RTT Deviation vs Time for the three nodes

Snapshot of the Data Sheet for the Generation of CI suppressed vs Receiver graph



```
Receiver at 24.1 statistics: CI sent = 57 , CI suppress
Source at 8.1 statistics: CI received = 63
Receiver at 25.1 statistics: CI sent = 63 , CI suppress
Source at 9.1 statistics: CI received = 64
Receiver at 26.1 statistics: CI sent = 64 , CI suppress
Source at 10.1 statistics: CI received = 54
Receiver at 27.1 statistics: CI sent = 54 , CI suppress
Source at 11.1 statistics: CI received = 70
Receiver at 28.1 statistics: CI sent = 70 , CI suppress
Source at 12.1 statistics: CI received = 63
Receiver at 29.1 statistics: CI sent = 63 , CI suppress
Source at 13.1 statistics: CI received = 66
Receiver at 30.1 statistics: CI sent = 66 , CI suppress
Source at 14.1 statistics: CI received = 67
Receiver at 31.1 statistics: CI sent = 67 , CI suppress
Source at 15.1 statistics: CI received = 61
Receiver at 32.1 statistics: CI sent = 61 , CI suppress
Source at 16.1 statistics: CI received = 59
Receiver at 33.1 statistics: CI sent = 59 , CI suppress
Source at 17.1 statistics: CI received = 182
Receiver at 18.3 statistics: CI sent = 9 , CI suppresse
Receiver at 19.3 statistics: CI sent = 12 , CI suppress
Receiver at 20.3 statistics: CI sent = 5 , CI suppresse
Receiver at 21.3 statistics: CI sent = 12 , CI suppress
Receiver at 22.3 statistics: CI sent = 10 , CI suppress
Receiver at 23.3 statistics: CI sent = 8 , CI suppresse
Receiver at 24.3 statistics: CI sent = 9 , CI suppresse
Receiver at 25.3 statistics: CI sent = 8 , CI suppresse
Receiver at 26.3 statistics: CI sent = 5 , CI suppresse
Receiver at 27.3 statistics: CI sent = 9 , CI suppresse
Receiver at 28.3 statistics: CI sent = 53 , CI suppress
Receiver at 29.3 statistics: CI sent = 11 , CI suppress
Receiver at 30.3 statistics: CI sent = 8 , CI suppresse
Receiver at 31.3 statistics: CI sent = 9 , CI suppresse
Receiver at 32.3 statistics: CI sent = 9 , CI suppresse
Receiver at 33.3 statistics: CI sent = 5 , CI suppresse
```

With the help of this datasheet we have created a graph between the receiver and CI suppressed and also created a graph between the receiver and the CI sent.

