Media-Aware Rate Control

Zhiheng Wang[*]
zhihengw@eecs.umich.edu
University of Michigan
Ann Arbor, MI 48109

Sujata Banerjee
sujata@hpl.hp.com
Hewlett-Packard Laboratories
Palo Alto, CA 94304

Sugih Jamin[†]
jamin@eecs.umich.edu
University of Michigan
Ann Arbor, MI 48109

December 1, 2004

Abstract

Streaming media transfers over the Internet are expected to behave in a TCP friendly manner and react appropriately to congestion similar to the way TCP does. However, our work shows that existing "TCP-friendly" protocols are not necessarily "media friendly". In this paper, we present our findings on the impact of the TFRC protocol on streaming media quality. We propose the MARC (Media Aware Rate Control) protocol that, unlike TFRC, exhibits significant tolerance towards transient changes in background workload, while still maintaining TCP-friendliness.

1 Introduction

With emerging advanced technologies on both end hosts and network connections, the deployment of multimedia applications over the Internet has increased rapidly in recent years [1, 2, 3]. Media applications are highly resource intensive, with respect to computing resources, networking resources, as well as storage resources. There are two options for media content transfers: downloading through bulk data transfer or streaming to a receiver from a media source. Each option faces unique challenges. In this paper we concentrate on providing streaming media transfers with end-to-end network supports. Streaming is the only option in the following two scenarios:

[*]Part of this work was performed while Zhiheng Wang was an intern at HP labs.

[†]This research is supported in part by the NSF CAREER Award ANI-9734145, the Presidential Early Career Award for Scientists and Engineers (PECASE) 1998, the Alfred P. Sloan Foundation Research Fellowship 2001, and by the United Kingdom Engineering and Physical Sciences Research Council (EPSRC) Grant no. GR/S03577/01. Part of this work was completed when Sugih Jamin was visiting the Computer Laboratory at the University of Cambridge.

- For real time transfer of live events, streaming is the only option because the data needs to be transmitted as soon as they are generated. These applications are non-interactive and can tolerate only a short one-way delay. Live interactive applications such as video conferencing or multi-player games have stringent constraints in that the amount of delay they can tolerate is very small.
- Stored (non-live) media can be downloaded rather than streamed if the user provides sufficient lead time between the download request and desired playback time. However, for storage constrained clients, particularly light weight handheld clients, streaming is the only option even with stored media. To minimize the storage requirements of such clients, media data must be sent at the same rate as it is consumed at the receiver.

The implication of streaming is that network sending rate must be closely matched to the data generation rate and the rate at which the data is consumed by the client application. If these rates do not match, receiver buffers can overflow or underflow and the end user perceived media quality can suffer. Most applications have a small startup buffer at the receiver which is filled before media playout can begin. This is done to absorb variable network delays to ensure continuous playback. For interactive applications, the size of this buffer must be very small to reduce interaction latency.

Currently, wide-spread deployment of high fidelity streaming media on the Internet is hindered by the best-effort nature of the networks. Streaming applications using UDP or TCP for data transfer are under the risk of either overloading network resources or receiving lower perceived quality [4, 1]. TCP congestion control interferes with streaming media throughput requirements by introducing sudden drops in the sending rate when congestion occurs. There have been a number of attempts (e.g., [5, 6, 7, 8]) to reduce these sudden fluctuations, while complying with TCP congestion control principle of avoiding overloading network resources [9]. Subsequent to these efforts, the concept of *TCP-friendliness* [5] was introduced. TCP-friendliness ensures the compatibility between a proposed protocol and existing TCP. "A congestion control mechanism is *TCP-friendly* if it displays congestion control behavior that, on time scales of several round-trip times (RTTs), obtains roughly the same throughput as a TCP connection in steady-state when the available bandwidth does not change with time" [6]. Very few previous efforts on developing TCP friendly protocols have examined whether they are *media friendly*. We are not aware of any systematic study on the impact of congestion control mechanisms on user's perceived quality. In this work, we try to evaluate TFRC's effectiveness on meeting streaming media quality by considering both the streaming application characteristics and transport level design. Our study reveals some problems which could not have been discovered by examining either of these components individually. Based on these experiments, we devise a new mechanism called Media Aware Rate Control (MARC) that is TCP friendly as well as media friendly.

2 Related Work

Congestion control has been the topic of a large number of studies [9, 10, 11, 12, 13, 5, 6, 7, 8]. For their robustness and efficiency, TCP and its variants [10, 11, 7, 8, 14] have become the dominant transport level congestion control mechanism on the current Internet. A measurement study in 2001 [15] reported that TCP constitutes more than 90% of the Internet backbone traffic.

Despite its efficiency in bulk data transfer, TCP is not suitable for streaming media transfers [5]. Compared to other applications, streaming media imposes more constraints on network delay and loss performance. Authors of [3] enumerated a number of challenges on video streaming over the Internet. Authors of [2] further explored the research issues arising from the networks' having to carry large amounts of media content. Among these challenges, researchers in [1] found that delayed and lost packets equally affect perceived video quality.

TCP halves its congestion window at the onset of congestion. Such behavior introduces additional packet delay to streaming applications. To prevent this, a series of TCP-friendly slowly-responsive congestion control mechanisms have been developed in [5, 7, 8]. On receiving congestion signals, these mechanisms reduce their congestion window by less than half but then increase their congestion window more slowly, which makes them TCP-friendly.

Current congestion control schemes in the literature fall into two major categories: (1) additive-increase multiplicative-decrease (AIMD) [16] and its variants such as those in [9, 8, 7, 17, 18, 14], and (2) rate-based schemes such as [11, 5]. Rate-based schemes can be further categorized into *AIMD-based* [11] and *equation-based* [5]. Rate Adaptation Protocol (RAP) is proposed in [8]. RAP follows the AIMD principle in its rate increase/decrease algorithm. Fine grain rate adjustment based on path RTT is used to reduce rate fluctuation. In [7], the authors generalize the AIMD principle and introduce a family of new AIMD congestion control mechanisms. They also establish the precondition with which these mechanisms can compete fairly with TCP. In [19], the authors argue that *TCP-friendliness* should be considered over longer time scales than path RTT. The authors propose to modify the TCP response function by taking advantage of the Explicit Congestion Notifi cation (ECN) signal [20, 21]. On receiving the ECN signal, their scheme decreases the congestion window less aggressively and as a result, alleviates the additional delay caused by reducing the sending rate. TCP-Friendly Rate Control (TFRC) is proposed in [5]. TFRC is an equation-based congestion control mechanism that uses the TCP throughput function presented in [22]. Studies in [6, 23, 24] have examined the performance of these TCP-friendly congestion control mechanisms and their compatibility with TCP. All of these studies try to reduce congestion window oscillation to provide streaming media with better perceived quality, while maintaining TCP-friendliness. Nonetheless, the requirements of streaming media has not been fully addressed in these studies.

The rest of the paper is organized as follows. First we evaluate the smoothness of TFRC for both bulk data transfer and streaming applications in Section 3. A token-based mechanism is proposed in Section 4. The evaluation of this mechanism is discussed in Section 5. We conclude our work in Section 6 and present some future directions in Section 7.

3 Performance Analysis of TFRC

Authors of [6] found that the performance of a slowly-responsive congestion control mechanism highly depends on the packet loss patterns. In this section we examine TFRC's smoothness under several packet loss models. Different from [24], where the authors compared the transient behavior of several TCP-friendly congestion control schemes, our study focuses on TFRC's smoothness and its implication for streaming applications. To closely examine TFRC's behavior when streaming media is present we use CBR-like traffi c, in addition to bulk data transfer, in our study. We have also conducted a preliminary perceptual video quality tests, described in Appendix B.

