TEAR: TCP emulation at receivers – flow control for multimedia streaming*

Injong Rhee, Volkan Ozdemir
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
{rhee,vozdemi}@csc.ncsu.edu

Yung Yi
Department of Computer Engineering
Seoul National University
Seoul, Korea
yiyung@mmlab.snu.ac.kr

April 28, 2000

Abstract

Congestion and flow control is an integral part of any Internet data transport protocol. It is widely accepted that the congestion avoidance mechanisms of TCP have been one of the key contributors to the success of the Internet. However, TCP is ill-suited to real-time multimedia streaming applications. Its bursty transmission, and abrupt and frequent wide rate fluctuations cause high delay jitters and sudden quality degradation of multimedia applications. For asymmetric networks such as wireless networks, cable modems, ADSL, and satellite networks, transmitting feedback for (almost) every packet received as it is done in TCP causes congestion in the reverse path. In this environment, TCP may severely under-utilize the forward path throughput. Use of multicast further complicates the problem; TCP-like frequent feedback from each receiver to the sender in a large scale multicast session cause well-known scalability limitations (e.g. acknowledgment implosion).

We have developed a new flow control approach for multimedia streaming, called TCP emulation at receivers (TEAR). TEAR shifts most of flow control mechanisms to receivers. In TEAR, a receiver does not send to the sender the congestion signals detected in its forward path but rather processes them immediately to determine its own appropriate receiving rate. TEAR can determine this rate using congestion signals observed at the receiver. These signals are used to emulate the TCP sender's flow control functions at receivers.The emulation allows receivers to estimate a TCP-friendly rate for the congestion conditions observed in their forward paths. TEAR also allows receivers to adjust their receiving rates to a TCP-friendly rate without actually modulating the rates to probe for spare bandwidth, or to react to packet losses directly. Thus, the perceived rate fluctuations at the application are much more smooth than in TCP.

A unicast version of TEAR is implemented. This report describes the implementation of TEAR, examine the performance of this TEAR implementation from the NS simulation and Internet experiments, and compare it with that of other TCP-friendly flow control techniques. Our preliminary tests indicate that TEAR shows superior fairness to TCP with significantly lower rate fluctuations than TCP. TEAR's sensitivity to feedback interval is very low, so that even under high feedback latency, TEAR flows exhibit acceptable performance in terms of fairness, TCP-friendliness, and rate fluctuations. Finally, I will discuss the future extension of TEAR for multicast environments.

*This work is supported in part by NSF CAREER ANI-9875651.

1 Introduction

As the Internet becomes more diversified in its capabilities, it becomes feasible to offer services that were not possible under earlier generations of Internet technologies. Real-time multimedia streaming and IP multicast are two such emerging technologies. The development and use of commercial applications based on these technologies, such as Internet telephony, will become increasingly prevalent, and their traffic will constitute a large portion of the Internet traffic in the future.

Congestion and flow control is an integral part of any Internet data transport protocol whose traffic travels a shared network path. It is widely accepted that the congestion avoidance mechanisms employed in TCP have been one of the key contributors to the success of the Internet. However, few commercial streaming applications today are equipped with end-to-end flow control. The traffic generated by these applications are unresponsive to congestion and can completely lock out other competing flows, monopolizing the available bandwidth. The destructive effect of such traffic on the Internet commonwealth, namely *congestion collapse*, has been well studied [1].

TCP is ill-suited for real-time multimedia streaming applications because of their real-time and loss-tolerant natures. Its bursty transmission, and abrupt and frequent deep fluctuations in the transmission rate cause delay jitters and sudden quality degradation of multimedia applications. For asymmetric networks such as wireless networks, cable modems, ADSL, and satellite networks, transmitting feedback for (almost) every packet received as it is done in TCP is not very attractive because of lack of bandwidth on the reverse links. In asymmetric networks, packet losses and delays occurring in reverse paths severely degrade the performance of existing round trip based protocols such as TCP, resulting in reduced bandwidth utilization, fairness, and scalability [2, 3, 4]. Use of multicast further complicates the problem; in large-scale multicast involving many receivers (10,000 to 1M receivers), frequent feedback sent directly to the sender causes implosion.

The objective of our work is to develop, verify analytically and experimentally, and implement a suite of end-to-end flow control protocols for unicast and multicast real-time streaming applications. The developed protocols are evaluated based on TCP-friendliness, stability, and scalability. These properties must hold regardless of the types of networks, or more specifically whether networks are symmetric or asymmetric in bandwidth and delays.

We informally define these properties as follows:

- Fairness and TCP-friendliness: let B be the total bandwidth used by n TCP flows when they are only flows running on an end-to-end path. Suppose that there are m flows (of any protocol) running on that same path, then each flow must use B/m bandwidth share.
- Stability: after a network undergoes some perturbation because of flows joining and leaving, if the network reaches steady state, no matter what the state of the protocol at the end of the perturbation is, the protocol eventually reaches the fair and TCP-friendly rate.
- Scalability: the performance of a protocol instance does not depend on the number of its receivers.

We plan to achieve our goal by implementing a technique, called *TCP emulation at receivers (TEAR)*, that shifts most of flow control mechanisms to receivers. In TEAR, a receiver does not send to the sender the congestion signals detected in its forward path but rather processes them immediately to determine its own appropriate receiving rate. In receiver-driven flow control such as ones in [5, 6], this rate can be used by

that receiver to control its receiving rate independently without any feedback. In sender-driven flow control such as the one in [7], the rate can be sent to the sender for controlling the transmission rate. Note that this feedback happens much less frequently than that used for acknowledgment in TCP. Thus, it is more scalable and suitable for multicast and asymmetric networks.

TEAR can determine the "appropriate" receiving rates of receivers based on congestion signals observed at the receiver, such as packet arrivals, packet losses, and timeouts. Using these signals, TEAR emulates the TCP sender's flow control functions at receivers including slow start, fast recovery, and congestion avoidance. This emulation allows receivers to estimate a TCP-friendly rate for the congestion conditions observed in their forward paths. Further, TEAR smoothes estimated values of steady-state TCP throughput by filtering noise. This smoothed rate estimate will be reflected into the rate adjustment of receiving rates (e.g., by asking the sender to set the transmission rate to the smoothed rate estimate). Therefore, TEAR-based flow control can adjust receiving rates to a TCP-friendly rate without actually modulating the rates to probe for spare bandwidth, or to react to packet losses directly. Thus, the perceived rate fluctuations at the application are much more smooth than in TCP.

In this report, we present a unicast TEAR implementation. The detailed description on unicast TEAR can be found in Section 3. We conducted some preliminary simulation experiments using ns. Our ns source code is available on the public (we will also incorporate it into the daily snapshot of the ns distribution soon). Experiments are focused on studying fairness, TCP-friendliness, rate fluctuations, and its suitability under asymmetric networks. We compare the performance of TEAR with that of TFRC [8] (the daily snapshot version of Mar. 2, 2000) under the same simulation environments. The followings summarize our preliminary findings.

- Fairness and TCP-friendliness: TEAR and TFRC show excellent fairness and TCP-friendliness under high bottlneck bandwidth. However, as the number of competing TCP (SACK) flows increases over a connection with a drop-tail router, both TEAR and TFRC use less than the fair bandwidth share. Nontheless, the ratio of the fair share over the TEAR's bandwidth share can be bounded within a factor of 2 or 3 in the worst case. However, TFRC's share drops to zero under this environment (we reported this problem to the TFRC folks).

  TEAR and TFRC perform much better under a RED router. TEAR shows almost perfect fairness in all cases (low to high bandwidth) regardless of the number of TCP flows. However, TFRC's bandwidth share still drops to zero when the number of competing TCP flows is high and the available bandwidth is low.

- Rate fluctuations: both TEAR and TFRC flows show a much fewer and lower amount of rate fluctuations than TCP flows. However, TFRC flows exhibit much more fluctuations than TEAR when competing with TCP. This is the case even with the "CA" option of TFRC which the TFRC folks claim to have an effect of reducing rate fluctuations. We noticed that under our testing environments the option does not seem to have much effect.

- Sensitivity to feedback frequency (or time-scale of feedback): under asymmetric environments or multicast environments with many receivers, sending frequent feedback from receivers could degrade bandwidth utilization and scalability. TEAR seems to show good stability even with high feedback latency; when feedback latency is increased up to 10 RTTs, TEAR flows exhibit acceptable performance in terms of fairness, TCP-friendliness, and rate fluctuations. However, TFRC does not work well when feedback latency increases (its flows tend to monopolize the bandwidth, completing locking out TCP flows). TFRC folks told us that the current implementation of TFRC is not designed to

handle feedback latency larger than 1.5 RTTs (thus implying it may not be an inherent problem of TFRC).

We are also currently performing Internet experiments over various Internet connections. As their data become available, we will report results on this page.

Disclaimer: both TEAR and TFRC implementations are evolving even as we write this document. Thus, the performance result we report here may not be of inherent characteristics of the protocols (some bugs might still be present). In addition, more engineering and optimization are required to the current version of TEAR. For instance, TEAR's slow start behavior or response time to changes in available bandwidth is not studied. As we apply more optimization, we will report that on this page.

2 Related work

Existing flow control protocols for multimedia streaming applications can be roughly categorized into two approaches: *sender-based AIMD (SAD)* [9, 10, 11, 12, 13] and *model-based flow control (MFC)* [14, 15, 16, 17, 6, 7, 18].

SAD performs additive increase and multiplicative decrease (AIMD) rate control at a sender as in TCP. Typically, the protocols adopting SAD require a receiver to send *one acknowledgment for every received packet* to detect congestion indications such as packet loss and timeouts. Because of their AIMD property, they are provably stable and fair under steady state. These protocols also depend on RTT. Because of their frequent flow control feedback packets, reverse path conditions can severely degrade their performance.

MFC uses a stochastic TCP model [19, 20, 21] which represents the throughput of a TCP sender as a function (or formula) of packet loss rates and round trip time (RTT). Since MFC protocols can run at receivers, the problems associated with congestion in reverse paths may not occur. However, there are several issues that need further study. First, its stability property is not certain. Our preliminary work [22, 23] suggests that under certain circumstances, the protocols do not converge to the fair bandwidth share of network paths (resulting in either over-allocation or under-allocation of bandwidth). This happens because of inaccuracy in estimating loss rates and in the formula itself. It is not clear whether their deviation from fairness can be bounded. Second, modeling assumptions made by these protocols may not universally hold in all network environments. For instance, the TCP formula is not reliable when packet losses or RTT is correlated to the transmission rate of the flow being controlled [17]. This situation is commonly observed in asymmetric networks or under low statistical multiplexing environments.