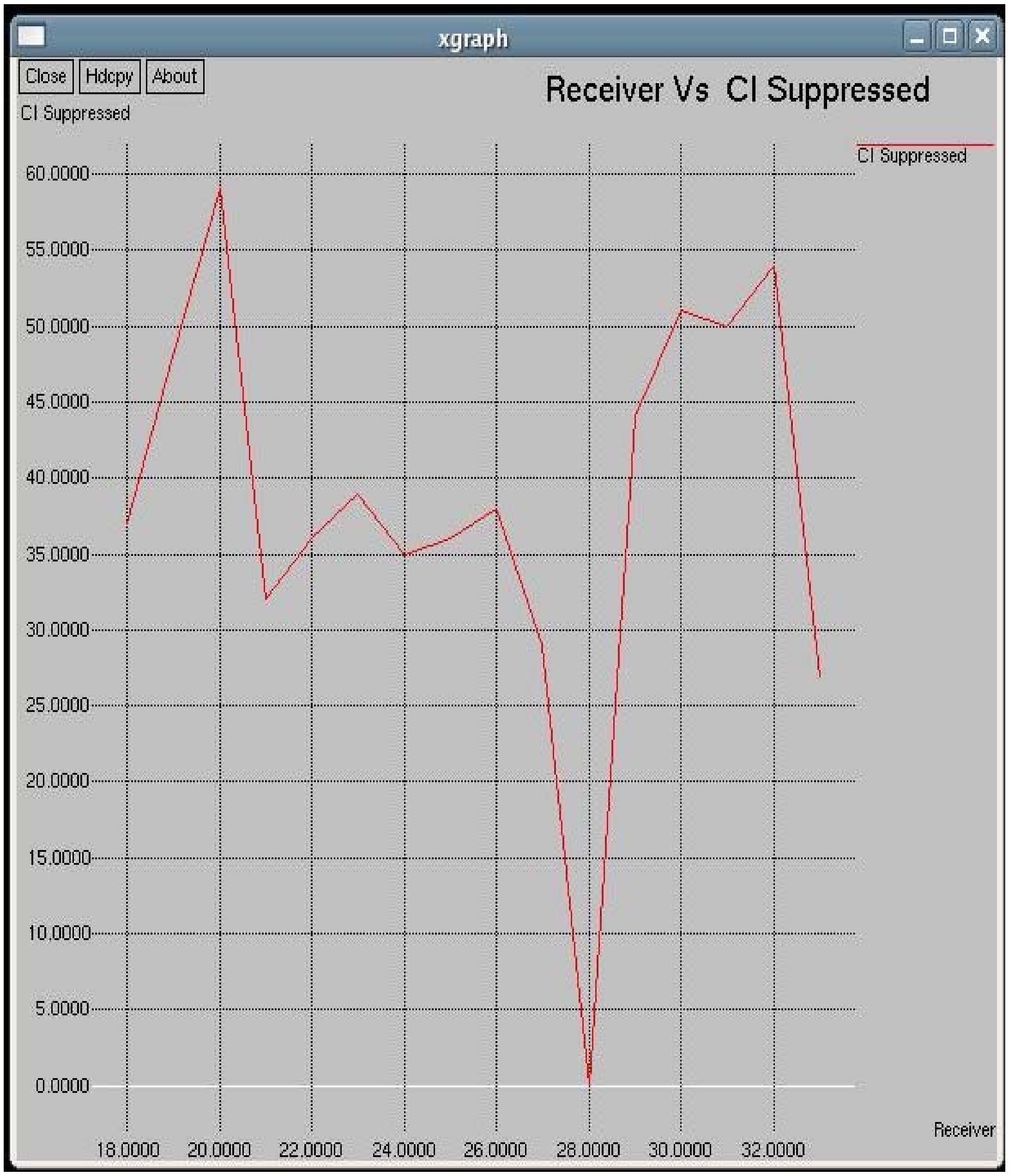


Figure 14: Graph plotted between Receiver vs CI suppressed

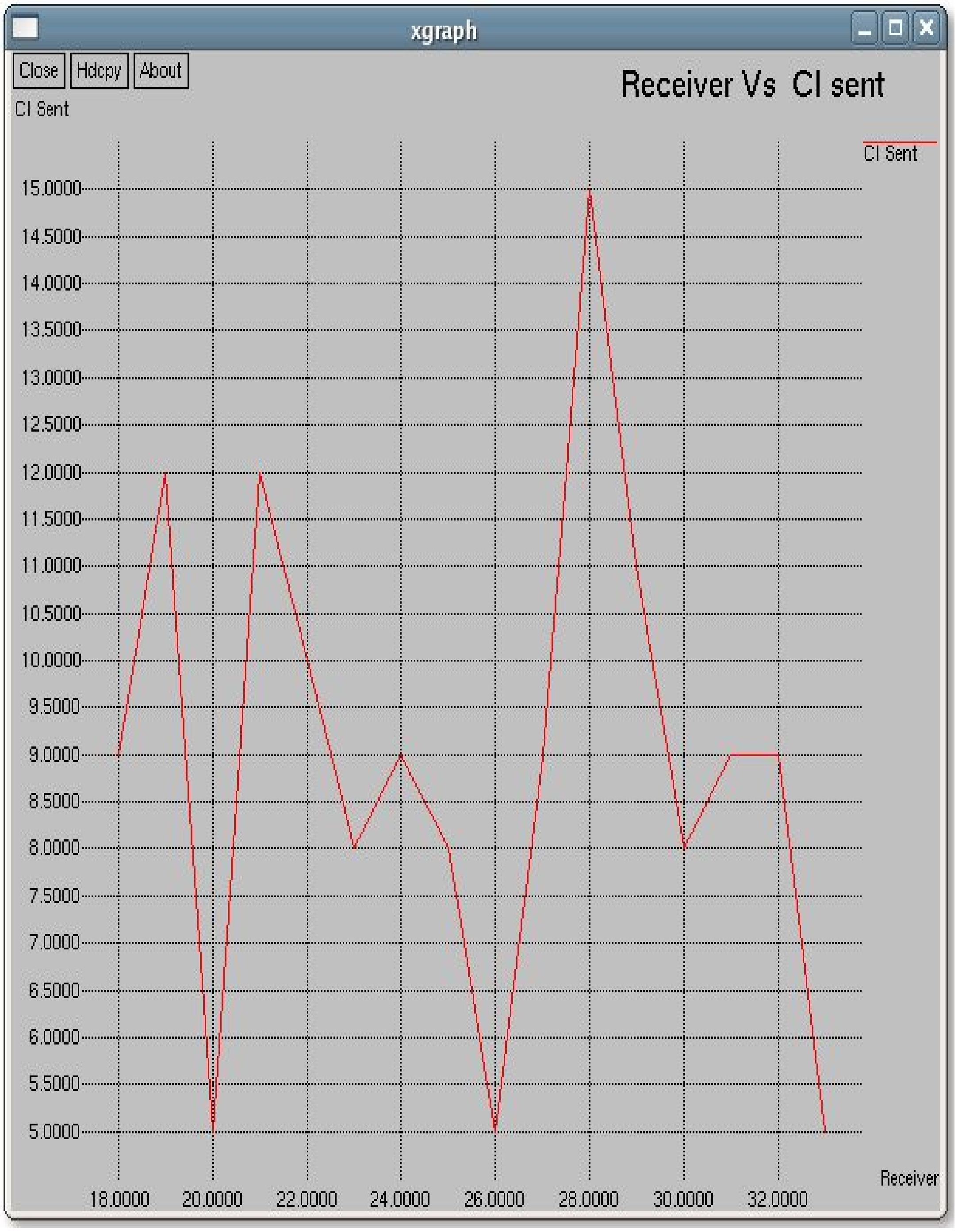


Figure 15: Graph plotted between Receiver vs CI sent

Feedback Suppression

To check the effectiveness of the feedback suppression mechanism in ESMCC, we refer back to the simulation of TCP-friendliness and drop-to-zero avoidance. In this simulations, the average total number of feedbacks sent by all receivers is 182, the average total number of suppressed feedbacks is 615. The average number of feedbacks would have been sent by a receiver if without suppression, is (615 + 182)/32 ~ 24, realistic measurement error can lead to a little bit more feedbacks. The high ratio of feedbacks suppressed, 615/(615 + 182) x 100% ~ 77.7%, shows that *our feedback suppression is very effective.*