3.1 TCP-Friendly Rate Control

Among the proposed slowly-responsive congestion control mechanisms, *TCP-friendly Rate Control* (TFRC), an equation-based congestion control scheme for unicast applications proposed in [5], is a promising mechanism for streaming media transfer. To facilitate our description in the rest of the paper, in this section we give an overview of TFRC. TFRC's goal is to provide streaming application with smoother throughput profi le while maintaining TCP-friendliness. TFRC is based on the TCP throughput function proposed in [22]:

$$T = \frac{s}{R\sqrt{\frac{2p}{3}} + t_{RTO}(3\sqrt{\frac{3p}{8}})p(1+32p^2)}. \qquad (1)$$

The upper bound of the sending rate T of a TCP connection is a function of the packet size s, path round-trip time R, steady state loss rate p and the retransmit timeout value t_{RTO}. By measuring the loss rate p, and the path round-trip time R, TFRC is able to determine the *TCP-friendly* congestion window by using Eqn. 1. The loss rate p is measured at a TFRC receiver and a feedback message containing p is periodically sent back to the TFRC sender. To prevent halving the congestion window on single-packet losses as TCP does, TFRC uses *weighted average loss interval* (WALI) [25] to calculate the average *loss event rate* p from multiple congestion signals within a single RTT. Smooth traffic profile is achieved in TFRC by averaging p, and therefore T, over the past N round-trip times, where N is the loss history size. N affects TFRC's smoothness as well as it is responsiveness to persistent congestion. TFRC's responsiveness to sudden loss rate increases can be tuned by choosing the appropriate N. It has been observed that TFRC could react too slowly to sudden loss rate decreases. To address this issue, *history discounting* is added as a component of WALI. With history discounting, TFRC smoothly decreases the weights for older loss intervals if the interval between two subsequent packet losses is large.

In addition to WALI, authors in [6] proposed to add *self-clocking* to TFRC to achieve faster response to incipient persistent congestion. In [5], the authors proposed to limit TFRC's sending rate to be no more than twice of the received rate in previous feedback interval. With self-clocking, during the round-trip time following a packet loss, TFRC further limits its sending rate to at most the rate at which data is received by the receiver in the previous round trip time; if no packet loss is detected, its sending rate will be no larger than 1.5 times the rate seen by the receiver in the previous round trip.[1] With self-clocking, TFRC takes into account the rate seen at the receiver. Since the received rate is affected by variable queueing times on intermediate nodes, TFRC's congestion window is also affected by these variable queueing times. We will further discuss how self-clocking affects TFRC's throughput smoothness in Section 3.6.

We define the following terms for use in this paper:

Fair share: the amount of bandwidth allocated to a connection at the bottleneck gateway so that the bottleneck bandwidth is distributed evenly among different flows. This definition assumes a single bottleneck.

Calculated allowed-rate: the rate calculated by TFRC with Eqn 1.

Self-clocked rate: TFRC's self-clocked rate is the rate after applying rate change constrains such as self-clocking. To have a consistent description, we also use the term of self-clocked rate to describe TCP's congestion window. For this purpose, we need to normalize TCP's congestion window to its equivalent self-clocked rate. This is done with the following equation:

$$S = \frac{cwnd}{R}, \tag{2}$$

where $cwnd$ is the TCP congestion window in bytes, S is the translated TCP self-clocked rate in Bps, and R is the path round-trip time in second. S is calculated everytime TCP changes its congestion window.

Data rate: the rate at which applications generate their data.

Sending rate: the outgoing data rate for a connection measured at the sender. The sending rate of a connection can never exceed its self-clocked rate. In the case of bulk data transfer, sending rate is equivalent to the self-clocked rate.

Throughput: the incoming data rate for a connection measured at the receiver.

3.2 Simulation Configurations

We use *ns* for our performance analysis of TFRC. In all the simulations, we use a "dumbbell" topology as shown in Fig. 1. The bottleneck link is between *R1* and *R2*. The bottleneck bandwidth is 1.5 Mbps in the default setting. Access link bandwidth is sufficiently provisioned so that congestion only occurs on the bottleneck link. The

[1]Self-clocking is turned off in ns's default settings.

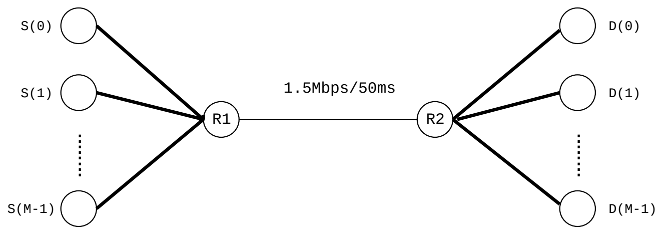


Figure 1: Simulation Topology.

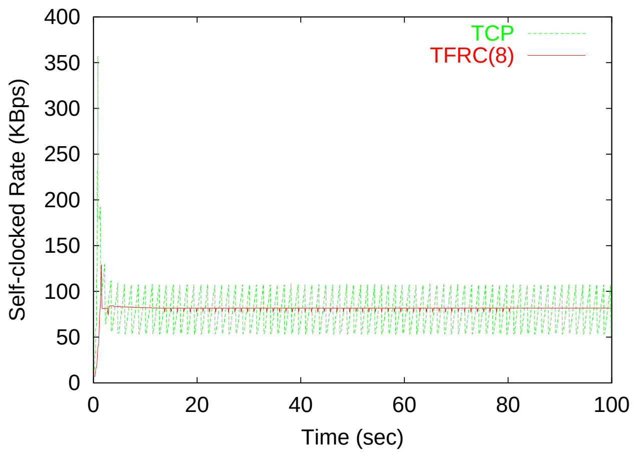


Figure 2: Self-clocked rates for TFRC(8) and TCP with 1% periodic loss.

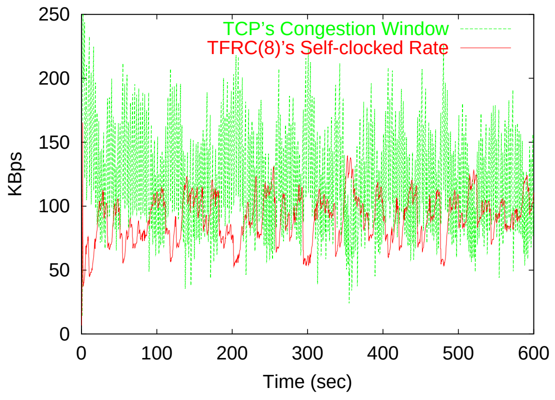
one-way delay for the bottleneck link is 50 ms and for all access links, 10 ms. By default, RED [26] is deployed at the bottleneck router. We set the queue size to twice the bandwidth delay product. The threshold settings for RED are taken from [6].

All traffi c flows are generated at source nodes *S*(i), $i = 0, ..., M-1$ and received at nodes *D*(i), $i = 0, ..., M-1$. We use TCP-SACK with suffi ciently large receiver advertised window to represent TCP connections. We also assume sources have infi nite send buffers such that data that cannot be sent out during congestion will be buffered for later transmission. For TFRC, we turn on self-clocking [6] and history discounting [5]. We use "TFRC(N)" to denote TFRC with a loss history size of N. We will discuss N more thoroughly in Section 3.4. We use both bulk data transfer and constant bit rate (CBR) sources as data source. Some of our simulations use Pareto distributed ON-OFF UDP flows as *background traffic* to model long-range dependent Internet traffi c [27, 28]. Each of these ON-OFF traffi c flows has a mean ON time of 1 second and a mean OFF time of 2 seconds. The data rate during the ON time is 62.5 KBps. The shape parameter for the Pareto distribution is 1.05. The packet size is 1 KB for all the connections.

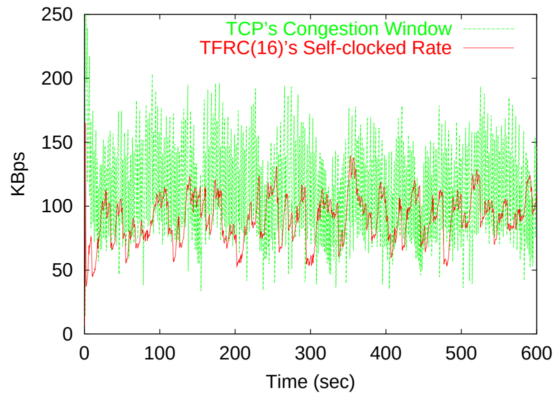
3.3 Smoothness of bulk data transfer

We fi rst look at periodic losses. Studies in [5, 6, 24] have shown that when packet loss is periodic, TFRC could greatly improve the smoothness of a connection's self-clocked rate. Fig. 2 illustrates the self-clocked rate for TFRC(8) and TCP with 1% periodic loss rate. We clearly see that TFRC(8) indeed effectively dampens the "saw-tooth" self-clocked rate fluctuation exhibited by TCP. By averaging the loss event rate with WALI, TFRC can accurately predict the fair share and adapt to the corresponding self-clocked rate when losses are periodic.