Tuan and Park [24, 25] propose a novel congestion control scheme for real-time traffic which takes advantage of self-similarity of Internet traffic [26, 27, 28, 29]. The scheme extracts the correlation structure present over multiple time scale of Internet traffic and uses that information to predict the congestion level of future. Predicted congestion level is used to determine the degree at which a flow rate increases; when the predicted congestion level is low, a flow can afford to increase its transmission rate faster than the normal linear increase and when it is high, it can increase more slowly than the normal increase. We believe that use of the self-similarity property of Internet traffic in predicting future traffic characteristics can be applicable to many different parts of traffic control.

Rhee et al. [30] and Golestani and Sabnani [31] propose window-based hierarchical flow control for reliable multicast. The protocols roughly run in a similar fashion where receivers compute TCP-window and

report their window sizes to the sender through a logical tree structure imposed on the receiver set. Since these protocols require each receiver to send feedback (or a summary of feedback) per window update, these protocols do not require explicit RTT estimation. However, to consolidate frequent feedback from receivers, a hierarchical logical relation among receivers and the sender is necessary. Another problem with the window-based approach is its susceptibility to the drop-to-zero problem [32, 33]. Transient network congestion conditions can cause receiver windows to be reduced. Since the sender takes the minimum window size reported from receivers to determine the number of packets to send, following instantaneous minimum window sizes can cause severe under-utilization of the available bandwidth on the bottleneck path.

One of the protocols being considered by the IRTF-RM group is called *TCP-friendly model-based flow control (TFMC)* [7] where using a TCP formula [21], the TCP throughput on the bottleneck path is estimated, and the sender's rate is set to that throughput. TFMC is considered to be mature enough to be submitted for standardization to IETF [34]. However, we believe that the number of issues we mentioned above for MFC must be resolved before the standardization.

Receiver-driven layered multicast (RLM) [5] is the first scalable flow control protocol that combines layered encoding with receiver-driven flow control. In RLM, receivers join a new layer if they do not observe packet losses for some period of time, and leave a layer if they experience packet losses. Joining a new layer during no loss periods, which is referred to as *join experiments*, allows receivers to probe for spare bandwidth. Join experiment fails if packet losses occur after joining a new layer, in which case, receivers immediately leave that new layer. The stability property of RLM, however, is uncertain. The main problem lies in that when packet losses occur, only one layer is dropped at a time which results in linear rate reduction during packet loss. Since the rate increases by adding one layer at a time, the rate also increases only linearly. This symmetry in rate adjustment causes instability and unfairness. It is possible that the bandwidth shares of TCP and RLM diverge and RLM can monopolize the bandwidth by completely locking out TCP.

Vicisano, Rizzo, and Crowcroft [35] proposes a technique that improves on RLM by adding a TCP-like rate adjustment scheme to layering-based flow control. We call this technique *VRC*. VRC mimics TCP's AIMD control by setting the data rate of each layer exponentially. So when a receiver subscribing up to the i-th layer does not see any packet loss for a given time t_i, it can add a new layer. The data rate of this new layer and t_i are exponential functions of i. VRC also adds an innovative sender-oriented coordination mechanism for join experiments. Some preliminary study [36] shows that this coordination can substantially improve the max-min fairness of the protocol. One potential drawback in VRC is that it lacks a mechanism to determine TCP-friendly shares. Although its AIMD control can improve fairness and stability among the flows of its protocol, the data rate of receivers may not be fair with TCP flows competing on the same end-to-end paths (i.e., there is no bound on fairness to TCP flows). This is because the rates determined by VRC are completely independent of network delays. It is uncertain whether a flow rate can be made TCP-friendly while being completely unresponsive to network delays. Another potential problem with VRC is its multiplicative rate increase when adding a new layer. Although the waiting time before joining the new layer is spaced multiplicatively to mimic additive increase, this sudden increase might also cause big oscillation in receiving rates. VRC attempts to avoid this problem by injecting artificial burst traffic into existing layers to test whether the new layer's rate can be accommodated by receivers. However, packet losses induced by these bursts may incur sudden playout quality degradation. Although the "damage" due to the bursts can be restored later on, the sudden variations in playout quality can be very disconcerting for users.

Wu et al.[37] propose using small equal bandwidth layers called "ThinStreams" to achieve smooth fluctuation in rate. This allows flow control based on layering to perform fine-grained rate adjustment.

Li et al.[6], Turletti et al.[16] and Wang and Zhakhor [18] propose to use a TCP formula [19, 14] where receivers determine whether to drop a layer based on a TCP formula. Li et al.[6] also provides

an innovative scheme that ensures inter-session fairness among competing flows by altering the sensitivity of each layer to respond to loss rates. However, as we discussed in [22], a model based approach can potentially cause instability; under certain initial rates set by receivers, receiving rates can converge to a non-fair bandwidth share. It requires further study to verify whether the similar instability due to inaccuracy in the TCP throughput formula does not occur in their techniques.

There are some newly emerging class of flow control for multimedia stream and reliable multicast [38, 39, 40, 41] using router support. Especially, the approach [39, 42] based on packet filtering in PGM [43] is gaining some interest from the Internet community. The protocol requires a down link router to monitor the aggregate of all flows across the link, and to report the congestion measurement up toward the source of the IP multicast tree. At a regulation point in the tree, a router uses packet filtering to regulate the rate of the IP multicast flow to the link. The rate is determined using a TCP-formula [21] which takes the mean loss rate and RTT on the link. In some sense, we can view this problem as a unicast problem between the up-stream regulation router and the down-stream router. The only difference is that the rate control is performed at routers. Thus, the result of our proposed work can be applicable to this type of flow control.

3 TCP emulation at receiver (TEAR)

3.1 Basic assumption

We assume that the probability of having a packet loss within a window of x consecutively transmitted packets does not depend on their transmission rate. That is, no matter how large or small intervals they are transmitted in, the probability that at least one of them in that window is lost is the same given the network conditions do not change during the transmission period. We call this assumption *rate independence.*

In today's Internet, packets are dropped from routers indiscriminately of the transmission rates of flows when routers lack buffer. Even in future Internet where more fair queuing and QoS mechanisms are provided, it will be still the case at least for the flows within the same class (because QoS provisioning is likely applied to aggregated flows). Rate independence holds if packet losses happen independently because packets are dropped indiscriminately at routers. Unfortunately, in today's Internet where drop-tail routers prevail, packet losses are highly correlated.

However, there have a number of studies that loss bursts in the Internet are short and the loss correlation does not span long, typically less than one RTT. Further, TCP can be typically modeled using a "loss event" which is, informally, defined to be a single loss burst (or the losses within the same TCP congestion window). This is because TCP reacts only once per loss event. In fact, many TCP literatures (e.g., [14, 20, 15, 16, 17, 18]) assume that loss events are not correlated and happen independently. Therefore, if we treat losses within the same loss burst as a single loss event, we can describe the behavior of loss events by a Bernoulli model. When emulating TCP, TEAR ignores losses that are likely correlated, and treat them as a single loss event. We believe that under such operating conditions, rate independence can be generally assumed. However, in this report, we provide no evidence for rate independence in the current Internet. Further study is required.

Rate independence plays an essential role in establishing the theoretical foundation of our approach. The problem we face is to estimate the throughout of a TCP connection over the same end-to-end path only by observing packet arrival process of a TEAR connection at the receiver. Note that packets in TEAR are possibly transmitted at a different rate than those in the TCP connection. This assumption implies that a

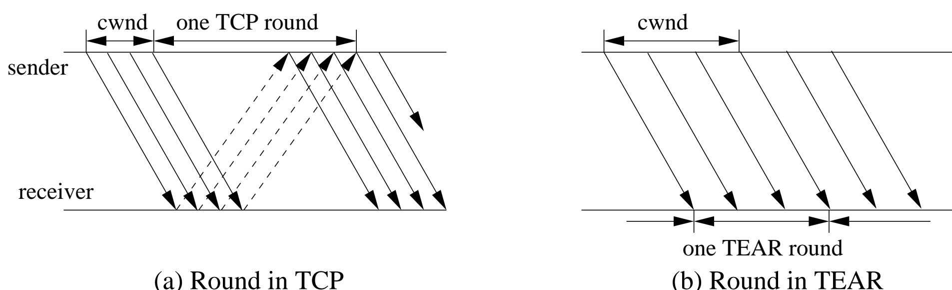
window of x packets in the TCP connection has the same loss probability as that in the TEAR connection regardless of their transmission rates. Thus, TEAR can ignore real time over which a window of packets is transmitted or the transmission rate of the connection where the estimation takes place.

3.2 Round

TCP maintains a variable called *cwnd* that indicates the number of packets in transit from the sender to the receiver. *cwnd* is updated when the sender learns via an acknowledgment that a packet is received by the receiver. TEAR also maintains the same variable at the receiver (instead of at the sender) and updates it according to the same algorithm based on the arrival of packets. However, since TEAR and TCP might be sending at different rates, the window update function cannot be described in terms of real-time (e.g., round trip time). We model the TCP window adjustment protocol in terms of *round* instead of round-trip times (RTT).

A transmission session is partitioned into non-overlapping time periods, rounds. A new round begins when the current round ends. A round contains roughly an arrival of the *cwnd* number of packets. In TCP, a round is recognized at the sender when an acknowledgment packet is received for the reception of packets in the current congestion window (*cwnd*) whereas in TEAR, the receiver can recognize a round when receiving packets.

This difference may cause *cwnd* to be updated at a different rate in TEAR than in TCP since *cwnd* is updated at each round instead of each RTT. In TEAR, the duration of a round depends on the inter-arrival times of *cwnd* packets which depend on the transmission rate of TEAR. However, in TCP, a round implies one RTT since TCP updates its window at the sender at the reception of acknowledgment. Figure 1 illustrates this difference. To account for this discrepancy, TEAR estimates TCP throughput by assigning a fictitious RTT time to each round. When estimating the transmission rate during one round, TEAR divides the current value of *cwnd* by the current estimate of RTT instead of the real-time duration of the round. The TEAR receiver estimates the TCP throughput by taking a long-term weighted average of these per-round rates, and reports it to the sender. The sender sets its rate to that reported rate. Below, we provide more details on the TEAR protocol.



(a) Round in TCP   (b) Round in TEAR

Figure 1: Rounds

3.3 State transition

The states of the TEAR protocol consist of seven states: SLOW-START-READY, SLOW-START, CONGESTION-AVOIDANCE-READY, CONGESTION-AVOIDANCE, FAST-RECOVERY, TIMEOUT, and GAP. Figure 2 shows the state transition diagram of the protocol. SLOW-START, CONGESTION-AVAOIDANCE, FAST-RECOVERY, and TIMEOUT corresponds to the states of TCP during slow-start, congestion avoidance, fast recovery, and timeout respectively. SLOW-START-READY, CONGESTION-AVOIDANCE-READY, and GAP are intermediary states required to run the window adjustment protocol at the receiver.