5) Conclusion

In this dissertation, we have proposed a enhancement of the single-rate multicast congestion control scheme, which uses a conventional concept of representative named *Congestion Representative (CR).* However, by leveraging a new metric *TROC,* the ESMCC scheme is capable of effectively addressing the problems of TCP friendliness, drop-to-zero, slowest receiver tracking and feedback suppression. The states maintained by source and receivers are 0(1); operations of source and receivers are all simple without requiring intense computation. In particular there is no need to measure RTTs between all receivers and the source. ESMCC also shows that non-probabilistic-timer-based feedback suppression is highly effective. To confirm the performance of ESMCC, we have not only provided theoretical analysis, but also performed simulations. Both simulation and implementation results show ESMCC's excellent performance.

We believe that further studies of ESMCC-like schemes will benefit the area of multicast congestion control. A point that deserves further investigation is the EWMA smoothing technique used at various places of the scheme. Particularly, it is worthwhile to study averaging techniques that can use the timestamp differences of arriving data packets at the receiver. Also, adaptive tuning of various parameters (e.g. less that 4σ in determining slowest receiver with its average TROC) can provide incremental improvements to ESMCC.

6)References:

[1] D. DeLucia, K. Obraczka, A multicast congestion control mechanism using representatives, in: Proceedings of IEEE ISCC, 1998.

[2] L. Rizzo, Pgmcc: A tcp-friendly single-rate multicast congestion control scheme, in: Proceedings of ACM SIGCOMM, 2000.

[3] J. Widmer, M. Handley, Extending equation-based congestion control to multicast applications, in: Proceedings of ACM SIGCOMM, 2001.

[4] J. Macker, R. Adamson, A tcp friendly, rate-based mechanism for nack-oriented reliable multicast congestion control, in: Proceedings of IEEE GLOBECOM, 2001.

[5] P. Thapliyal, Sidhartha, J. Li, S. Kalyanaraman, Le-sbcc: Loss-event oriented source-based multicast congestion control, Multimedia Tools and Applications 17 (2-3) (2002) 257-294.

[6] J. Byers, M. Luby, M. Mitzenmacher, Fine-grained layered multicast, m: Proceedings of IEEE INFOCOM, 2001.

[7] J. Byers, G. Kwon, Stair: Practical aimd multirate multicast congestion control, in: Proceedings of NGC, 2001.

[8] J. C. Lin, S. Paul, Rmtp: A reliable multicast transport protocol, in: Proceedings of IEEE INFOCOM, 1996.

[9] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, Modeling tcp throughput: A simple model and its empirical validation, in: Proceedings of ACM SIGCOMM, 1998.

[10] M. Mathis, J. Semke, J.Mahdavi, T.Ott, The macroscopic behavior of the tcp congestion avoidance algorithm, ACM Computer Communications Review 27(3).

[11] T. T. Fuhrmann, J. Widmer, On the scaling of feedback algorithms for very large multicast groups, Computer Communications 24 (5) (2001) 539-547.

[12] J. Nonnenmacher, E. W. Biersack, Scalable feedback for large groups, IEEE ACM Transactions on Networking 7 (3) (1999) 375-386.

[13] Van Jacobson congestion avoidance and control ACM SIGCOMM 88 pages 273-288 1988.

[14] P.B. Danzing. Optimally selecting the parameter of adaptive Backoff algorithms for Computer Networks and multiprocessors.

[15]B.Whetten J.Conlan, "A rate based congestion control scheme for reliable multicast", Technical white paper Global cast communication, Oct 1998.

[16]J.W.Byers M.Luby M.Mitzenmacher A.Rege," A digital fountain approach to reliable distribution of bulk data", ACM SIGCOMM 98, Vancouver, CA, sep1998.

7) Appendix

A) Algorithm

A.l Operations at Source

Some of the following operations take place when either a feedback packet from a receiver r is received, or an RTT time period has been completed:

Variables:

r	: The receiver sending the received feedback
λ	: Current transmission rate at the source
Ω_r	: Throughput rate at congestion (TROC) in the received feedback from r
$\overline{\mu},$	: Average TROC of the CR
$\overline{\sigma}$	: Deviation of TROC of the CR
s	: Packet size
RTT_{max}	: Maximum RTT
$R\overline{T}T$	: RTT between the source and the CR
T	: CR response time when the bottleneck is fully loaded
$E[T]$	: Average of T
T_σ	: Deviation of T
cr_valid	: Indicates whether the CR is valid
cr_response_timer	: Indicates whether the bottleneck is estimated to be full
t_o	: The estimated time bottleneck started to fill up
t	: Current time

```
Initialization:

  $cr\_valid$ = false
  $RTT_{max}$ = 0
  to = 0
  $cr\_response\_timer$ = false
```

```
Event every $\overline{RTT}$:
    if there is no rate reduction within the recent $\overline{RTT}$ then
        $\lambda \leftarrow \lambda + s/\overline{RTT}$
        if $\lambda \geq \overline{\mu} + 4\overline{\sigma}$ and cr_response_timer is false then
            $t_0 \leftarrow t$
            cr_response_timer $\leftarrow$ true
        endif
    endif
    if $t - t_0 \geq E[T] + 8T_\sigma$ then
        cr_valid $\leftarrow$ false
        cr_response_timer $\leftarrow$ false
    endif

Send packet:
    if cr_valid is true then
        Send a packet with real $\overline{\mu}$ and $\overline{\sigma}$
endif
```