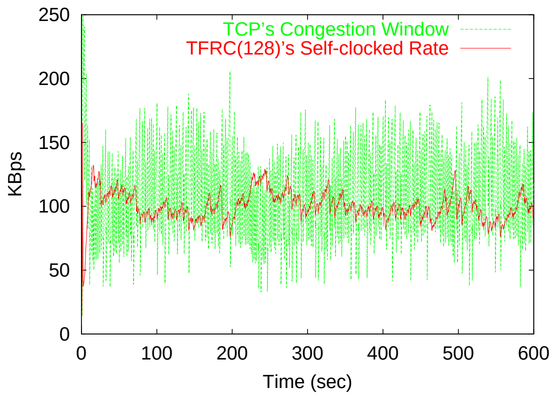
We then examine the scenario with one TFRC(8) flow and one TCP flow competing for the bottleneck bandwidth. Fig. 3(a) plots the self-clocked rate of TCP and TFRC during the simulation. In the small time scale, e.g., several RTTs, TFRC(8) has a smoother self-clocked rate than TCP. However, in time scale of seconds, com-



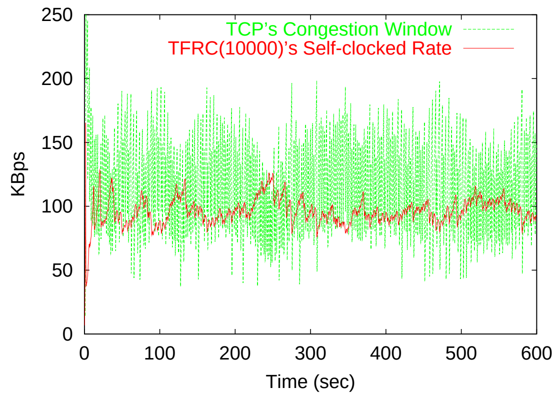
(a) N=8, CoV of TFRC's self-clocked rate is 0.19.



(b) N=16, CoV of TFRC's self-clocked rate is 0.14.



(c) N=128, CoV of TFRC's self-clocked is 0.12.



(d) N=10,000, CoV of TFRC's self-clocked rate is 0.11.

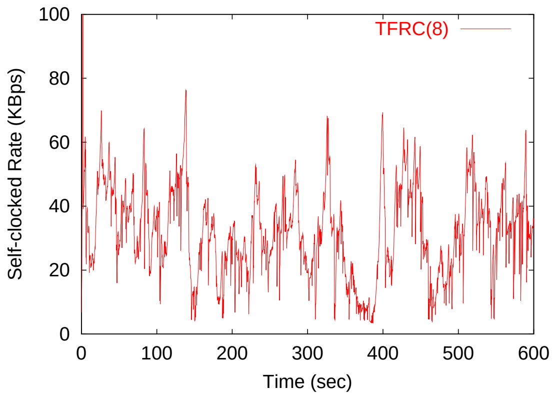
Figure 3: Self-clocked rate when a TFRC(N)-FTP flow competes with a TCP-FTP flow.

peting TCP traffic still causes TFRC(8) to experience fluctuations in its self-clocked rate. This simple example illustrates that while TFRC is designed to reduce the abrupt changes seen by TCP at time scale of RTTs, the smoothness of its self-clocked rate is still heavily affected by other concurrent flows at a larger time scale.

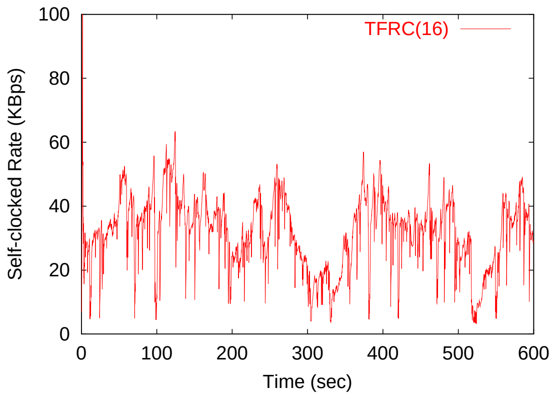
To improve TFRC's smoothness, we can increase the loss history size N used by WALI. A larger N corresponds to a larger averaging window, which causes TFRC to response more slowly to packet loss. We repeat the simulation above with $N = 16$, $N = 128$, and $N = 10,000$. The outcome of the simulations are shown in Figs. 3(b), (c), and (d) respectively. Note that N affects the smoothness of TFRC's self-clocked rate. We make the following observations by comparing the sending rate with different N.

- TFRC(N) with larger N has relatively smoother sending rate.
- In turn, TFRC's smoother traffic profile results in smoother profile of the competing TCP traffic. However, in smaller time scale, TCP's self-clocked rate oscillates more than that of TFRC.
- Even with a sufficiently large N, e.g., $N = 10,000$, TFRC(N) is still not able to achieve a smooth traffic profile. For example, the ideal sending rate in our simulations would be around the fair share of 94 KBps.

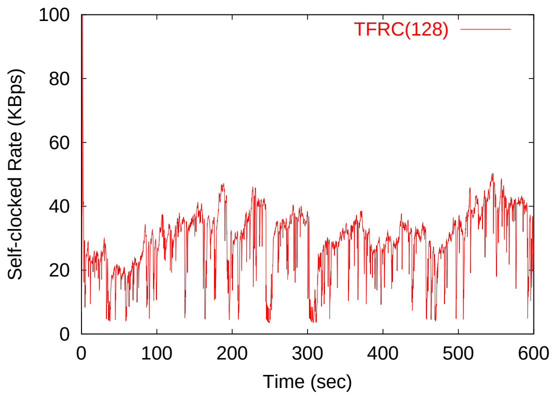
Authors of [28] pointed out that aggregate LAN traffic shows long-range dependency. To incorporate this characteristic into our simulation, we introduce five Pareto distributed ON-OFF flows, in addition to the competing TCP traffic, as background traffic [27]. Fig. 4 shows TFRC(N)'s sending rate with N = 8, 16, 128, and 10,000. Similar to the observations made above, TFRC(N) with larger N has smoother self-clocked rate. However, in-



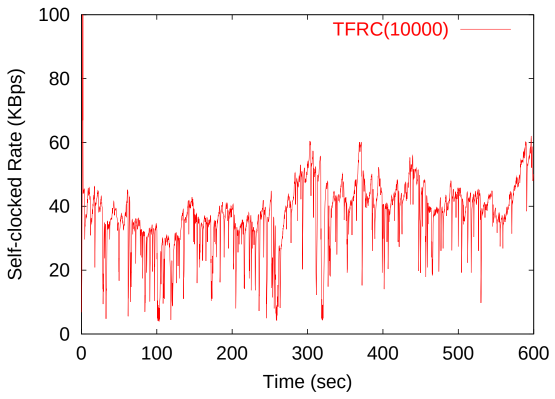
(a) N=8, CoV of TFRC's self-clocked rate is 0.42.



(b) N=16, CoV of TFRC's self-clocked rate is 0.35.



(c) N=128, CoV of TFRC's self-clocked rate is 0.34.



(d) N=10,000, CoV of TFRC's self-clocked rate is 0.27.

Figure 4: The self-clocked rate of TFRC(N)-FTP in the presence of long-range dependent background traffic.

creasing N alone can not completely remove the oscillations in sending rate seen by TFRC(N). Increasing N also results in a less responsive reaction to persistent congestion, even with self-clocking [6].

3.4 Choosing Loss History Size N

While [5] suggests $N = 8$ as a default value for loss history size, the appropriate value of N has not been carefully studied. We repeat the simulations in Fig. 4 with N taken from a wide range of values. For each N, we repeat the simulation 100 times and calculate the average coefficient of variation of TFRC(N)'s self-clocked rate. The results are shown in Fig. 5, along with their 95% confidence intervals. This figure indicates diminishing return effect of increasing N to improve TFRC(N)'s smoothness. In our simulations, little improvement can be achieved when N is increased beyond 128. Even with $N = 10,000$, non-trivial self-clocked rate fluctuation still exists.

In summary, compared to TCP, TFRC does alleviate self-clocked rate fluctuations to a large degree under the loss scenarios studied, especially within time scales on the same order as path RTT. Its ability to prevent self-clocked rate oscillation in a larger time scale, to satisfy the requirements of streaming applications, is questionable. A larger loss history size N can improve TFRC(N)'s performance. However, increasing the loss history size alone does not sufficiently reduce fluctuation in self-clocked rate. We use $N = 128$ in the rest of our study and denote TFRC(128) simply as TFRC.