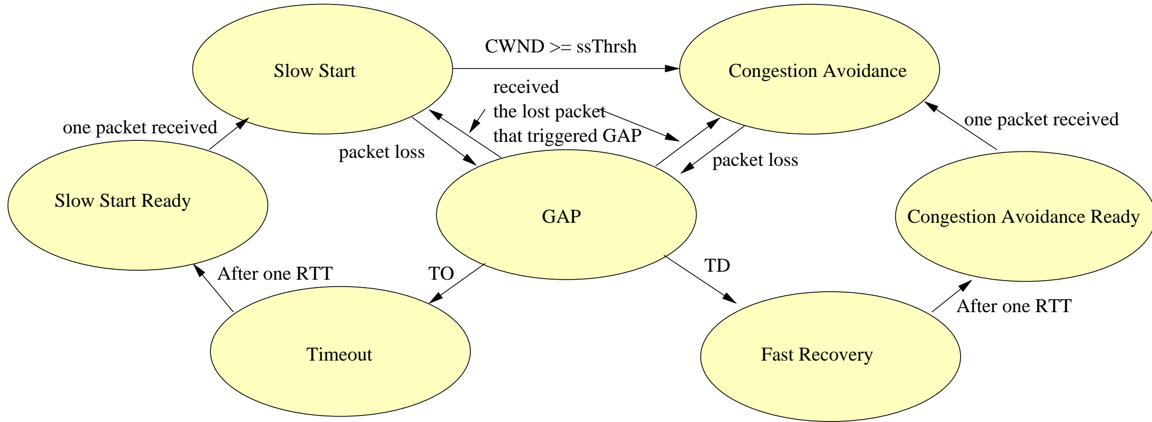


Figure 2: State transition of TEAR

The first round begins at the beginning of a transmission session, and the state is initialized to SLOW-START-READY. Initially *cwnd* is set to 1, and a variable *ssThrsh* is set to a default value larger than 2. It is used to transit the protocol state from SLOW-START to CONGESTION-AVOIDANCE. When receiving the first data packet, the second round begins and the state is changed to SLOW-START. During the CONGESTION-AVOIDANCE or SLOW-START state, a round ends only when the $\lfloor lastCwnd \rfloor$ number of packets are received from the beginning of that round. *lastCwnd* is the value of *cwnd* at the end of the previous round. A new round also begins when the state is changed to the FAST-RECOVERY-READY or SLOW-START-READY state. No new round starts in the GAP, FAST-RECOVERY, or TIMEOUT state.

3.4 Increase window algorithm

We say that a packet is received *in sequence* if the difference between the sequence number of that packet and that of its last received packet is exactly one. *cwnd* is incremented when a new packet is received in sequence at the CONGESTION-AVOIDANCE, or SLOW-START state. *cwnd* is also incremented when the receivers enters the CONGESTION-AVOIDANCE, or SLOW-START state. When a packet is received at the CONGESTION-AVOIDANCE state or when the state is changed to the CONGESTION-AVOIDANCE state, *cwnd* is incremented by $1/lastCwnd$. This emulates TCP window increase during congestion avoidance. When a packet is received in sequence at the SLOW-START state or when the state is changed to SLOW-START, *cwnd* is incremented by one. This emulates TCP window increase during slow start. At the beginning of each round, *lastCwnd* is updated to the value of *cwnd* to be used in computing the next round's

increment. When an updated *cwnd* is larger than *ssThrsh* at the SLOW-START state, the state is changed to CONGESTION-AVOIDANCE.

3.5 Decrease window algorithm

Suppose that the last packet received has a sequence number l. When a new packet received has a sequence number larger than $l + 1$ (i.e., it is not in sequence), and the state is SLOW-START or CONGESTION-AVOIDANCE, we change the state to the GAP state (i.e., a packet loss is detected). During the GAP state, *cwnd* is not modified. The GAP state is an intermediary state where the receiver determines whether the losses are for timeout or triple duplicate acknowledgment events. In TCP, when a packet loss occurs, the sender either does not receive any acknowledgment or receives only duplicate acknowledgments and during this time, *cwnd* is unchanged. GAP mimics the state of TCP during this time.

Fast recovery In TCP, packets received after a packet loss triggers a duplicate acknowledgment. Thus, the reception of three packets after the losses will trigger three duplicate acknowledgments in TCP (assuming no delayed acknowledgment). If these acknowledgments are received before the timeout, the TCP sender enters the fast recovery phase. Note that in TCP (SACK) at most $lastCwnd - 1$ packets are transmitted after the transmission of the packet that is lost. Emulating this behavior, the TEAR receiver enters FAST-RECOVERY from GAP when at least two packets are received before receiving any packet with sequence number larger than $l + lastCwnd$. In addition, these packets must be received within a $T_{timeout}$ period after the reception of packet l (the last packet received in sequence before the GAP state). $T_{timeout}$ is an estimated time for *lastCwnd* packets to arrive, and is defined below. Figure 3 illustrates the fast recovery detection of TCP and TEAR.

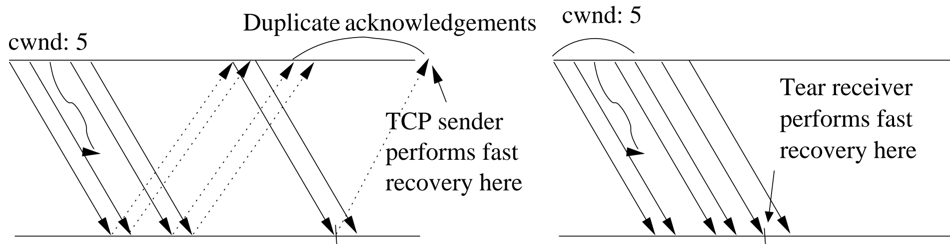


Figure 3: Fast recovery

In the GAP state, if packet $l + 1$ is received, it returns to the last state before GAP, and *cwnd* is updated according to the increase algorithm. This happens when packet $l + 1$ is reordered in the network. All those packets received before the reception of this reordered packet $l + 1$, but have a higher sequence number of $l + 1$ are considered to be received at once when the state is resumed from GAP. Thus, *cwnd* is incremented for each of those packets if there is no missing packets. If there is any packet i whose next packet in sequence is not received, but some subsequent packets to i (i.e., some packet is missing) are, then the state is changed to a new GAP state. At this moment, the last packet received before entering this GAP state is considered to be packet i (i.e., $l = i$). $T_{timeout}$ is also counted from the reception time of packet i.

At the FAST-RECOVERY state, the receiver waits for an RTT period. All the packets received during this RTT period are ignored. This mimics the TCP behavior during packet losses; it reduces its window only once for all the losses of packets transmitted within the same congestion window. This waiting can be achieved by setting a timer for the current estimate of RTT. To be more accurate, the receiver can send a feedback packet when a loss occurs, and the receiver can wait until the sender acknowledges the reception of the feedback. In the current unicast implementation, we adopt the latter approach.

At the end of that RTT period, the state is changed to CONGESTION-AVOIDANCE-READY state and a new round begins. Thus, the round before this new round spans from the beginning of the last round and to the end of the RTT period. During the last round, *cwnd* is not changed. As the new round begins in the CONGESTION-AVOIDANCE-READY state, the receiver reduces *cwnd* and *lastCwnd* to the half of the value of *cwnd* at that time. When a new packet is received after this state, the state is changed to CONGESTION-AVOIDANCE and *cwnd* is incremented according to the increase algorithm. At least one packet must be received after the losses triggered FAST-RECOVERY. This ensures that before *cwnd* is increased again, the network state has recovered from the losses.

Timeout If the receiver does not enter FAST-RECOVERY from GAP until $T_{timeout}$ time has past since the reception of packet l, it enters TIMEOUT. In addition, if no packet is not received before $T_{timeout}$ after the transition to CONGESTION-AVOIDANCE-READY, then it enters TIMEOUT. $T_{timeout}$ is computed as follows:

$$T_{timeout} = T_{interarrival} * lastCwnd * 2DEV \quad (1)$$

$T_{interarrival}$ is the inter-packet transmission time and can be computed by taking the inverse of the current transmission rate. This information is embedded in the packet header by the sender. *DEV* is the deviation in RTT estimates which is computed in the same way as in TCP by the sender from feedback. This deviation can be also be computed by taking deviation in the time difference from the sending time stamp of a packet and its receiving time stamp, and multiplying the deviation by $\sqrt{2}$. This technique is useful when direct feedback from the receiver to the sender is not allowed for scalability such as in multicast.

This timeout period is different from TCP's. TCP enters timeout when a packet is not acknowledged until its retransmission timer expires. If fast retransmit and recovery are triggered and recover the packet before that event, the timeout is avoided. Typically, retransmission timers are set to a value large enough so that triple duplicate acknowledgments can be received before the timeout (if they are indeed sent). Thus, when fast retransmit and recovery are possible, the timer value are large enough to allow it.

In TEAR, since no acknowledgment is sent, timeout must be detected at the receiver. This makes it difficult to detect timeout. However, since the receiver can detect packet losses, it can obtain some hints for timeout from packet arrivals. For instance, we know that in TCP, prior to the detection of fast recovery, the sender transmits exactly $lastCwnd - 1$ packets after the first packet that is lost to cause GAP. Therefore, if the receiver gets less than three packets after a packet loss until it learns that $lastCwnd - 1$ packets are sent by the sender after the lost packet was sent, it knows that fast recovery will not be triggered if such situation occurs in TCP. $T_{timeout}$ is the time to allow at least $lastCwnd - 1$ packets to arrive at the receiver. We allow additional $2 \times DEV$ per packet interval to account for delay jitters in the forward path. Note that TCP uses $4 \times DEV$ for jitters in round trip times.

After entering TIMEOUT, the receiver again waits for an RTT period to ignore packets lost during the

same loss burst that caused the timeout. At the end of the RTT period, the state is changed to the SLOW-START-READY state, *ssThrsh* is set to the half of $\min\{cwnd, 2\}$, *cwnd* and *lastCwnd* are set to 1, and $T_{timeout}$ is doubled. A new round begins at this time. Note that its last round spans from the beginning of the last round and to the end of this RTT period. *cwnd* is not changed during this last round. TEAR waits to receive a new packet before entering SLOW-START from SLOW-START-READY. SLOW-START-READY is required for the receiver to know the sequence number of the next packet to be received. If no packet arrives before $T_{timeout}$ after the transition to the SLOW-START-READY, it enters TIMEOUT again. When entering SLOW-START, the receiver resets $T_{timeout}$ to the value in Eqn 1.

3.6 Rate calculation

At the end of each round, the receiver records the current values of *cwnd* and *RTT* to a *history* array if that round does not involve the TIMEOUT state; otherwise, it records the current values of *cwnd* and *RTO*. RTO is defined to be $SRTT + 4DEV$ where $SRTT$ is an exponentially weighted moving average of *RTT*. These values are used to estimate TCP-friendly rates.