Subroutine : CutRate()

```
if $\lambda$ has not been cut within the most recent $R\overline{T}T$ then
        $\lambda \leftarrow \min(\lambda, 0.75\Omega_r)$
        cr_valid$\leftarrow$ true
        cr_response_timer $\leftarrow$ false
endif
```

Subroutine:UpdateStats()

```
    Update $\overline{\mu}$ and $\overline{\sigma}$ with $\Omega_r$
    Update $R\overline{T}T$ with $RTT_r$
    if $RTT_{max} < R\overline{T}T$ then
        $RTT_{max} \leftarrow R\overline{T}T$
    Endif
```

```
Event upon receipt of feedback from r:
  if r is *CR* then
      if *cr_response_timer* is true then
         Update E[T] and $T_\sigma$ with $(t-t_0)$
      endif
      do UpdateStats()
      do CutRate()
      return
  endif
```

```
/* The feedback is NOT from CR*/
   if *cr_valid* is false then
      Choose r as the CR
      Start CR grace period as 2 $RTT_{max}$
  else if *In CR grace period* then
        if $RTT_r > R\overline{T}T$ then
           Choose r as the CR
```

```
          Endif

/* NOT in longer RTT period */
     else if $\Omega_r < \overline{\mu} - \overline{\sigma}$ then
             Choose r as the CR
   endif



if CR has been changed at the receipt of this feedback then
        do UpdateStats()
        do CutRate()
endif
```

A.2 Operations at Receiver i

The following operations take place when a data packet is received at receiver i:

Variables:

Ω_i : A throughput rate on congestion (TROC) sample

μ_i : Average TROC of this receiver

$\overline{\mu}$: Average TROC of the CR

$\overline{\sigma}$: Deviation of TROC at the CR

Event upon receipt of a packet:

```
if $\overline{\mu}$ *and* $\overline{\sigma}$ *has been changed* then
    Update the local copy of $\overline{\mu}$, and $\overline{\sigma}$
endif
if *This packet indicates packet losses* then
    Measure $\Omega_i$ and update $\mu_i$
  if $\overline{\mu}$ *and* $\overline{\sigma}$ *are invalid* or $\mu_i < \overline{\mu} - \overline{\sigma}$ then
      Send a feedback to the source
  endif
endif
```