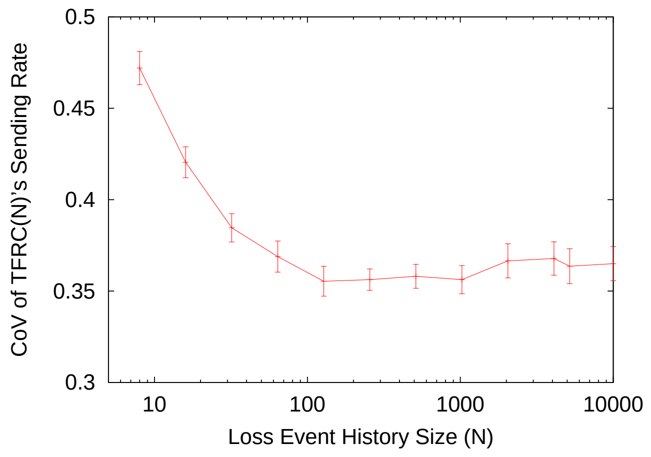
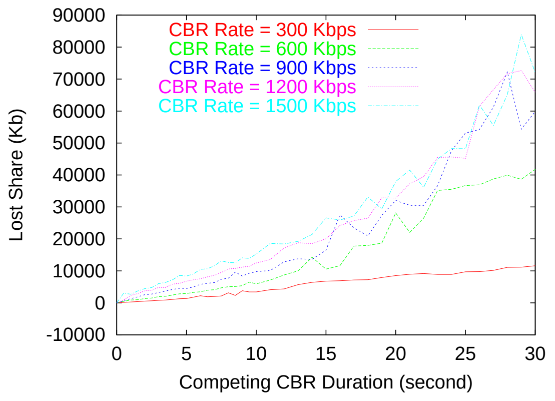
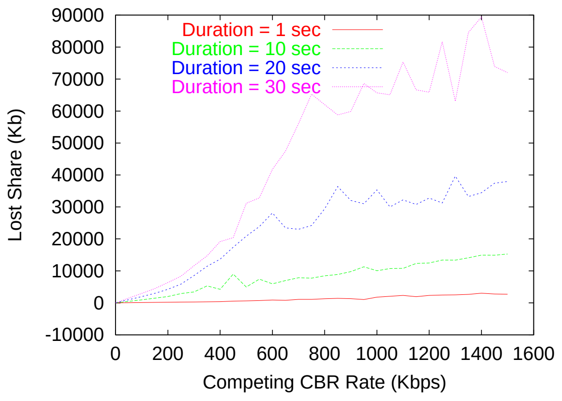


Figure 5: The effect of N on TFRC sending rate variation.



(a) Different CBR traffi c duration.



(b) Different CBR traffi c rate.

Figure 6: TFRC bandwidth loss when competing with transient CBR traffi c.

3.5 Reacting to Transient Congestion

In this section, we examine how transient changes in available bandwidth affects TFRC's sending rate. We inject a transient CBR (UDP) flow during the steady state of a TFRC flow. By steady state, we mean TFRC's calculated allowed-rate remains constant when there is no changes in available bandwidth. For our evaluation purposes, we defi ne a connection's *lost share* during congestion as its cumulative sending rate reduction (measured in bytes) caused by other competing traffi c during the congestion period. This indicates how much sending rate the tested flow losses due to interference from the CBR flow. In our simulation, the tested flow is a TFRC flow. The duration of the CBR flow ranges from 1 second to 30 seconds with its data rates ranging from 0 KBps to 1.5 Mbps in different simulations. Each run of the simulation lasts for 10 minutes. TFRC traffi c starts at time 0 and the CBR flow starts at the 400th second. We note that in each simulation, TFRC always manages to enter its steady state before the CBR flow starts.

Fig. 6 illustrates the lost share of a TFRC connection when there is competing CBR traffi c. Fig. 6(a) shows TFRC connection's lost share for different CBR traffi c duration. Fig. 6(b) shows the results for different CBR data rates. We expect TFRC connection's lost share to be smaller than the cumulative throughput of the CBR traffi c because TFRC averages the loss event rate—so it reduces its sending rate slowly at the onset of the CBR traffi c. However, Fig. 6(a) shows that when the duration of the CBR traffi c is small, the TFRC connection's lost share is no less than the cumulative CBR traffi c's throughput. When the duration of the transient CBR traffi c increases, the TFRC connection's loss share shows a non-linear increase for various CBR rates. Fig. 6(b)

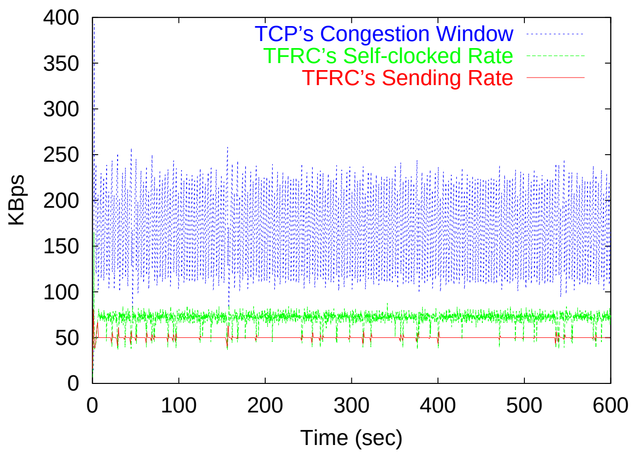


Figure 7: Unfairness when TFRC is used with TCP.

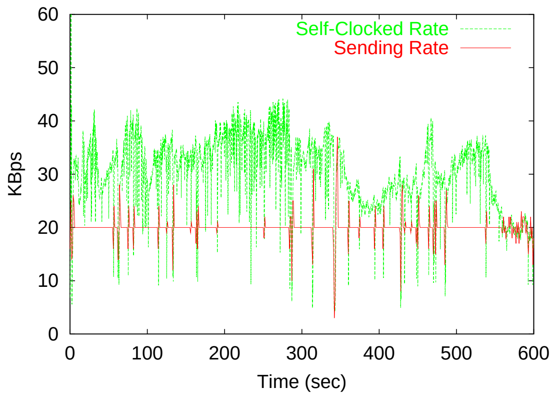
confirms this observation. For instance, when the CBR rate is 1,000 Kbps and lasts for 30 seconds, the TFRC connection's lost share is more than twice the amount when the CBR traffic lasts for 20 seconds. TFRC behaves as designed and responds to the increased background workload slowly. Also as designed, when congestion is removed and more bandwidth becomes available again, TFRC also increases its sending rate slowly. The longer the transient workload lasts, the longer it takes TFRC to fully recover from sending rate loss when available bandwidth increases. In other words, WALI slows down the recovery process while it dampens rate reduction.

3.6 Smoothness under CBR Traffic

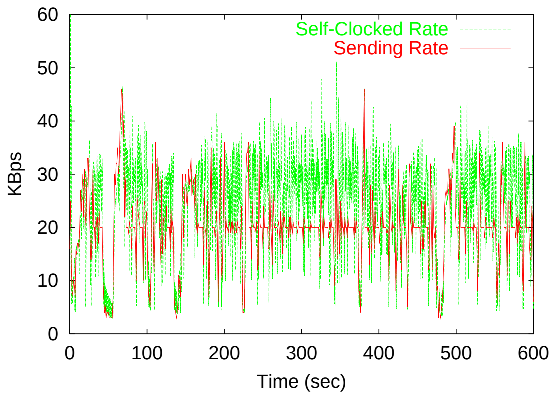
Previous studies of TFRC's performance use bulk data transfer with infinite amount of data to drive their simulations. TFRC, however, was designed to provide a smooth sending rate to streaming applications, which are usually rate limited. We model streaming media sources as CBR sources instead. Using rate-limited data sources, we discovered several performance issues not found in previous studies. For ease of exposition, we use "TFRC-FTP" to denote a TFRC connection for bulk data transfer and "TFRC-CBR" for a TFRC connection with CBR source. Similarly we also have "TCP-FTP" and "TCP-CBR".

We begin our discussion by looking at the scenario where only one TFRC-CBR and one TCP-FTP are competing for the bottleneck link. The application data rate for the TFRC-CBR connection is 50 KBps. Fig. 7 shows TCP's self-clocked rate, TFRC's self-clocked rate and its sending rate. In this simulation, the fair share for TCP and TFRC are both 94 KBps. However, because TFRC-CBR's sending rate is limited by its data rate, its sending rate is only about half of its fair share. When a streaming application's data rate is less than available bandwidth, TFRC's sending rate is not necessarily equal to its self-clocked rate. Instead, TFRC's self-clocked rate becomes an upper bound of its sending rate for each report cycle. As a result, TFRC's fair share is *under-utilized*. Meanwhile, the TCP-FTP connection takes over all the available bandwidth, which is larger than its fair share. In this case, TCP-FTP's fair share is *over-utilized*. Without knowing its fair share, a TCP-FTP connection always tries to increase its self-clocked rate until loss occurs. While such greedy behavior ensures high utilization level of the bottleneck link, it also causes TFRC-CBR's self-clocked rate to oscillate, even when the TFRC flow is sending less data than its fair share.