TCP's transmission rate can be computed by dividing *cwnd* by *RTT*. However, TEAR cannot set its transmission rate to this value (computed for each round) because it will cause the level of rate fluctuations as TCP which we want to avoid. Figure 4 plots the values of *cwnd* over rounds for a typical run of TEAR. The saw-teeth-like pattern indicates the additive increase and multiplicative decrease (AIMD) behavior of TCP window management. From the figure, we observe that although instantaneous rates would be highly oscillating, long-term throughput would be fairly stable. The idea is to set the TEAR transmission rate to an averaged rate over some long-term period T.

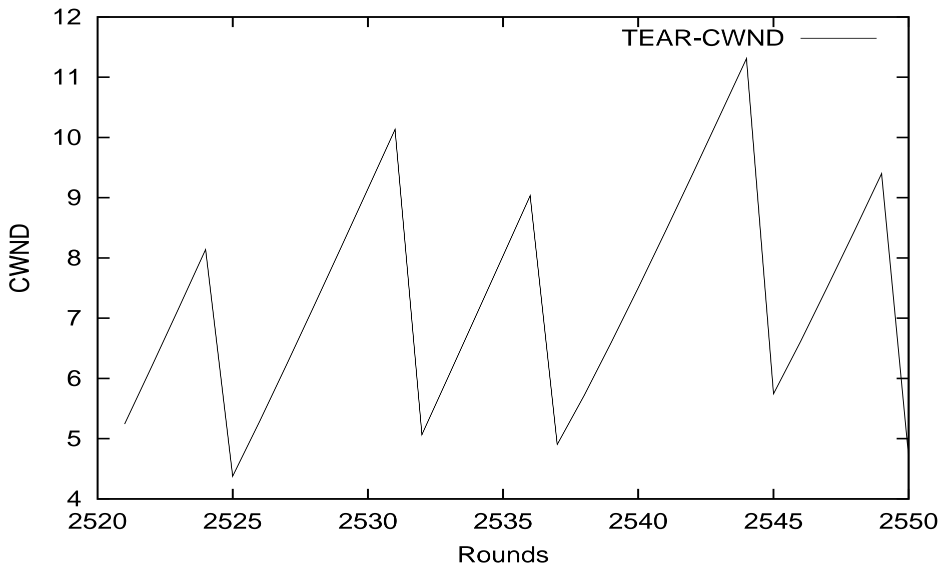


Figure 4: The values of congestion window over rounds in a typical run of TEAR

Epoch The big question is how large T should be. Note that if T is set too small, the rate would fluctuate too much. If it is too large, then rate adjustment would be too insensitive to network congestion state. Certainly, T must be larger than the length of one "saw tooth". If it is less than that, it will show the same fluctuation pattern as TCP. An *epoch* defines one "saw tooth". An epoch is a period that begins either when the receiver enters the SLOW-START or CONGESTION-AVOIDANCE state or at the beginning of the transmission session. When a new epoch starts, the current epoch ends which happens when the receiver enters the SLOW-START or CONGESTION-AVOIDANCE state (i.e., after a packet loss).

Suppose that the current epoch is the kth epoch. At the end of each round, the receiver divides the sum of all the *cwnd* samples recorded in the kth epoch by the sum of the RTTs or RTO recorded in that epoch (note that there can be only one RTO in an epoch). We call the result the *rate sample* of epoch k. Setting the rate to a rate sample at the end of each epoch would result in much more smoothed rate adjustment. However, some unnecessary rate fluctuations might still be present because some rate samples may not be representative of the actual fair share rate due to noise in loss patterns. In the current environments, loss patterns are highly noisy. Since the end of an epoch is determined by packet losses, if we set T to be the size of one epoch, the estimated rate would also be subject to the noise. We need to look further back than one epoch.

To filter out the noise, we need to apply some form of weighted averaging over rate samples taken over several W epochs in the past. At the end of each round, the TEAR receiver computes a weighted average of the last W rate samples taken at the end of each of the last W epochs. If the current epoch k is in process, then that sample is used only if adding the current sample in the averaging increases the current rate. This is because while the current epoch is in progress, its rate sample can be too small. Until the epoch becomes sufficiently large or it ends (with packet losses), that sample is not reliable, so ignored. This is done as follows. If the kth epoch is in progress, then we take a weighted average of the samples from the $k-1$th to $k-W-1$th epochs and compare the result with the weighted average from kth to $k-W-1$th epochs. The larger of the two averages multiplied by the packet size P is taken as a candidate for a *feedback rate* to the sender. We call this candidate f_{cand}. If there has been less than W epochs (i.e., $k < W$), then the missing samples are set to 0.

For the current implementation, we choose W to be 8, and we apply the following weights.

epoch	k	k-1	k-2	k-3	k-4	k-5	k-6	k-7
weight	1/6	1/6	1/6	1/6	2/15	1/10	1/15	1/30

Other distribution functions are possible such as a Gaussian or exponential distribution, and they provide a similar performance. We need more experience with other distributions. Currently the choice of W and weights are still arbitrary.

Feedback The sender sets its current transmission rate to the most recently received rate estimate from the receiver. If f_{cand} is less than the previously reported rate, then the receiver reports f_{cand} immediately to the sender. Otherwise, the receiver sends its rate estimate at the end of a *feedback round*. The duration of a feedback round is a parameter to the system. The rate estimate reported at the end of a feedback round is equal to f_{cand} computed at that time.

4 Simulation Result

Our simulation experiments are designed to study the TCP-friendliness, fairness, and smoothness of TCP-based rate adjustment in a unicast environments. In all experiments, we run TCP-SACK flows and TEAR flows at the same time.

We conduct the same experiments for TFRC. The default values of TFRC parameters are used which are shown below:

```
Agent/TFRC set packetSize_ 1000
Agent/TFRC set df_ 0.95 ;       # decay factor for accurate RTT estimate
Agent/TFRC set tcp_tick_ 0.1 ;
Agent/TFRC set ndatapack_ 0 ;   # Number of packets sent
Agent/TFRC set srtt_init_ 0 ;   # Variables for tracking RTT
Agent/TFRC set rttvar_init_ 12
Agent/TFRC set rtxcur_init_ 6.0
Agent/TFRC set rttvar_exp_ 2
Agent/TFRC set T_SRTT_BITS 3
Agent/TFRC set T_RTTVAR_BITS 2
Agent/TFRC set InitRate_ 1000 ; # Initial send rate
Agent/TFRC set overhead_ 0 ;    # If > 0, dither outgoing packets
Agent/TFRC set ssmult_ 2 ;      # Rate of increase during slow-start:
Agent/TFRC set bval_ 1 ;        # Value of B for TCP formula
Agent/TFRC set ca_ 1 ;          # Enable Sqrt(RTT) congestion avoidance
Agent/TFRC set printStatus_ 0
Agent/TFRC set rate_ 0.0
Agent/TFRC set bval_ 1

Agent/TFRCSink set packetSize_ 40
Agent/TFRCSink set InitHistorySize_ 100000
Agent/TFRCSink set NumFeedback_ 1
Agent/TFRCSink set AdjustHistoryAfterSS_ 1
Agent/TFRCSink set NumSamples_ -1
Agent/TFRCSink set discount_ 1; # History Discounting
Agent/TFRCSink set printLoss_ 0
Agent/TFRCSink set smooth_ 1 ;  # smoother Average Loss Interval
```

Figure 5 shows the symmetric network topology used in our experiments. Each experiment is run with different values of following parameters: the bottleneck bandwidth, denoted as xx, (10Mbs, 5Mbs, 2.5Mbs, 128Kbs) the number of competing TCP flows (1, 2, 4, 8, 16) and the number of competing TEAR (or TFRC) flows (1, 2, 4, 6, 16), the router types (Drop Tail or RED) of the bottleneck link, the feedback latency (1 RTT,4 RTTs,10 RTTs)[1]. Link delays are fixed. The running time are set to 400 seconds, and each network flow is started with one second interval.

The complete results can be found following this link. The figures in the link plot the performance of TEAR and TFRC when competing with different numbers of TCP flows and their own flows (denoted $x : y$ where x is the number of TEAR (or TFRC) flows, and y is the number of TCP flows).

[1] When running TFRC, we vary the feedback latency from 1.5 RTTs, 4 RTTs, and 10 RTTs.

```
    20Mb/s,10ms     xx Mb/s,10ms     20Mb/s,10ms
n0 ------------ n1 ------------ n2 ------------ n3
```

Figure 5: Simulation topology

At each column, the top figure shows the aggregate throughput obtained by the number of bytes sent divided by the total elapsed time. TEAR flows and TFRC flows are blue lines and TCP flows are red lines. The green line indicates the fair share (i.e., the total bottleneck bandwidth divided by the total number of flows).

The bottom figure shows instantaneous rate samples. It shows only the values of one TEAR or TFRC flow, and the values of one TCP flow although actual runs are with x TEAR flows and y TCP flows. TCP rates are sampled at every 100ms interval by dividing the number of bytes sent over one interval by 100ms. The black color also indicates the rate samples of TEAR or TFRC taken at every 100ms interval. The green line shows the transmission rate taken whenever the rate is updated. The red line indicates the fair share.

Below, we highlight some subset of the results to illustrate the performance comparison between TEAR and TFRC.

4.1 Fairness and TCP-friendliness

10Mps, Droptail, 8:8 Figure 6 shows the instantaneous rate samples of TCP and TEAR, and TCP and TFRC respectively with the bottleneck bandwidth 10Mbps, the droptail router, and the number of flows 8:8. Both TEAR and TFRC rates follow the fair share very well.

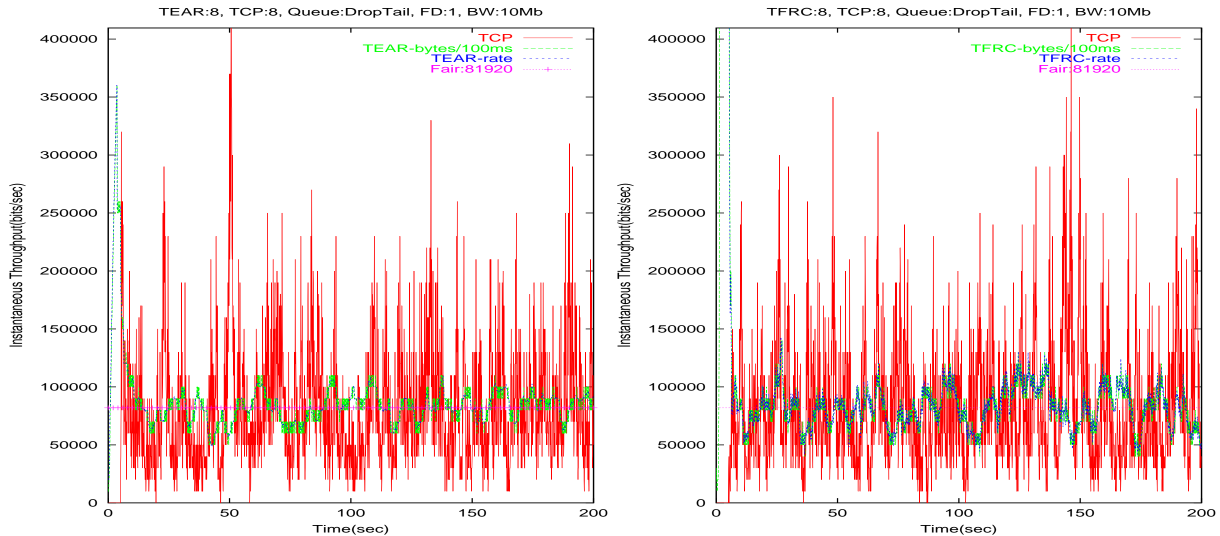


Figure 6: Bottleneck:10Mbs, Droptail router: Left (8 TCPs, 8 TEARs, Feedback 1 RTT), Right (8 TCPs, 8 TFRCs, Feedback 1.5 RTT)

2.5Mps, Droptail, 1:16 Figure 7 is from the run with the bottleneck bandwidth 2.5Mbps, the droptail router, and the number of flows 1:16. TEAR uses less than the fair share (about one half). TFRC's rate

drops to zero.