B) Coding

This program is used to generate the data sheet of the ESMCC .In this program we update statistics, also cut the rate when the rate is increased beyond the threshold. There is one more functioning which is perform by this program is up update of the RTT.

```
#include <stdlib.h>
#include <sys/types.h>
#include <math.h>
#include <assert.h>
#include <float.h>
#include <iostream.h>
#include "Esmcc.h"

int hdr_Esmcc::offset_;
int hdr_Esmcc_ci::offset_;

//operator overloading to be used to enhance the funcitionaliy of operator


ostream & operator<< (ostream & o, ns_addr_t & nsAddr) //
  o << nsAddr.addr_ << '.' << nsAddr.port_;
  return o;
}

int operator!= (ns_addr_t a, ns_addr_t b)
{
  return (a.addr_ != b.addr_ || a.port_ != b.port_);
}

int SqnGT (unsigned int s1, unsigned int s2) {
  return (((int) ((s1 > s2) ? (s1 - s2) : (s2 - s1))) >= 0) ?
    (s1 > s2) : (!(s1 > s2));
}
int SqnGE (unsigned int s1, unsigned int s2) {
  return (((int) ((s1 > s2) ? (s1 - s2) : (s2 - s1))) >= 0) ?
    (s1 >= s2) : (!(s1 >= s2));
}
int SqnLT (unsigned int s1, unsigned int s2) {
  return (((int) ((s1 > s2) ? (s1 - s2) : (s2 - s1))) >= 0) ?
    (s1 < s2) : (!(s1 < s2));
}
int SqnLE (unsigned int s1, unsigned int s2) {
  return (((int) ((s1 > s2) ? (s1 - s2) : (s2 - s1))) >= 0) ?
    (s1 <= s2) : (!(s1 <= s2));
}
```

```
//This is used to provide the connectivity  for the NS-2

static class ESMCCHeaderClass : public PacketHeaderClass {
public:
  ESMCCHeaderClass() : PacketHeaderClass("PacketHeader/ESMCC",
        sizeof(hdr_Esmcc)) {
     bind_offset(&hdr_Esmcc::offset_);
  }
} class_Esmcchdr;

static class ESMCC_CIHeaderClass : public PacketHeaderClass {
public:
  ESMCC_CIHeaderClass() : PacketHeaderClass("PacketHeader/ESMCC_CI",
        sizeof(hdr_Esmcc_ci)) {
     bind_offset(&hdr_Esmcc_ci::offset_);
  }
} class_Esmcc_cihdr;

static class EsmccClass : public TclClass {
public:
    EsmccClass() : TclClass("Agent/ESMCC") {}
    TclObject* create(int, const char*const*) {
       return (new EsmccAgent());
    }
} class_Esmcc;

static class EsmccSinkClass : public TclClass {
public:
    EsmccSinkClass() : TclClass("Agent/ESMCCSink") {}
    TclObject* create(int, const char*const*) {
        return (new EsmccSinkAgent());
    }
} class_EsmccSink;


/**********************************
 ESMCC  T I M E R
 **********************************/


void EsmccTimer::expire (Event *)
{
  agent_->TimeOut (timerType_);
}


/**********************************
 ESMCC  S O U R C E
 **********************************/


EsmccAgent::EsmccAgent() : Agent(PT_ESMCC),
  rateIncrTimer_ (this, ESMCC_RATE_INCR_TIMER),
  congEpochTimer_ (this, ESMCC_CONG_EPOCH_TIMER),
  sendPktTimer_ (this, ESMCC_SEND_PKT_TIMER),
  crGracePeriodTimer_ (this, ESMCC_CR_GRACE_PERIOD_TIMER),
  crRespTimer_ (this, ESMCC_CR_RESPONSE_TIMER)
{
  bind ("packetSize_", &size_);
  bind ("minRate_", &minRate_);
  bind ("initRate_", &initRate_);
  bind ("initRtt_", &initRtt_);
}

int EsmccAgent::command(int argc, const char*const* argv) {
  if (argc == 2) {
    if (strcmp (argv[1], "start") == 0) {
      Start();
      return TCL_OK;
    }
    if (strcmp (argv[1], "stop") == 0) {
      Stop ();
      return TCL_OK;
    }
    if (strcmp (argv[1], "print-statistics") == 0) {
      PrintStat ();
      return TCL_OK;
    }
  }
  return (Agent::command (argc, argv));
}


void EsmccAgent::PrintStat ()
{
  /* Statistic */
  cout << "Source at " << here_ << " statistics: CI received = "
    << ciRcvd_ << '\n';
}


void EsmccAgent::Start ()
{
  crGone_ = 1;
  cr_ = here_;
  sqn_ = 0;
  rate_ = initRate_;
  rttAvg_ = initRtt_;
  rttDev_ = 0;
  maxRtt_ = 0;
  crTracAvg_ = -1;
  crTracDev_ = 0;
  crChkBegTime_ = -1;
  crRespTimeAvg_ = 100; /* Set to large so that it won't time out */
  crRespTimeDev_ = -1;
  newCrRtt_ = -1;
  active_ = 1;
  ciRcvd_ = 0;

  SendPkt ();
  sendPktTimer_.resched (size_ / rate_);
  rateIncrTimer_.resched (rttAvg_);
}


void EsmccAgent::Stop ()
{
  active_ = 0;

  rateIncrTimer_.force_cancel ();
  congEpochTimer_.force_cancel ();
  sendPktTimer_.force_cancel ();
  crGracePeriodTimer_.force_cancel ();
  crRespTimer_.force_cancel ();
}


void EsmccAgent::SendPkt ()
{
  if (! active_) return;

  double now = Scheduler::instance().clock();
  Packet * p = allocpkt ();
  hdr_cmn * ch = HDR_CMN (p);
  hdr_Esmcc * oh = hdr_Esmcc::access (p);
  oh->cr_ = cr_;
  oh->timestamp_ = now;
  oh->sqn_ = sqn_ ++;
  oh->crTracAvg_ = crGone_ ? -1 : crTracAvg_;
  oh->crTracDev_ = crTracDev_;

  /* For measuring realistic throughput rate */
  hdr_ip * ih = HDR_IP (p);
  static p_info pinfo;
  cout << now << " : " << here_ << " -> " << ih->dst_
    << " , " << pinfo.name (ch->ptype ())
    << " , size = "  << ch->size_;

  #ifdef ESMCC_DATA_PKT_DEBUG
  cout << " , sqn = " << oh->sqn_;
  #endif

  cout  << '\n';

  send (p, 0);
}


void EsmccAgent::recv (Packet * p, Handler *)
```

```
{
 if (! active_) {
  Packet::free (p);
  return;
 }

 ++ ciRcvd_;

 hdr_Esmcc_ci * och = hdr_Esmcc_ci::access (p);
 double err, now = Scheduler::instance().clock();