TFRC-CBR's self-clocked rate oscillation seen in Fig. 7 can be exacerbated by bursty background traffic. To illustrate, we introduce five ON-OFF Pareto traffic flows, in addition to the TCP-FTP flow, to our simulation scenario. The resulting self-clocked rate and sending rate for the TFRC-CBR connection are plotted in Fig. 8(a). The data rate of the CBR source is reduced to 20 KBps due to reduction of the available bandwidth. TFRC's sending rate can never be larger than its self-clocked rate. Thus the sending rate overlaps with the self-clocked



(a) TFRC without self-clocking.



(b) TFRC with self-clocking.

Figure 8: TFRC's self-clocked rate and sending rate with CBR traffi c.

rate whenever the self-clocked rate drops below the data rate (20 KBps in the fi gure). The sending rate is only distinguishable in the fi gure when it is smaller than the self-clocked rate. Similarly for the other fi gures in the rest of this paper. From this fi gure, we observe increased fluctuations on TFRC-CBR's self-clocked rate. The coeffi cient of variation for the self-clocked rate is 0.35, for the sending rate, 0.25. When the TFRC-CBR's self-clocked rate goes smaller than the application data rate, the TFRC-CBR's sending rate is reduced and data packets are delayed, which harms temporally sensitive applications such as streaming media.

The above observation indicates a new fairness issue when TFRC is used with streaming applications. Due to its limited data rate, a streaming application could end up using less bandwidth than a bulk transfer connection during non-congestion period. However, on congestion, a TFRC-CBR connection reacts to congestion signals similar to the way a bulk data transfer connection does. While this ensures TFRC's TCP-friendliness, it causes potential performance degradation for streaming applications, even if the streaming applications have been "good citizens".

As mentioned in Section 3.1, with self-clocking, TFRC's self-clocked rate is also affected by variable queueing times because variable queueing time would result in varied received rate at a receiver. In Fig. 8(a) we show a sample path of TFRC's self-clocked rate and sending rate when we turn TFRC's self-clocking off. Without self-clocking, the coeffi cient of variation of the self-clocked rate and sending rate are 0.33 and 0.15, respectively. Comparing Figs. 8(a) and (b), we see that self-clocking (Fig. 8(b)) introduces a large amount of sending rate fluctuation in the small time scales. Despite this negative effect, in our study we still use *TFRC with self-clocking* for the following reasons:

- The improvement on traffi c smoothness by simply removing *self-clocking* is limited. Without *self-clocking*, as shown in Fig. 8(b), TFRC's sending rate still suffers from transient reductions in self-clocked rate.
- Authors in [6] pointed out that when TFRC is used for bulk data transfer without *self-clocking*, TFRC's stabilization time and stabilization cost become signifi cant on incipient persistent congestion for large loss event history sizes (N).

3.7 Summary

We summarize our observations as follows:

1. Streaming applications can not always use all of their fair shares. When there is no congestion, streaming applications use less bandwidth than does TCP. However, during congestion period, TFRC applies the same self-clocked rate reduction principle as for bulk data transfer.

2. Without knowing its fair share, TCP continuously probes for available bandwidth by increasing its sending rate until it causes congestion, which in turn causes the performance of other "well-behaving" applications to suffer.

3. Transient background workload increases caused by short-lived flows result in disturbance of streaming applications.

We have also carried out some initial assessments to examine the impact of transient self-clocked rate reduction on perceived video quality. The results show that transient reduction in TFRC's self-clocked rate is sufficient to cause perceived quality degradation for streaming applications. Currently we are incorporating the approaches proposed in [29] to carry out a more comprehensive study on perceived video quality.

4 Media-Aware Rate Control

According to [5], "a congestion control mechanism is *TCP-friendly* if it displays congestion control behavior that, on time scales of several round-trip times (RTTs), obtains roughly the same throughput as a TCP connection in steady-state when the available bandwidth does not change with time". This definition does not dictate how each congestion control mechanism must response to individual packet loss. In [23], TCP-friendliness is categorized into *short term friendliness* and *long term friendliness*. Authors of [19] argues for a TCP-friendly congestion control mechanism that reduces its self-clocked rate less aggressively in the short term, yet maintains its long-term throughput to match that of TCP.

In this study, we focus on *long term TCP-friendliness*. More formally, we use $T_{cc}(t)$ and $T_{tcp}(t)$ to denote the throughput of a generic congestion control mechanism and that of TCP under the same network condition, respectively. The long term *TCP-friendliness* requirement can be expressed as:

$$\int_0^t T_{cc}(x) \leq \int_0^t T_{tcp}(x),\ t \to \infty. \tag{3}$$

With this definition, the short-term throughput of a TCP-friendly congestion control scheme can temporally deviate from TCP's, given that its long term throughput is comparable to that of TCP. This gives some applications, such as streaming media, the flexibility to send more data than their fair share when needed. We leverage such flexibility in our proposed mechanism to prevent increases in transient background workload from disturbing streaming applications, while keeping the streaming applications TCP-friendly.

4.1 Media-aware rate control

When a connection's data rate is less than its fair share, its sending rate is no longer determined by the self-clocked rate. In this case, discussion on the sending rate of a congestion control mechanism should be separated from that of the self-clocked rate. Separating TFRC's sending rate from its self-clocked rate allows TFRC to "remember" how much it underutilizes its fair share. We propose a *media-aware rate control* (MARC) that reduces its sending rate less aggressively during transient congestion while maintaining TCP-friendliness.

MARC takes advantage of mechanisms in TFRC that have been proven to be effective, e.g., WALI and self-clocking. MARC uses Eqn. 1 to calculate its self-clocked rate, but measures the connection's sending rate separately. Whenever its sending rate is less than its calculated allowed-rate, which is an estimation of its fair share, MARC obtains some credit *tokens*. With such tokens, MARC can reduce its self-clocked rate less aggressively during the congestion period, i.e., if MARC has been sending less data than its fair share prior to congestion, it is allowed to decrease its sending rate more slowly during congestion. Tokens should decay with time so that applications running MARC could not abuse the network by accumulating tokens over a prolonged period of time. The use of tokens ensures that in the long term MARC sends no more traffic than a TFRC connection, which has been proved to be TCP-friendly [5]. This can be described more formally as follows:

1. We define *token value* T to count the amount of the "unused" fair share of a connection. Upon receiving a congestion feedback from the receiver, a MARC sender updates T as:

$$T = \beta T' + D, \tag{4}$$

$$D = (W - W_{snd})I, \tag{5}$$

   where W and W_{snd} are the calculated allowed rate and the sending rate during the previous feedback interval, respectively. I is the interval between two receiver feedbacks. D is the "unused" fair share in the current feedback interval. We will discuss in detail the token decay factor, β, in the following section.

2. Upon receiving congestion feedback, MARC uses TFRC's calculation of self-clocked rate as described in [5] to determine TFRC's new self-clocked rate. If this new self-clocked rate is less than $(1 - \delta)$ times MARC's previous self-clocked rate and the token value is positive, i.e., $T > 0$, the following steps are applied to adjust MARC's self-clocked rate:

   (a) if no congestion signal is reported during the last feedback interval, MARC keeps its previous self-clocked rate.

   (b) if congestion signal is reported during the last feedback interval, MARC reduces its self-clocked rate to $(1 - \delta)$ times that of the previous feedback interval.

   Otherwise, MARC adopts the new self-clocked rate.

Since MARC also uses WALI to estimate loss event rate, the effect of the congestion signals will propagate to the feedback intervals following the congested interval. However, we only limit the self-clocked rate reduction during the congested interval. As a result, MARC's self-clocked rate might see sudden reduction in the case that a congested interval is followed by a non-congested interval. Rule 2a is used to prevent this. Furthermore, by adopting TFRC's self-clocking, as we described in Section 3.6, we ensure that MARC would not overshoot the available bandwidth by a factor of larger than two after the congested period. Similar to TFRC, MARC applies the self-clocking mechanism to its calculated allowed-rate. However, instead of directly using the self-clocked rate as TFRC does, MARC compares this self-clocked rate with its rate in previous feedback interval. If its current self-clocked rate decreases from its previous rate by a factor larger than δ, MARC limits its rate reduction from its previous rate if there is available token. Hence it is possible that such limited rate could be larger than TFRC's self-clocked rate when $T > 0$. This will be shown in Section 5. For consistency, we still use the term "self-clocked rate" to denote this limited rate.