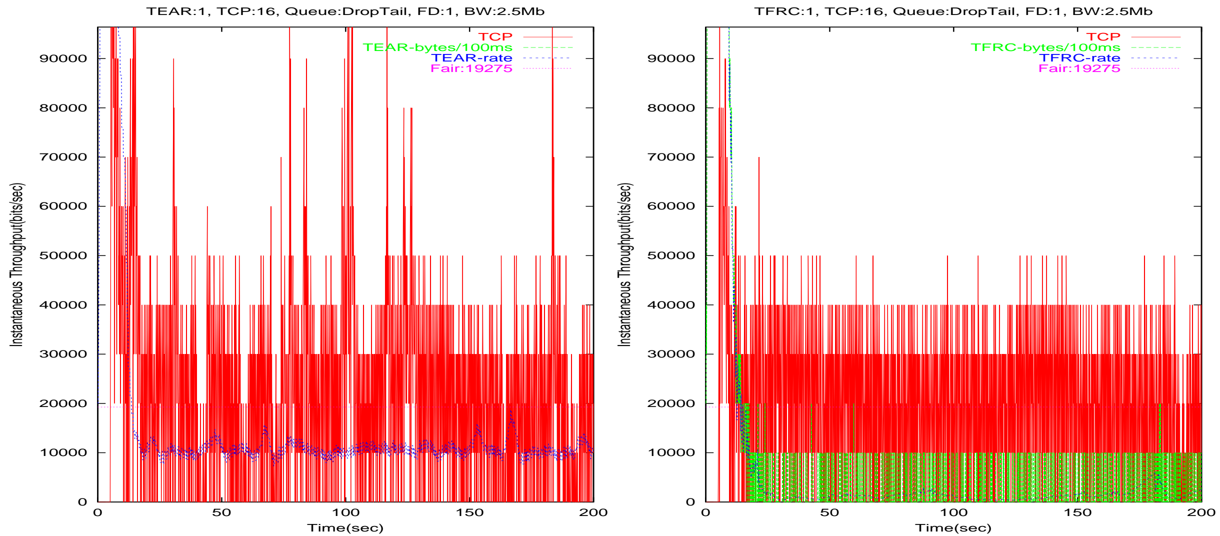


Figure 7: Bottleneck 2.5Mbs, Droptail router: Left(1 TEAR, 16 TCPs, Feedback 1 RTT), Right (1 TFRC flow, 16 TCP flows, Feedback 1.5 RTT)

2.5Mps, Droptail, 8:8 Figure 8 is from the run with the bottleneck bandwidth 2.5Mbps, the droptail router, and the number of flows 8:8. TEAR uses slightly less than the fair share. TFRC's rate drops to zero.

2.5Mps, RED, 1:16 Figure 9 is from the run with the bottleneck bandwidth 2.5Mbps, the RED router, and the number of flows 1:16. TEAR's rate follows the fair share pretty well. TFRC's rate is still very low, and sometimes drops to zero.

2.5Mps, RED, 8:8 Figure 10 is from the run with the bottleneck bandwidth 2.5Mbps, the RED router, and the number of flows 8:8. Both TEAR's rate and TFRC's rate oscillate around the fair share. (although TFRC's rate sometimes gets very low).

4.2 Rate fluctuations or smoothness

In all figures shown above, both TEAR and TFRC show much fewer and lower fluctuations than TCP (in the order of magnitude). However, we observe that TFRC tends to show a little more and larger fluctuations.

Figure 10 is from the run with the bottleneck bandwidth 2.5Mps, the RED router, and the number of flows 1:1. TEAR shows very stable rate transitions around the fair share. However, TFRC shows almost as many and as much fluctuations as TCP. When the droptail router is used, the phenomenon gets worse.

As the number of competing flows increases, the rate fluctutations of TFRC greatly subsides (especially, in terms of size). However, in terms of the number of rate fluctuations, we still see many fluctuations. Figure 12 is from the run with the bottleneck bandwidth 2.5Mps, the RED rounter, and the number of

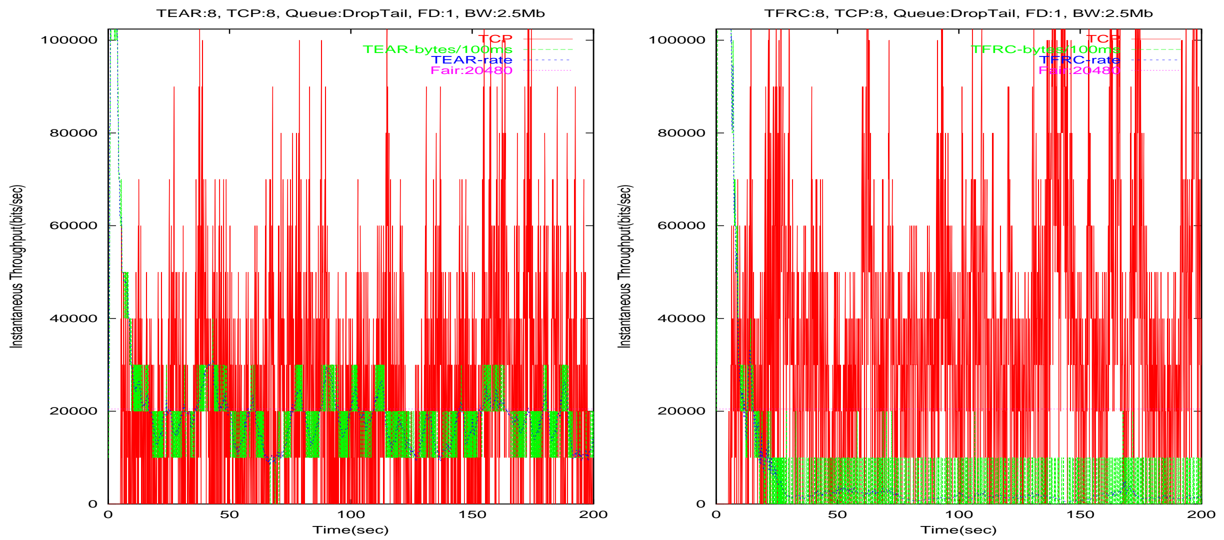


Figure 8: Bottleneck:.2.5Mbs, Droptail router: Left (8 TCPs, 8 TEARs, Feedback 1 RTT), Right (8 TCPs, 8 TFRCs, Feedback 1.5 RTT)

flows 4:4. While the TEAR flow shows very stable rate oscillations (fewer and lower), the TFRC flow still undergoes many fluctuations.

4.3 Sensitivity to the feedback latency

In this section, we examine the performance of TEAR and TFRC over various values of feedback latency. Note that the current implementation of TFRC is not designed to handle larger feedback latency. So the results with larger feedback delays may not be of the inherent characteristics of TFRC.

Figures 13 and 14 are from the runs with 8 TEAR flows and 8 TCP flows on a 2.5Mbs droptail bottleneck. We run four experiments, each with a different value of the feedback latency taken from 1 RTT, 2 RTTs, 4 RTTs, and 10 RTTs. In all runs, TEAR shows consistent fairness. The rate fluctuations are consistently low.

Figures 15 and 16 are from the runs with 8 TFRC flows and 8 TCP flows on a 2.5Mbs droptail bottleneck. We run three experiments, each with a different value of the feedback latency taken from 1.5 RTTs, 4 RTTs, and 10 RTTs. The performance of TFRC under these environments is a little unpredictable. When using 1.5 RTTs and 4 RTTs, the bandwidth shares of TFRC are very low. However, when using 10 RTTs, its bandwidth shares are very high.

5 Summary and future work

In this report, we describe a new approach to flow control called TCP emulation at receivers (TEAR) for unicast and multicast streaming. Our goal is to develop a flow control protocol that can be fair, TCP-friendly, stable and scalable. At the same time, the rate does not fluctuate much over the fair share. These properties must also hold under various network environments including traditional symmetric networks, and emerging asymmetric networks.

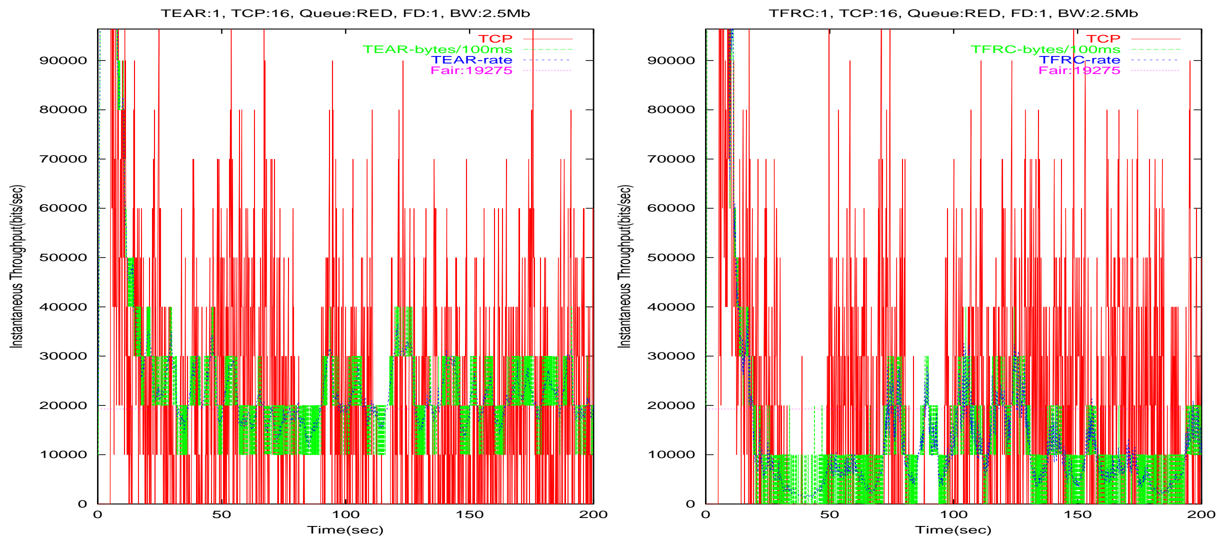


Figure 9: Bottleneck:2.5Mbs, RED router, Feedback 1 RTT: Left (1 TFRC flow, 16 TCP flows), Right (1 TEAR, 16 TCPs)

We presented a unicast TEAR protocol that can be suitable for long-running streaming applications. We reported a preliminary work on verifying the performance of the protocol, and compare it with that of TFRC, a competing TCP-friendly unicast protocol based on a TCP-friendly formula.