 double rttSample = now - och->timestampEcho_;

 int cutRate = 0, changeCr = 0;
 int reason;   /* for debugging */

 if (maxRtt_ < rttSample) maxRtt_ = rttSample;

 #ifdef ESMCC_CI_DEBUG
 cout << now
  << " : at " << here_
  << " CI arrived from " << och->ciSrc_
  << " , CR = " << cr_
  << '\n';
 #endif

 #ifdef ESMCC_RTT_DEBUG
 cout << now
  << " : at " << here_
  << " RTT sample from " << och->ciSrc_
  << " = " << rttSample
  << '\n';
 #endif

 /*
  * Fileter CIs, update CR / cut rate if necessary
  */
 do {
  /* Initialization or CCI is gone. */
  if (crGone_ && och->ciSrc_ != cr_) {
   reason = 1;
   changeCr = 1;
   break;
  }

  /* CI is from CR */
  if (och->ciSrc_ == cr_) {
   crGone_ = 0;
   err = och->trac_ - crTracAvg_;
   crTracAvg_ += TRAC_EWMA_FACTOR * err;
   crTracDev_ += TRAC_EWMA_FACTOR * (fabs (err) - crTracDev_);
   cutRate = 1;
   UpdateRtt (rttSample);
```

```
      if (crChkBegTime_ < 0) break;

      if (crRespTimeDev_ < 0) {
        crRespTimeAvg_ = now - crChkBegTime_;
        crRespTimeDev_ = 0;
      }
      else {
        err = (now - crChkBegTime_) - crRespTimeAvg_;
        crRespTimeAvg_ += 0.125 * err;
        crRespTimeDev_ += 0.125 * (fabs (err) - crRespTimeDev_);
      }
      crChkBegTime_ = -1;
      if (crRespTimer_.status () == TIMER_PENDING) crRespTimer_.cancel ();

      break;
    }

    if (crGracePeriodTimer_.status () == TIMER_PENDING) {
      if (rttSample + SMALL_FLOAT < rttAvg_ + 2 * rttDev_) break;
      changeCr = 1;
      reason = 2;
      break;
    }

    if (och->trac_ + SMALL_FLOAT
        < crTracAvg_ - crTracDev_ / COMPARE_TRAC_FACTOR) {
      changeCr = 1;
      reason = 3;
      break;
    }
  } while (0);

  /*
   * Update CR
   */
  if (changeCr) {
    if (crTracAvg_ < 0) {
      cout << now << " : at " << here_
        << " CR is initialized as " << och->ciSrc_ << '\n';
    }
    else {
      cout << now << " : at " << here_
        << " CR is changed from " << cr_ << " to " << och->ciSrc_
        << ". ";
      #ifndef ESMCC_CR_DEBUG
      cout << '\n';
      #else
      switch (reason) {
      case 1:
        cout << "Current CR is inactive.\n";
        break;
      case 2:
        cout << "CI with larger RTT sample within grace period."
        << now - och->timestampEcho_
        << " > " << rttAvg_ << " + 2 * " << rttDev_ << " = "
        << rttAvg_ + 2 * rttDev_<< '\n';
      break;
    case 3:
      cout << "New lower average TRAC. "
        << och->trac_ / 125000
        << " < " << crTracAvg_ / 125000 << " - " << crTracDev_ / 125000
        << " / " << COMPARE_TRAC_FACTOR << " = "
        << (crTracAvg_ - crTracDev_ / COMPARE_TRAC_FACTOR) / 125000 << '\n';
      break;
    }
    #endif
  }

  if (crChkBegTime_ > 0 && crGracePeriodTimer_.status () == TIMER_IDLE) {
    if (crRespTimeDev_ < 0) {
      crRespTimeAvg_ = now - crChkBegTime_;
      crRespTimeDev_ = 0;
    }
    else {
      err = (now - crChkBegTime_) - crRespTimeAvg_;
      crRespTimeAvg_ += 0.125 * err;
      crRespTimeDev_ += 0.125 * (fabs (err) - crRespTimeDev_);
    }
  }

  crChkBegTime_ = -1;
  if (crRespTimer_.status () == TIMER_PENDING) {
    crRespTimer_.cancel ();
  }

  UpdateRtt (newCrRtt_ = rttSample);
  if (crGracePeriodTimer_.status () == TIMER_IDLE) {
    crGracePeriodTimer_.resched (2 * maxRtt_);
  }
  crGone_ = 0;
  cr_ =  och->ciSrc_;
  cutRate = 1;
  crTracAvg_ = och->trac_;
}

if (congEpochTimer_.status () == TIMER_PENDING) {
  cutRate = 0;
}

if (cutRate) {
  if (rate_ > 0.75 * och->trac_) rate_ = 0.75 * och->trac_;
  if (rate_ < minRate_) rate_ = minRate_;
  congEpochTimer_.resched (rttAvg_ + 4 * rttDev_);
  /* For rate tracing */
  cout << now
    << " : at " << here_
```

```
      << " , Rate decreased = " << rate_ / 125000 << " Mbps\n";
  }
  Packet::free (p);
}


void EsmccAgent::TimeOut (int type)
{
  double srtt, now, oldRate;

  switch (type) {
  case ESMCC_RATE_INCR_TIMER:
    rateIncrTimer_.resched (srtt = rttAvg_ + 2 * rttDev_);
    if (congEpochTimer_.status () == TIMER_PENDING) break;
    oldRate = rate_;
    rate_ += size_ / srtt;

    now = Scheduler::instance().clock();
    if (crTracAvg_ > 0
        && oldRate < crTracAvg_ + 4 * crTracDev_
        && rate_ >= crTracAvg_ + 4 * crTracDev_
        && crChkBegTime_ < 0) {
      crChkBegTime_ = now;
      crRespTimer_.resched (crRespTimeAvg_ + 8 * crRespTimeDev_);
      #ifdef ESMCC_CR_DEBUG
      cout << now
        << " : at " << here_
        << " CR check starts. "
        << " CR resp time avg = " << crRespTimeAvg_
        << " , dev = " << crRespTimeDev_
        << " , a + 8d = " << crRespTimeAvg_ + 8 * crRespTimeDev_
        << '\n';
      #endif
    }

    /* For rate tracing */
    cout << now
      << " : at " << here_
      << " , Rate increased = " << rate_ / 125000 << " Mbps\n";
    break;
  case ESMCC_CONG_EPOCH_TIMER:
    break;
  case ESMCC_SEND_PKT_TIMER:
    SendPkt ();
    sendPktTimer_.resched (size_ / rate_);
    break;
  case ESMCC_CR_GRACE_PERIOD_TIMER:
    break;
  case ESMCC_CR_RESPONSE_TIMER:
    #ifdef ESMCC_CR_DEBUG
    cout << Scheduler::instance().clock()
      << " : at " << here_