The parameter δ is application dependent. Different application can use different δ value to avoid drastic changes in sending rates between two consecutive feedback intervals. Further, δ can be varied depending on the form of the congestion signals, e.g., packet loss or ECN [20]. In our simulation setting, we use $\delta = 0.05$ when the congestion signal is in the form of ECN, and $\delta = 0.1$ for packet losses. As a further optimization, δ can be computed as a function of available tokens.

We note that our modification is applied only when there are available tokens. If no token is available, MARC reverts back to TFRC. This provides MARC with the adaptability for both bulk data transfer and streaming applications.

4.2 Token decay factor

MARC tries to avoid decreasing a connection's self-clocked rate unnecessarily during transient congestion. However, we also need to avoid allowing a connection to send out too much traffic during persistent congestion. This is handled by the decay factor β. Intuitively, with a larger β, MARC can keep a longer history for its previous "good behavior". In turn, this allows MARC to reduce its self-clocked rate over a longer period on incipient congestion. In this section we derive an appropriate value for β both analytically and through simulations.

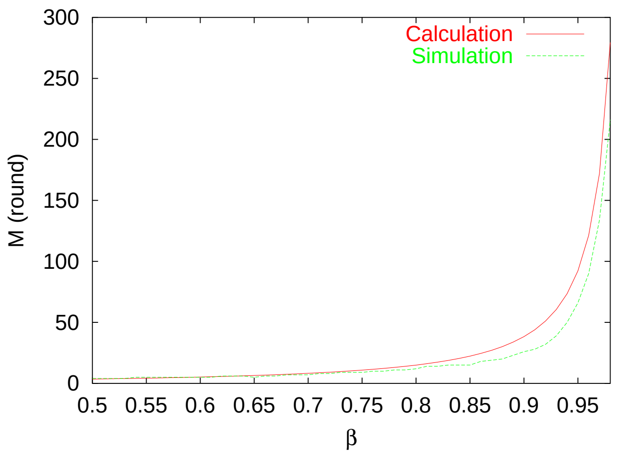


Figure 9: Number of round trip time to exhaust the token.

We assume that the data rate of the streaming application is A. The fair share for MARC is A/a, where a is the utilization of a connection's fair share. We also assume loss event history size $N = 1$ so that MARC's calculated allowed-rate is calculated only based on the most recent loss event rate. Hence its calculated allowed-rate drops below the streaming data rate quickly upon detecting congestion signals. To ensure TCP-friendliness, we only consider $\beta \leq 1$. A simple calculation shows that the token value during the nth report interval is:

$$T_n = \frac{(A/a - A)I(1 - \beta^n)}{1 - \beta}. \tag{6}$$

After a suffi ciently long period, we have:

$$T = \lim_{n \to \infty} T_n = \frac{(A/a - A)I}{1 - \beta}. \tag{7}$$

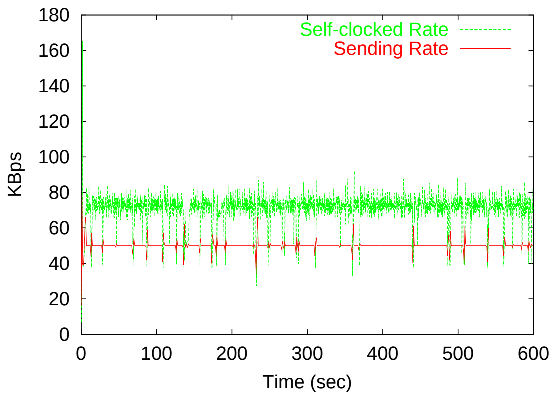
On the onset of persistent congestion, MARC decreases its self-clocked rate less aggressively until it has exhausted all of its available tokens. We can use the number of report intervals MARC takes to consume all of its tokens as a metric to evaluate its aggressiveness. We denote this number M. We need M such that:

$$T * \beta^M < AI, \tag{8}$$

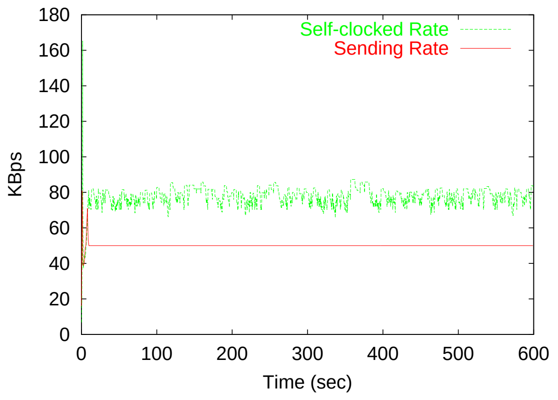
$$\frac{(A/a - A)I}{1 - \beta} * \beta^M < AI, \tag{9}$$

$$\frac{(1/a - 1)\beta^M}{1 - \beta} < 1. \tag{10}$$

Eqn. 10 indicates the relationship between β, a, and M. We evaluate Eqn. 10 against simulation results. Fig. 9 plots M versus β from both Eqn. 10 and the simulation results. We introduce 0.1% periodic packet loss at the beginning of our simulation. We increase the loss rate to 10% at the 800th second. Given this setting, we have $a = 0.15$. For different β values, we calculate the number of rounds needed for MARC to decrease its token to less than $A * I$. Fig. 9 shows that Eqn. 10 indeed characterizes the correlation between β and M. When β is relatively large, MARC sends out more packets during persistent congestion, and larger queueing delay is introduced. Since queueing delay is not considered in Eqn. 10, the simulation result shows that when smaller β is used, MARC exhausts its tokens more quickly, which implies a quicker response on persistent congestion. When β increases beyond 0.9, there is signifi cant increase in the response time. We decide to use $\beta = 0.9$ in our simulations.

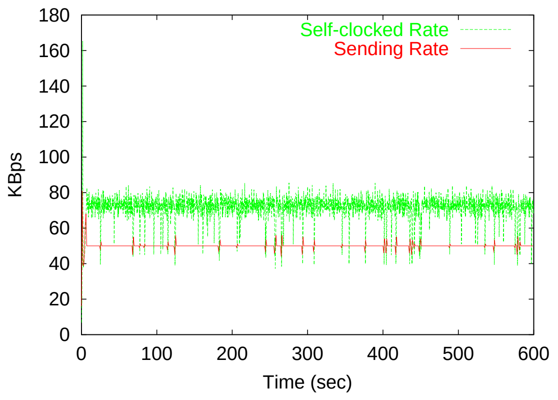


(a) TFRC

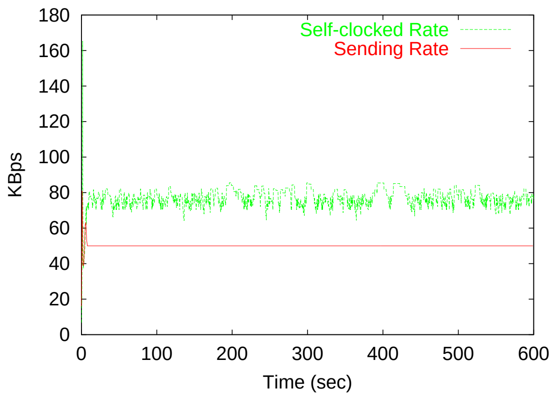


(b) *MARC*

Figure 10: Self-clocked rates and sending rates when competing with single long-lived TCP. The streaming data rate is 50 KBps, N=8.