We found that both protocols possess many desirable properties for streaming applications when their flows compete with long-running TCP-SACK flows. They show fairness and TCP-friendliness, and excellent smoothness in rate fluctuations. When compared to TFRC, TEAR shows better fairness and smoothness. TFRC shows performance glitches when competing with many TCP flows for a small amount of bottleneck bandwidth. Their rates under this environment drop almost to zero. We don't know the cause of this, but suspect that this might be due to inaccuracy in estimating loss rates and in the TCP formula itself. We analytically showed in an earlier technical report [22] that this problem is inherent in the model-based (or equation-based) approach.

Our experiments are focused on studying the behavior of TEAR and TFRC under steady state where all the traffic is generated by long-running flows. Clearly this environment is not realistic because today's Internet traffic is made of many short-lived flows. We will perform more experiments involving more realistic background traffic. We plan to use the traffic model developed in our earlier work [44] that try to depict today's bursty Internet traffic. We also plan to run extensive Internet experiments.

TEAR can be used to enhance the scalability of multicast flow control. In TEAR, receivers estimate their own appropriate receiving rates. Thus, the work is naturally distributed. Because it can provide pretty accurate estimate of TCP-friendly rates even with a low frequency of feedback, it helps solve feedback implosion problem.

Two types of TEAR-based multicast flow control are possible. First, in receiver-driven layered multicast, receivers can use TEAR to determine their TCP-friendly receiving rates, and receivers can join enough multicast layers (assuming all layers are transmitted at an equal rate) to receiver at their estimated rates. In this case, little involvement from the sender is needed for flow control. Second, in sender-driven multicast, receivers can periodically feedback their rates estimated by TEAR to the sender. The sender selects the bottleneck receiver based on these rate reports, and sets its rate to the one reported by that receiver.

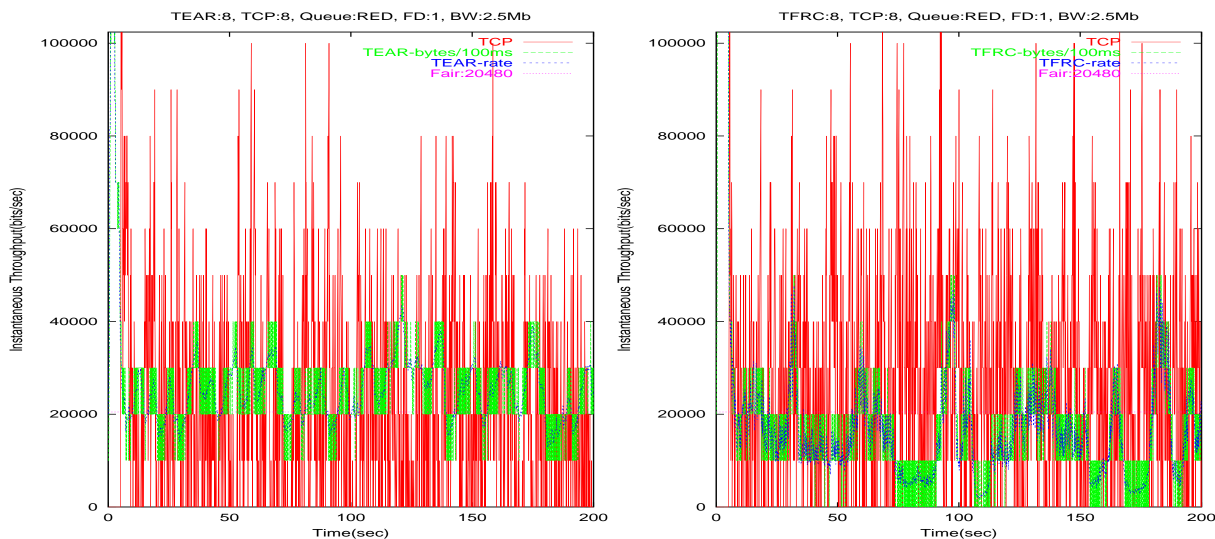


Figure 10: Bottleneck:2.5Mbs, RED router, Feedback 1.5 RTT: Left(8 TCPs, 8 TEARs, Feedback 1 RTT), Right (8 TCPs, 8 TFRCs, Feedback 1.5 RTT)

There are a number of issues that have to be resolved before these protocols can be realized.

References

[1] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, August 1999.

[2] H. Balakrishnan, V. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *ACM Mobile Networks and Applications (MONET)*, 1999.

[3] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, December 1997.

[4] T. V. Lakshman, U. Madhow, and B. Suter. Window-based error recovery and flow control with a slow acknowledgement channel: a study of TCP/IP performance. In *Proceedings of IEEE INFOCOM*, August 1997.

[5] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of SIGCOMM*, Stanford, CA, 1996.

[6] X. Li, S. Paul, and M. Ammar. Multi-session rate control for layered video multicast. In *Proceedings of Symposium on Multimedia Computing and Networking*, San Jose, CA, January 1999.

[7] M. Handley, S. Floyd, and B. Whetten. Strawman Specification for TCP Friendly (Reliable) Multicast Congestion Control(TFMCC). IRTF Reliable Multicast Research Group Meeting in Pisa, Italy, June 1999.

[8] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. Technical Report, ACIRI, Feb 2000.

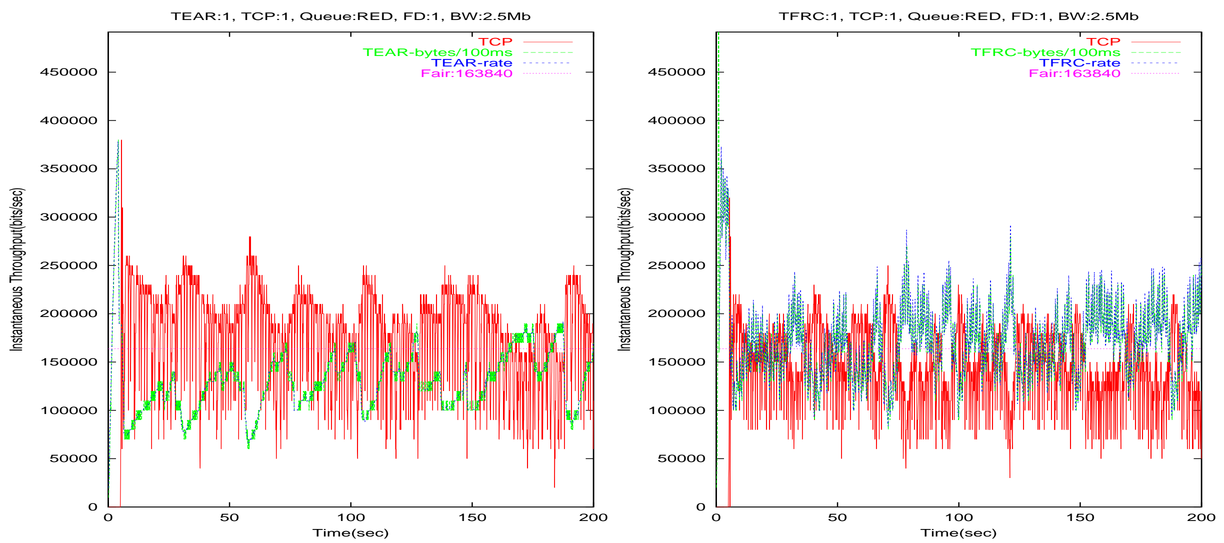


Figure 11: Bottleneck:2.5Mbs, Droptail router, Feedback 1 RTT: Left (1 TEAR, 1 TCP), Right (1 TFRC,1 TCP)

[9] S. Cen, C. Pu, and J. Walpole. Flow and congestion control for internet streaming applications. In *Proceedings of Multimedia Computing and Networking*, January 1998.

[10] R. Rejaie, M. Handley, and D. Estrin. RAP:An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proceedings of IEEE INFOCOM*, 1999.

[11] S. Jacobs and A. Eleftheriadis. Real-time dynamic rate shaping and control for internet video applications. In *Proceedings of the Workshop on Multimedia Signal Processing*, pages 23–25, June 1997.

[12] Jia-Ru Li, Sungwon Ha, and Vadavur Bhargavan. HPF: A transport protocol for supporting heterogeneous packet flows in the Internet. Research Paper, Coordinated Sciences Laboratory, University of Illinois at Urbana-Champaign, 1998.

[13] R. Talpade and M. Ammar. Single connection emulation (SCE): An architecture for providing a reliable multicast transport service. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Vancouver, BC, Canada, June 1995.

[14] J. Mahdavi and S. Floyd. TCP-friendly unicast rate-based flow control. Note sent to end2end-interest mailing list, 1997.

[15] D. Sisalem and H. Schulzrinne. The loss-delay adjustment algorithm: A TCP-friendly adaptation scheme. In *Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[16] T. Turletti, S. Parisis, and J. Bolot. Experiments with a layered transmission scheme over the Internet. Technical Report RR-3296, Inria, France, 1997.

[17] J. Padhye, J. Kurose, D. Towsley, and R. Koodli. A model based TCP-friendly rate control protocol. In *Proceedings of The Ninth International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 99)*, 1999.

[18] W.Tan and A.Zakhor. Real-time Internet video using error resilient scalable compression and TCP-friendly transport protocol. *IEEE Transactions on Multimedia*, 1(2):172–186, June 1999.