      << " CR is absent.\n";
    #endif
    crGone_ = 1;
    break;
  default:
    cout << "Unknow ESMCC timer.\n";
    cout.flush ();
    abort ();
  };
}


void EsmccAgent::UpdateRtt (double sample)
{
  double err = sample - rttAvg_;
  rttAvg_ += 0.125 * err;
  rttDev_ += 0.125 * (fabs (err) - rttDev_);

  /* For RTT tracing */
  cout << Scheduler::instance().clock()
    << " : at " << here_
    << " , RTT updated = " << rttAvg_
    << " , RTT dev = " << rttDev_
    << '\n';
}


/**********************************
 ES M C C  R E C E I V E R
 **********************************/


EsmccSinkAgent::EsmccSinkAgent() : Agent(PT_ESMCC_CI)
{
  bind("packetSize_", &size_);
  ciSent_ = ciSupp_ = 0;
  rateSmoother_ = NULL;
}


EsmccSinkAgent::~EsmccSinkAgent ()
{
  if (rateSmoother_ != NULL) delete rateSmoother_;
}


int EsmccSinkAgent::command(int argc, const char*const* argv) {
  if (argc == 2) {
    if (strcmp (argv[1], "start") == 0) {
      Start();
      return TCL_OK;
    }
    if (strcmp (argv[1], "stop") == 0) {
      Stop ();
      return TCL_OK;
    }
    if (strcmp (argv[1], "print-statistics") == 0) {
      PrintStat ();
      return TCL_OK;
    }
  }
  return (Agent::command (argc, argv));
}


void EsmccSinkAgent::Start ()
{
  cr_ = here_;
  rate_ = tracAvg_ = crTracAvg_ = lastPktTime_ = -1;
  crTracDev_ = 0;
  lastCont_ = 0;
  rateSmoother_ = new RateSmoother (RATE_SMOOTH_TIME);
}


void EsmccSinkAgent::Stop ()
{
  delete rateSmoother_;
  rateSmoother_ = NULL;
}


void EsmccSinkAgent::PrintStat ()
{
  /* Statistic */
  cout << "Receiver at " << here_ << " statistics: CI sent = "
    << ciSent_ << " , CI suppressed = " << ciSupp_ << '\n';
}


void EsmccSinkAgent::recv (Packet * p, Handler * h)
{
  double now = Scheduler::instance().clock(), err;
  hdr_cmn * ch = hdr_cmn::access (p);
  hdr_Esmcc * oh = hdr_Esmcc::access (p);
  hdr_Esmcc_ci * och;
  Packet * np;
  double rateSample;

  if (lastPktTime_ < 0) {  /* Initialization */
    lastCont_ = oh->sqn_ - 1;
    rateSample = ch->size_ / (lastPktTime_ = now);
    rate_ = rateSmoother_->GetSample (rateSample, now);
  }
  else {
    rateSample = ch->size_ / (now - lastPktTime_);
    rate_ = rateSmoother_->GetSample (rateSample, now);
    lastPktTime_ = now;
  }

  #ifdef ESMCC_DATA_PKT_DEBUG
  cout << now
    << " : at " << here_
    << " rcvd pkt sqn = " << oh->sqn_
    << " , rate = " << rate_ / 125000
    << " , rate sample = " << rateSample / 125000
    << '\n';
  #endif

  cr_ = oh->cr_;
  crTracAvg_ = oh->crTracAvg_;
  crTracDev_ = oh->crTracDev_;

  if (oh->sqn_ == lastCont_ + 1) {
    ++ lastCont_;
    Packet::free (p);
    return;
  }

  /* For simplicity, assume no out-of-order packets */
  if (SqnLE (oh->sqn_, lastCont_)) {
    Packet::free (p);
    return;
  }

  /* Now there are some losses */
  lastCont_ = oh->sqn_;

  if (tracAvg_ < 0) {
    /* Use crTracAvg_ to avoid unnecessary CR switch */
    tracAvg_ = crTracAvg_ < 0 ? rate_ : crTracAvg_;
  }
  else {
    tracAvg_ = (1 - TRAC_EWMA_FACTOR) * tracAvg_ + TRAC_EWMA_FACTOR * rate_;
  }

  #ifdef ESMCC_TRAC_DEBUG
  cout << now
    << " : at " << here_
    << " TRAC avg = " << tracAvg_ / 125000
    << " , sample = " << rate_ / 125000;

  if (crTracAvg_ > 0) {
    cout << " , CR TRAC avg = " << crTracAvg_ / 125000
      << " , dev = " << crTracDev_ / 125000
      << " , avg - d / " << COMPARE_TRAC_FACTOR << " = "
      << (crTracAvg_ - crTracDev_ / COMPARE_TRAC_FACTOR) / 125000
      << '\n';
  }
  else {
    cout << " , invalid CR TRAC\n";
  }
  #endif

  if (cr_ == here_ || crTracAvg_ < 0
      || tracAvg_ + SMALL_FLOAT
         < crTracAvg_ - crTracDev_ / COMPARE_TRAC_FACTOR) {
    np = allocpkt ();
    och = hdr_Esmcc_ci::access (np);
    och->trac_ = rate_;
    och->ciSrc_ = here_;
    och->timestampEcho_ = oh->timestamp_;
    send (np, 0);
    ++ ciSent_;
  }
  else {
    ++ ciSupp_;
  }

  Packet::free (p);
}


/**********************************
 R A T E   S M O O T H E R
 **********************************/


RateSmoother::RateSmoother (double span)
 : span_ (span), totalData_ (0)
{
  tail_ = &head_;
}


RateSmoother::~RateSmoother ()
{
  struct RateSample * cur = head_.next_, * tmp;

  while (cur != NULL) {
    tmp = cur->next_;
    delete cur;
    cur = tmp;
  }
}


/**
 * Get a sample and calculate smoothened rate
 * Return smoothened rate.
 */
double RateSmoother::GetSample (double rate, double time)
{
  struct RateSample * sample = new struct RateSample, * secondLatest,
                   * earliest;
  sample->time_  = time;
  sample->rate_ = rate;
  sample->next_ = NULL;
  tail_->next_ = sample;
  secondLatest = tail_;
  tail_ = sample;

  if (sample == head_.next_) return rate;

  totalData_ += rate * (time - secondLatest->time_);

  earliest = head_.next_;
  if (time - earliest->time_ > span_ + SMALL_FLOAT) {
    totalData_ -= (earliest->next_->time_ - earliest->time_)
                 * earliest->next_->rate_;
    head_.next_ = earliest->next_;
    delete earliest;
    earliest = head_.next_;
  }

  return totalData_ / (time - earliest->time_);
}
```