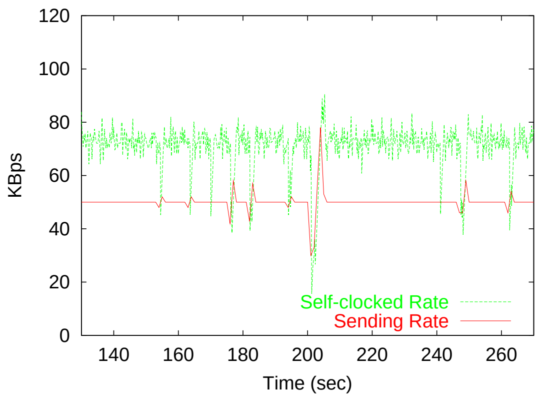


(a) TFRC

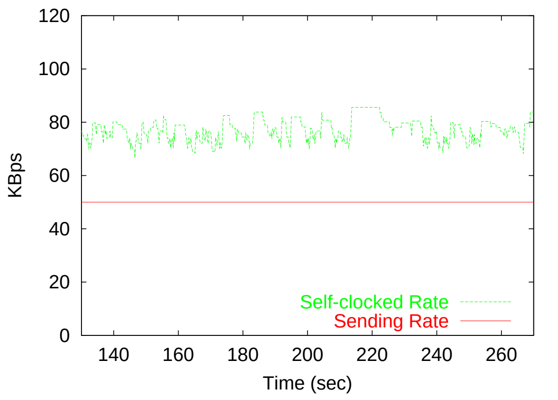


(b) *MARC*

Figure 11: Self-clocked rates and sending rates when competing with single long-lived TCP. The streaming data rate is 50 KBps. N=128

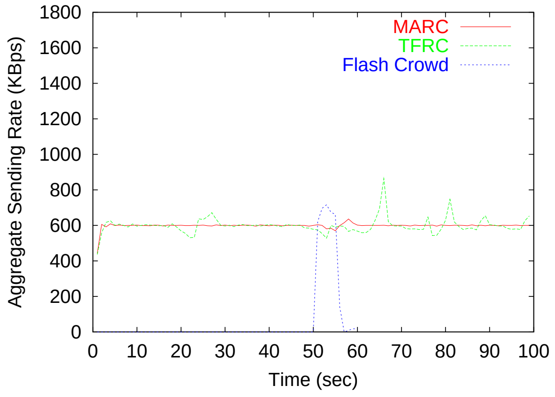


(a) TFRC

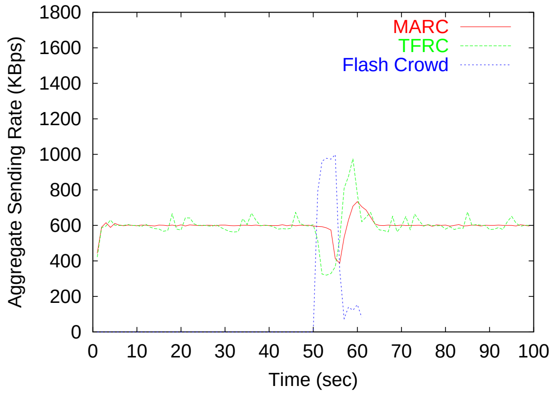


(b) *MARC*

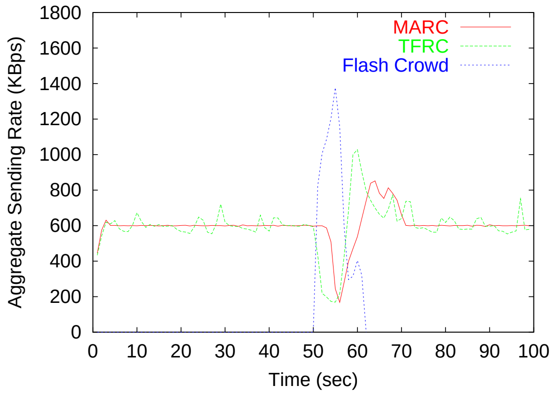
Figure 12: Self-clocked rate and sending rate when a short-live TCP flow exists.



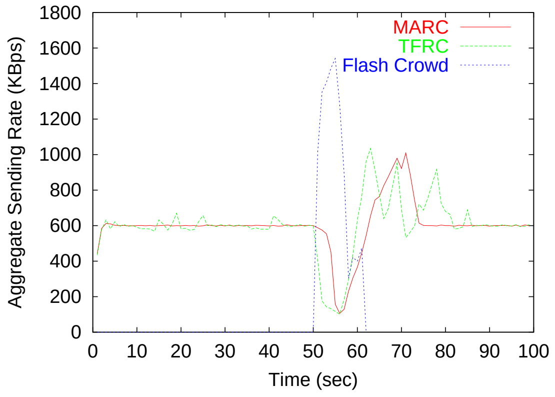
(a) 400 TCP flows



(b) 600 TCP flows

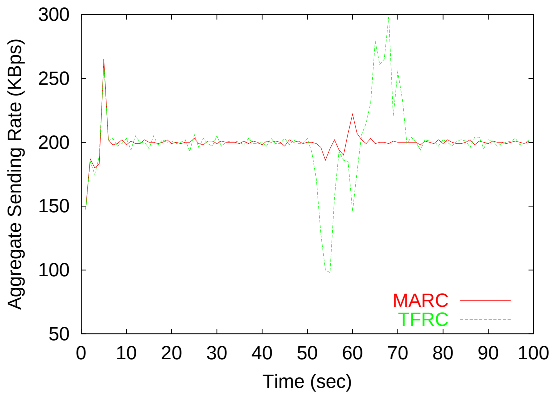


(c) 800 TCP flows

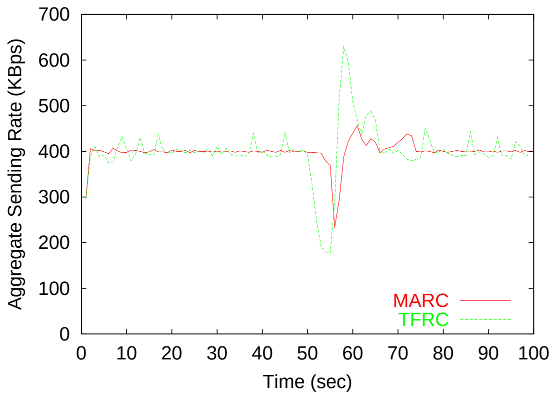


(d) 1,000 TCP flows

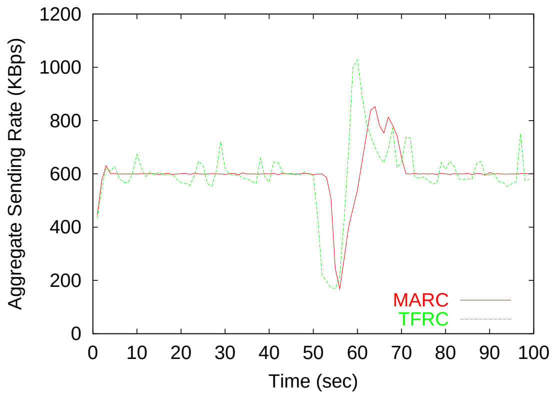
Figure 13: Aggregate sending rate of 20 tested flows when N short-lived TCP flows are present. The data rate for each tested flow is 30 KBps.



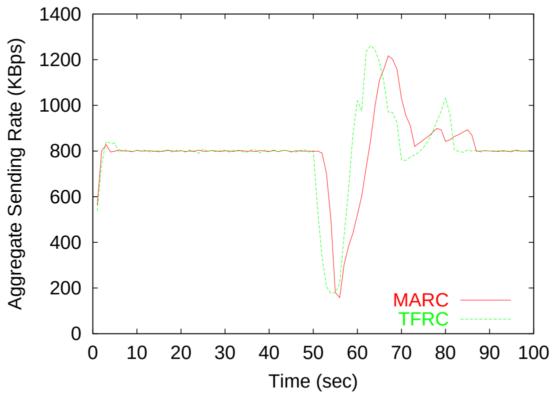
(a) Data rate = 10 KBps



(b) Data rate = 20 KBps



(c) Data rate = 30 KBps



(d) Data rate = 40 KBps

Figure 14: Aggregate sending rate of 20 tested flows with different data rate when 800 short-lived TCP flows are present.

5 Evaluation

5.1 Basic Behavior

The rest of this paper is dedicated to the evaluation of MARC's performance under different network condition. Our evaluations use the *ns-2* simulator, where we have implemented MARC as an extension to TFRC. [2]

Figs. 10 and 11 compare MARC and TFRC's self-clocked rates and sending rates with different loss history sizes N. The streaming data rate is 50 KBps and the fair share for both connections is 94 KBps. A TCP connection competes with MARC for the bottleneck link bandwidth. While TFRC's self-clocked rate drops below the streaming data rate from time to time, MARC maintains a more stable self-clocked rate. When we increase the loss history size N from 8 to 128, TFRC still fails to prevent transient rate decreases. In contrast, MARC's self-clocked rate smoothness depends less on its loss history size. In these fi gures, while TFRC's self-clocked rate is smaller than its fair share due to the self-clocking mechanism, MARC's self-clocked rate is larger than its fair share. This is due to the rate reduction limiting mechanism described in Section 4.1 by which MARC limits its self-clocked rate reduction to no more than δ of its previous self-clocked rate. Hence the new self-clocked rate could possibly be larger than MARC's calculated allowed-rate, which is the estimation of the current fair share. However, MARC's use of tokens ensures that, when the streaming data increases, MARC's sending rate will not exceed that of TCP's. We will discuss MARC's TCP-friendliness further in Section 5.2.