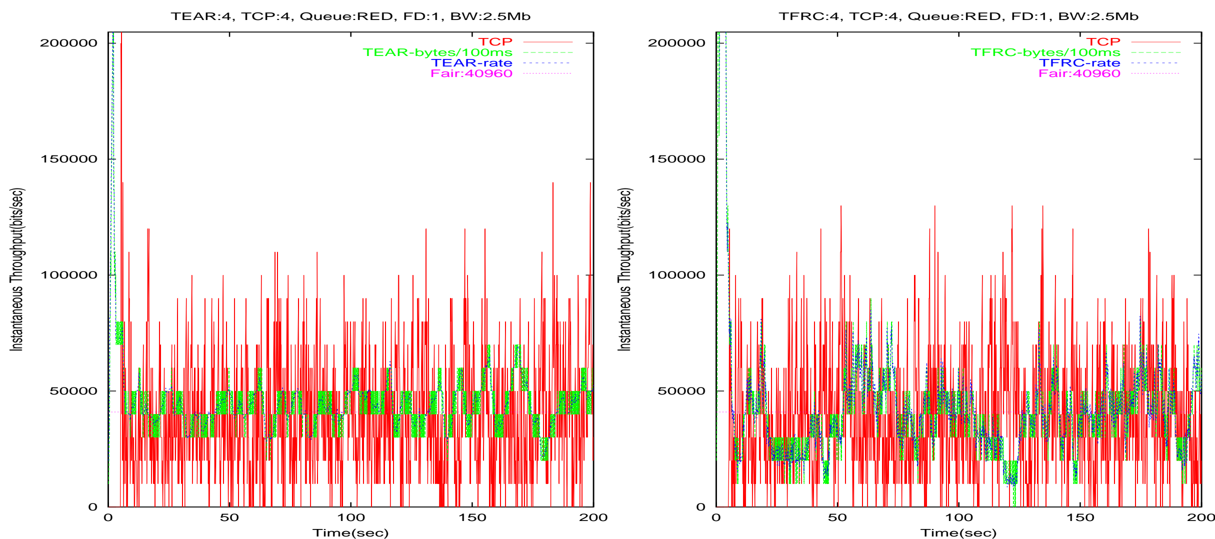


Figure 12: Bottleneck:2.5Mbs, RED router, Feedback 1 RTT: Left (4 TEARs, 4 TCPs), Right (4 TFRCs, 4 TCPs)

[19] S. Floyd. Connections with multiple congested gateways in packet-switched networks part 2: Two-way traffic. Unpublished draft, 1991.

[20] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behaviour of the TCP congestion avoidance algorithm. *ACM Computer Communication Review, 27(3)*, July 1997.

[21] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its emprical validation. In *Proceedings of SIGCOMM*, 1998.

[22] S. Ramesh and I. Rhee. Issues in model based flow control. IRTF Reliable Multicast Research Group Meeting in Herndon, Virginia, November 1999.

[23] S. Ramesh and I. Rhee. Issues in model based flow control. Technical Report TR-99-15, Department of Computer Science, North Carolina State University, November 1999.

[24] T. Tuan and K. Park. Multiple time scale congestion control for self-similar network traffic. In *Performance Evaluation*, 1999.

[25] T. Tuan and K. Park. Multiple time scale redundancy control for QoS-sensitive transport of real-time traffic. In *Proceedings of INFOCOM (to appear)*, 2000.

[26] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. In *Proceedings of 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1996.

[27] W.E. Leland, M.S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 1994.

[28] V. Paxson and S. Floyd. Wide-area traffic: Failure of poisson modeling. In *Proceedings of the ACM SIGCOMM*, pages 257–268, 1994.

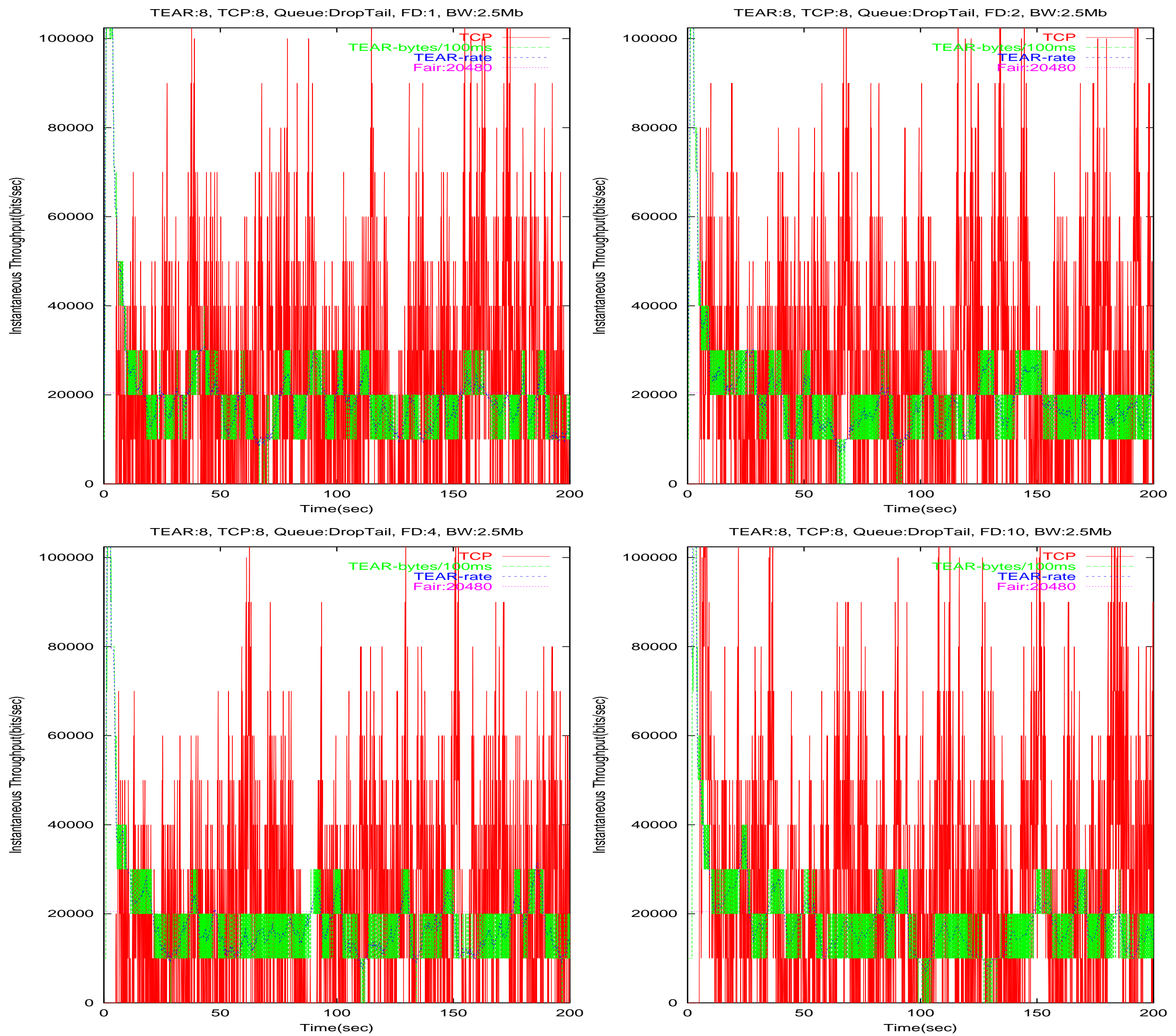


Figure 13: TEAR: Bottleneck 2.5Mbs, Droptail router, 8 TEARs, 8 TCPs with Feedback latency 1, 2, 4, and 10 RTTs

[29] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-similarity through high-variability: Statistical analysis of ethernet LAN traffic at the source level. In *Proceedings of the ACM SIGCOMM*, pages 100–113, 1995.

[30] I. Rhee, N. Balaguru, and G. Rouskas. MTCP: Scalable TCP-like congestion control for reliable multicast. In *Proceedings of IEEE INFOCOM*, New York, NY, March 1999.

[31] S. J. Golestani and K. K. Sabnani. Fundamental observations on multicast congestion control in the Internet. In *Proceedings of IEEE INFOCOM*, New York, NY, March 1999.

[32] B. Whetten. Target goals for RM congestion control algorithms. IRTF Reliable Multicast Research Group Meeting in George Mason, Virginia, December 1998.

[33] Supratik Bhattacharyya, Don Towsley, and Jim Kurose. The loss path multiplicity problem for multicast congestion control. In *Proceedings of IEEE INFOCOM*, 1999.

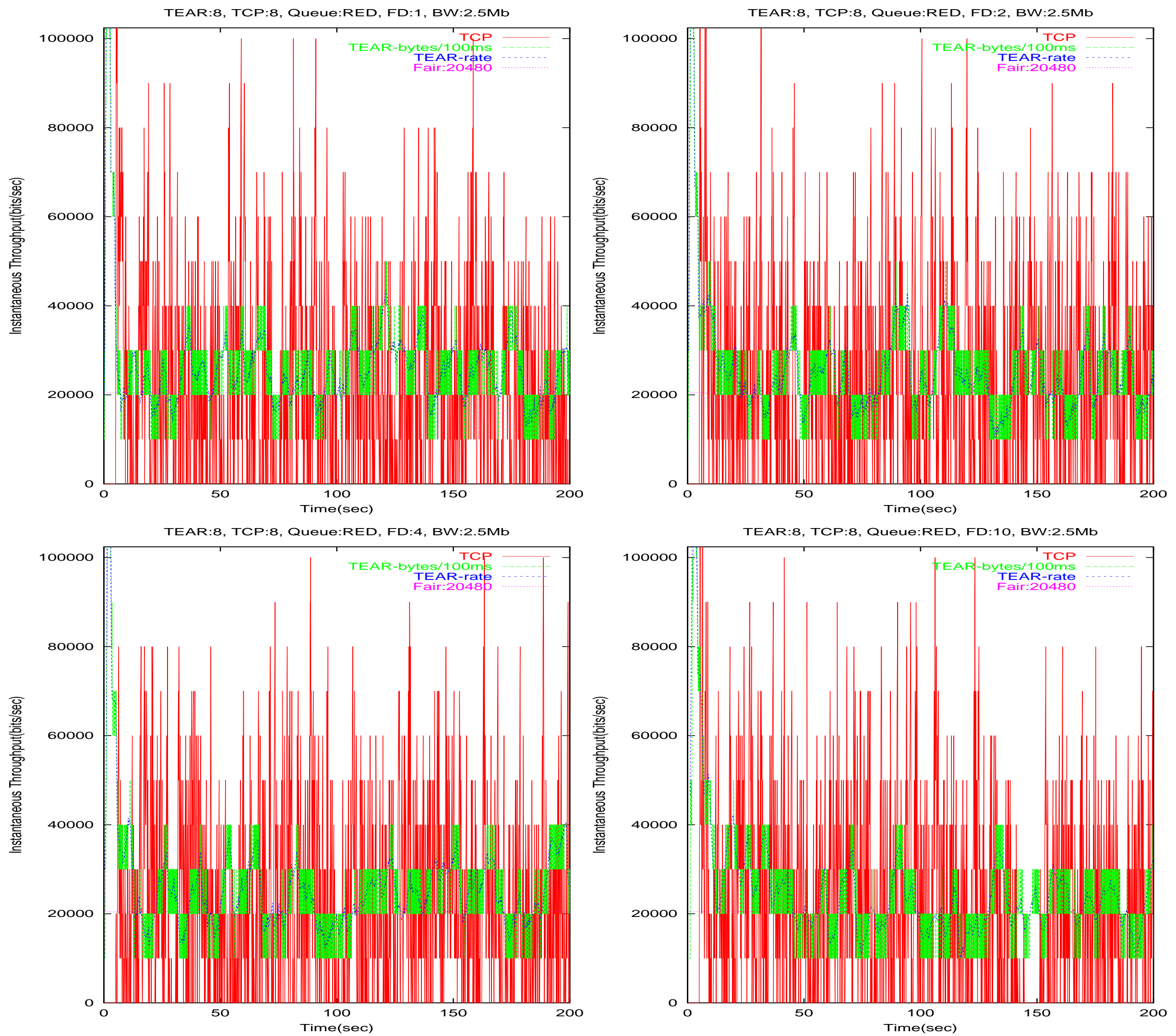


Figure 14: TEAR: Bottleneck 2.5Mbs, RED router, 8 TEARs, 8 TCPs with Feedback latency 1, 2, 4, and 10 RTTs

[34] B. Whetten, L. Vicisano, R. Kermode, M. Handley, S. Floyd, and M. Luby. Reliable multicast transport building blocks for one-to-many bulk-data transfer. IETF Internet-Draft, draft-ietf-rmt-buildingblocks-01.txt, June 1999.