// program generate the data sheet

TCL SCRIPT

This TCL script is used to provide the interfacing In this program we first of all set up the source and receiver

```
//Set up of the source

proc SetupEsmccSrc {ns startTime stopTime node} {
  set group [Node allocaddr]
  set src [new Agent/ESMCC]
  $src set dst_addr_ $group
  $src set dst_port_ 0
  $ns attach-agent $node $src
  $ns at $startTime "$src start"
  $ns at $stopTime "$src stop; $src print-statistics"

  return $src
}

//setup of the recevier
proc SetupEsmccRcv {ns src startTime stopTime node} {
  set group [$src set dst_addr_]
  set rcv [new Agent/ESMCCSink]
  $ns attach-agent $node $rcv
  $rcv set dst_addr_ [$src set agent_addr_]
  $rcv set dst_port_ [$src set agent_port_]

  $ns at $startTime "$rcv start; $node join-group $rcv $group"
  $ns at $stopTime \
    "$rcv stop; $rcv print-statistics; $node leave-group $rcv $group"

  return $rcv
}

//setup of the FTP agent
proc SetupFtp {ns type pktSize startTime stopTime srcNode rcvrNode} {
  switch -- $type \
  Sack {
    set tcp [new Agent/TCP/FullTcp/Sack]
    set sink [new Agent/TCP/FullTcp/Sack]
    $sink listen
  } Reno {
    set tcp [new Agent/TCP/Reno]
    $tcp set class_ 2
    set sink [new Agent/TCPSink]
  } Newreno {
    set tcp [new Agent/TCP/Newreno]
    set sink [new Agent/TCPSink]
  } Vegas {
    set tcp [new Agent/TCP/Vegas]
    $tcp set window_ 1000000
    set sink [new Agent/TCPSink]
  }  default {
    puts "Unknown TCP type"
    exit 0
  }

  $tcp set packetSize $pktSize

  $ns attach-agent $srcNode $tcp
  $ns attach-agent $rcvrNode $sink
  $ns connect $tcp $sink
  set ftp [new Application/FTP]
  $ftp attach-agent $tcp
  $ns at $startTime "$ftp start"
  $ns at $stopTime "$ftp stop"

  return $ftp
}

# NOTE: there should be "ms" or "s" in burstTime and idleTime
proc SetupParetoUDP {ns rng rate pktSize burstTime idleTime startTime stopTime srcNode rcvrNode} {
  set udp [new Agent/UDP]
  $ns attach-agent $srcNode $udp
  set sink [new Agent/Null]
  $ns attach-agent $rcvrNode $sink

  $ns connect $udp $sink

  set pareto [new Application/Traffic/Pareto]
  $pareto use-rng $rng
  $pareto set packetSize_ $pktSize
  $pareto set burst_time_ ${burstTime}
  $pareto set idle_time_  ${idleTime}
  $pareto set rate_ $rate
  $pareto attach-agent $udp

  $ns at $startTime "$pareto start"
  $ns at $stopTime "$pareto stop"

  return $pareto
}

//Agent TCP Pareto is setup
proc SetupParetoTCP {ns rng type pktSize burstTime idleTime startTime stopTime srcNode rcvrNode} {
  switch -- $type \
  Sack {
    set tcp [new Agent/TCP/FullTcp/Sack]
    set sink [new Agent/TCP/FullTcp/Sack]
    $sink listen
  } Reno {
    set tcp [new Agent/TCP/Reno]
    $tcp set class_ 2
```

```
   set sink [new Agent/TCPSink]
  } Newreno {
   set tcp [new Agent/TCP/Newreno]
   set sink [new Agent/TCPSink]
  } Vegas {
   set tcp [new Agent/TCP/Vegas]
   $tcp set window_ 1000000
   set sink [new Agent/TCPSink]
  }  default {
   puts "Unknown TCP type"
   exit 0
  }

  $tcp set packetSize $pktSize

  $ns attach-agent $srcNode $tcp
  $ns attach-agent $rcvrNode $sink
  $ns connect $tcp $sink
  set ftp [new Application/FTP]
  $ftp attach-agent $tcp

  set r [new RandomVariable/Pareto]
  $r use-rng $rng

  set t $startTime
  set run 1
  while {$t < $stopTime} {
   if {$run} {
     $ns at $t "$ftp start"
     $r set avg_ $burstTime
   } else {
     $ns at $t "$ftp stop"
     $r set avg_ $idleTime
   }
   set d [$r value]
   set t [expr $t + $d]
   set run [expr ($run + 1) % 2]
  }

  if {$run == 0} {
   $ns at $stopTime "$ftp stop"
  }

  return $ftp
}
```