[35] L. Vicisano, L. Rizzo, and J. Crowcroft. TCP-like congestion control for layered multicast data transfer. In *Proceedings of IEEE INFOCOM*, August 1997.

[36] D. Rubenstein, S. Kasera, D. Towsley, and J. Kurose. The impact of multicast layering on network fairness. In *Proceedings of SIGCOMM*, August 1999.

[37] L. Wu, R. Sharma, and B. Smith. Thinstreams:an architecture for multicasting layered video. In *Proceedings of The Seventh International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 97)*, May 1997.

[38] S Bhattacharjee, K. L. Calvert, and E. W. Zegura. On active networking and congestion. Technical Report GIT-CC-96-02, College of Computing, Georgia Tech, 1996.

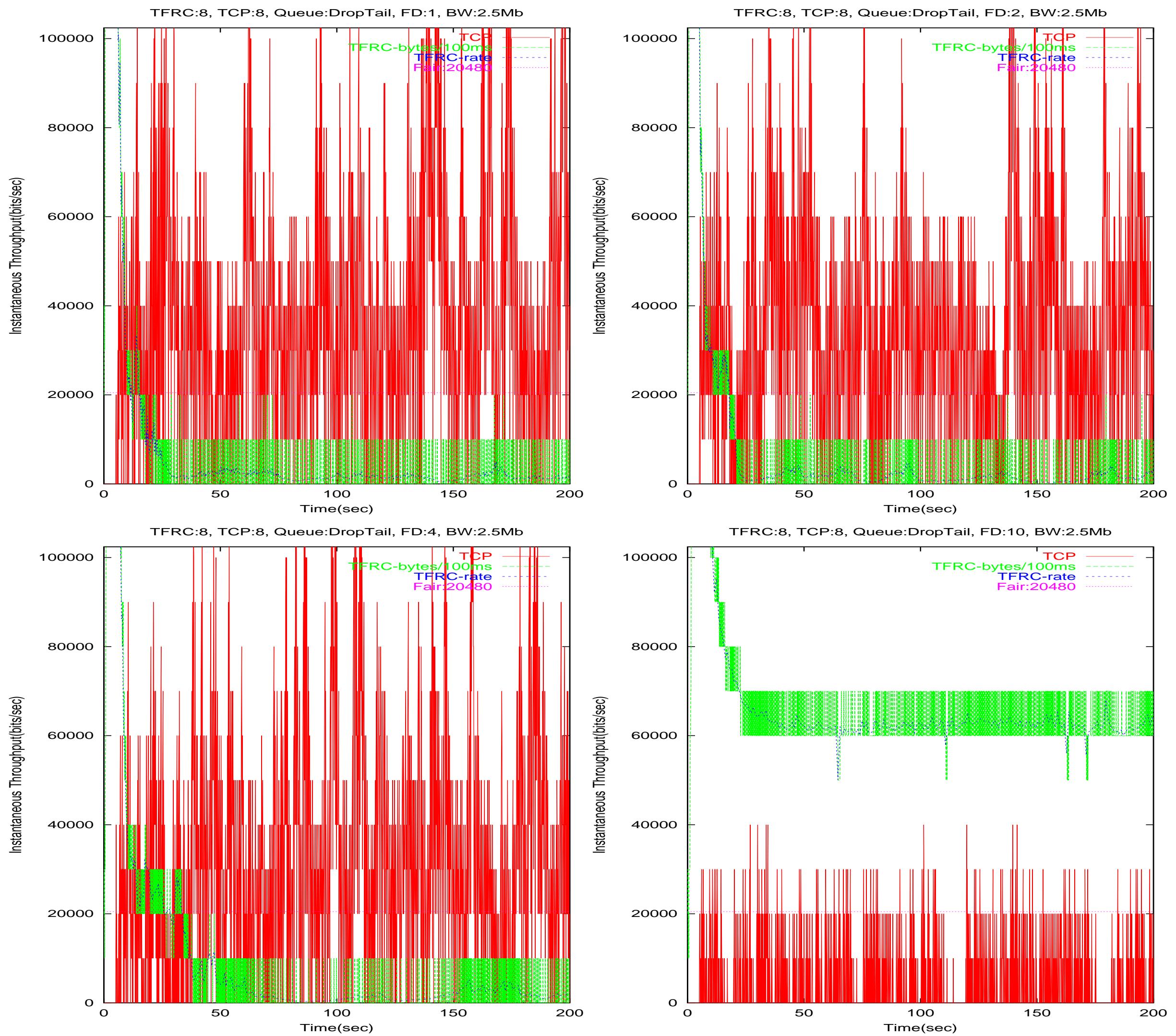


Figure 15: TFRC: Bottleneck 2.5Mbs, Droptail router, 8 TFRC, 8 TCPs with Feedback latency 1.5, 2, 4, and 10 RTTs

[39] M. Luby, L. Vicisano, and T. Speakman. Heterogeneous multicast congestion control based on router packet filtering. IRTF Reliable Multicast Research Group Meeting in Pisa, Italy, June 1999.

[40] K. Yano and S. McCanne. The Breadcrumb forwarding service and the digital fountain rainbow: Toward a TCP-friendly reliable multicast. Technical Report UCB/CSD-99-1068, Computer Science Division, University of California, Berkeley, Oct 1999.

[41] Kang-Won Lee, Sungwon Ha, Jia-Ru Li, and Vadavur Bhargavan. MHPF: A multicast architecture for multimedia communications in the Internet. IRTF Reliable Multicast Research Group Meeting in Herndon, Virginia, November 1999.

[42] L. Vicisano. Simulations of PGM stategies for congestion control. IRTF Reliable Multicast Research Group Meeting in Herndon, Virginia, November 1999.

[43] T. Speakman et al. PGM reliable transport protocol specification. Internet Draft draft-speakman-pgm-spec-03.txt, June 1999.

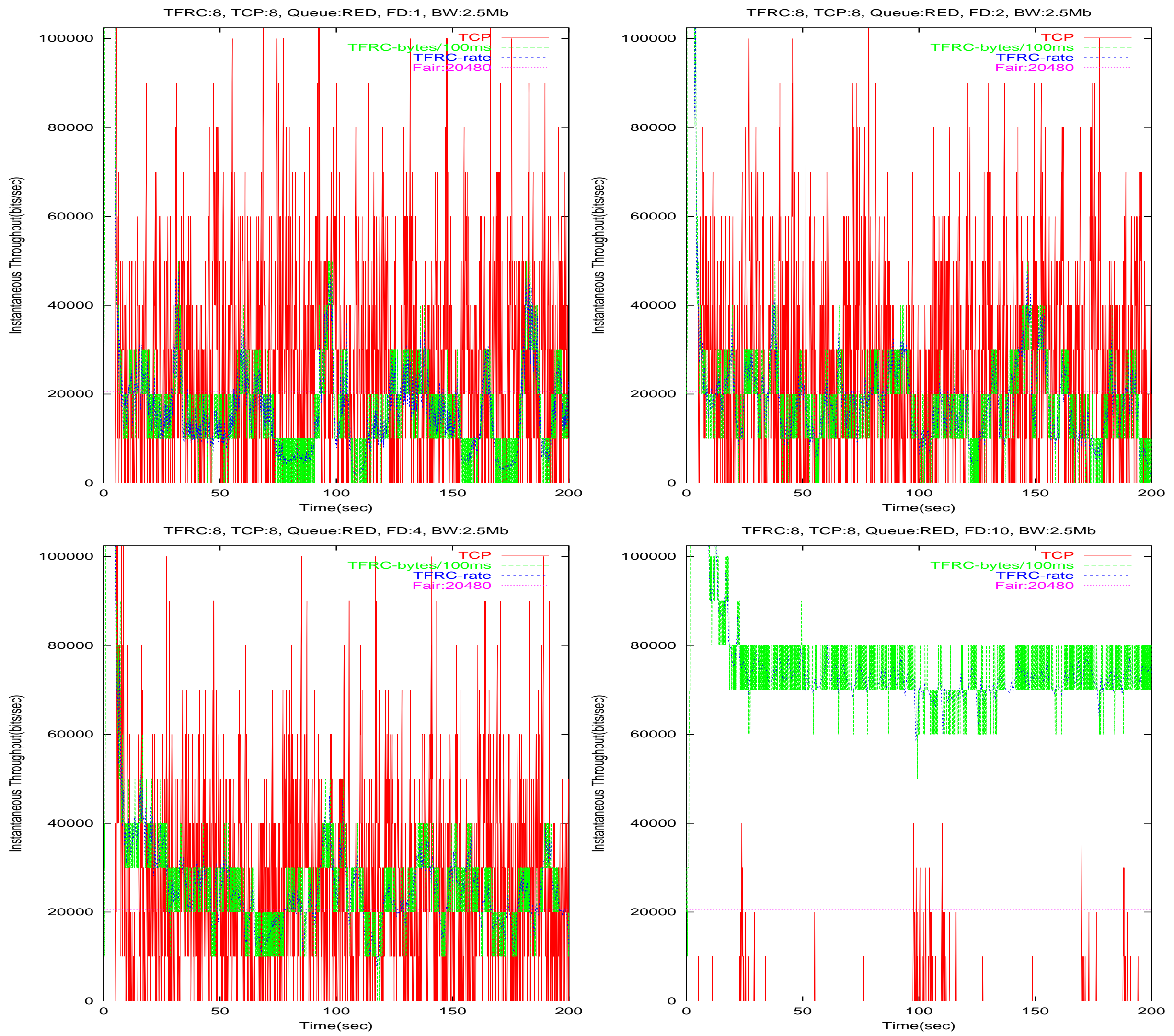


Figure 16: TFRC: Bottleneck 2.5Mbs, RED router, 8 TFRC, 8 TCPs with Feedback latency 1.5, 2, 4, and 10 RTTs

[44] J. Martin, A. Nilsson, and I. Rhee. The incremental deployability of RTT-based congestion avoidance over high-speed TCP Internet connections. In *to appear in Proceedings of ACM SIGMETRICS*, June 2000.