[2]The ns-2 codes for MARC are available at http://topology.eecs.umich.edu/codes/marc/ns-src/

In the stable state, the congestion control mechanism of a long-lived TCP connection will be in the congestion avoidance phase, probing for available bandwidth by increasing self-clocked rate one packet per round-trip time. This prevents the connection from drastically changing network load. A short-lived TCP connection, however, may never reach this stable state. Instead, its congestion control mechanism may not go beyond the slow-start phase. During slow-start, TCP doubles its sending rate per round-trip time until it sees the first packet loss. The exponentially increased self-clocked rate, on one hand, enables TCP to adapt to available bandwidth more quickly, but, on the other hand, it also causes significant workload increases seen by the network in a short period.

To examine the impact of competing short-lived TCP flows, we introduce a short TCP connection in our simulations. The short-lived TCP flow starts from the 200th second and lasts for 5 seconds. We plot the self-clocked rate and the sending rate for TFRC and MARC under this scenario in Fig. 12. Fig. 12(a) shows that the transient workload increase imposed by a single short-lived TCP connection is sufficient to cause TFRC to reduce its self-clocked rate below the streaming data rate. Meanwhile, the slight fluctuation in MARC's self-clocked rate is not significant enough to affect the connection's sending rate, as shown in Fig. 12(b).

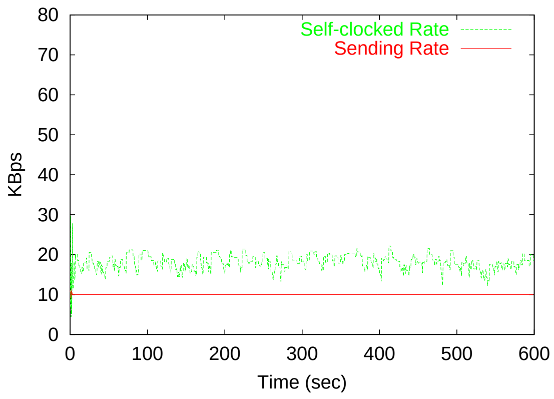
We then look at the case when multiple concurrent short-lived TCP flows are present. We use 20 tested flows, either running TFRC or MARC, to compete with a flash crowd, which constitutes a number of short-lived TCP flows. The number of the short-lived flows varies from 400 to 1,000, to simulate different magnitudes of transient workload. These short-lived flows are evenly distributed within a 5 second period starting at the 50th second. Other than the tested flows and the flash crowd, a long-lived TCP connection with a greedy source is also used to consume all the available bandwidth. The bottleneck bandwidth in this simulation is set to 10 Mbps and the streaming data rate is set to 30 KBps. The aggregate streaming data rate of the 20 tested flows is hence about 50% of the bottleneck bandwidth. These settings are taken from [6], except that we use streaming applications instead of bulk data transfer as the traffic source.

The aggregate sending rate of the tested flows and the flash crowd are shown in Fig. 13. First notice that without the interference of the flash crowd, MARC has a slightly smoother aggregate sending rate than TFRC. Further, when only 400 TCP flows are introduced (Fig. 13(a)), both TFRC and MARC manage to maintain stable sending rates during the presence of the flash crowd. With larger flash crowd, TFRC reduces its sending rate quickly while MARC reacts less aggressively before its token is exhausted. The larger the size of the flash crowd, the more similar to TFRC MARC behaves because its tokens are exhausted more quickly.

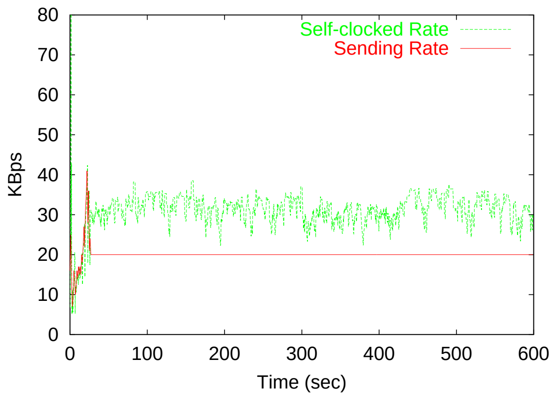
Fig. 14 illustrates another important property of MARC. In this figure, we show the aggregate sending rate of MARC and TFRC while varying the streaming data rate. For each figure, the flash crowd includes 800 short-lived TCP connections. Fig. 14(a) indicates that when the streaming data rate is small relative to a connection's fair share, 10 KBps out of 60 KBps in this case, MARC effectively maintains its sending rate while TFRC is heavily affected by the flash crowd, and must reduce its sending rate by more than a half. Should the applications increase their data rates, MARC becomes less effective in protecting the connections from transient increases in background workload, since less tokens can be accumulated prior to the congestion period. This property provides streaming applications with incentives to decrease their data rate to receive better tolerance towards transient network dynamics.

5.2 Fairness

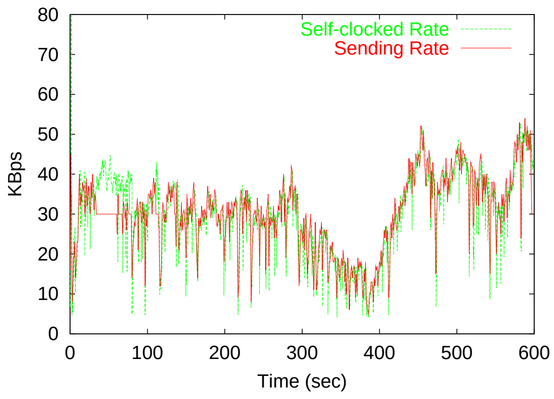
When it has tokens available, MARC reduces its self-clocked rate slower than does TFRC. Consequently, it could have a larger self-clocked rate than TFRC's during the onset of congestion. The use of tokens ensures that this is possible only when the sender has been sending less than its fair share prior to the onset of congestion. When there is no token available, MARC behaves exactly the same as TFRC and is not more aggressive than TFRC. We illustrate this in Fig. 15. The graphs show MARC's self-clocked rate and sending rate when it competes with five ON/OFF flows and one TCP connection. The fair share for MARC is 40 KBps. We vary the streaming data rate from 10 KBps to 40 KBps in the simulations to verify that MARC would not send more data than its fair share when the streaming application increases its data rate.



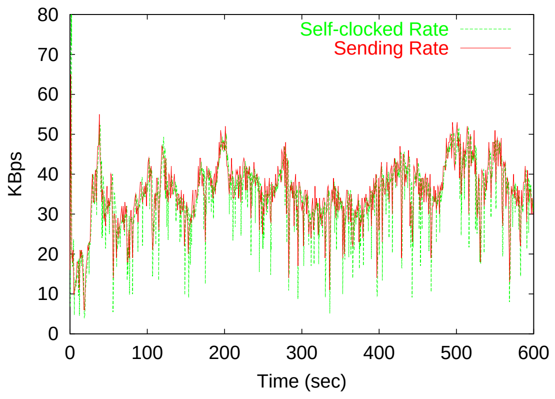
(a) Data rate = 10 KBps



(b) Data rate = 20 KBps

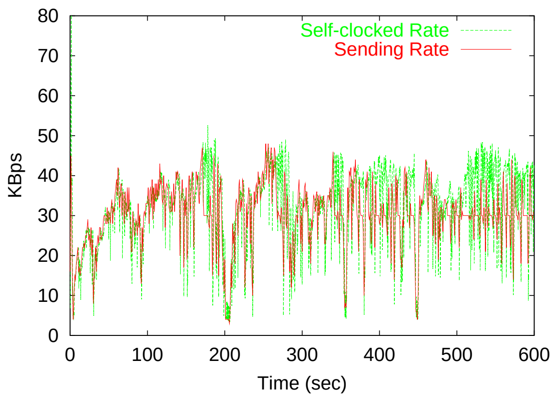


(c) Data rate = 30 KBps

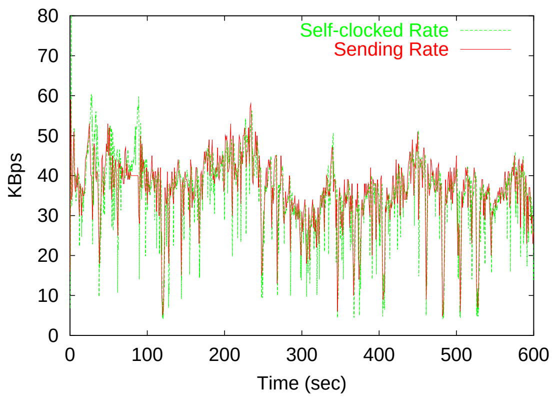


(d) Data rate = 40 KBps

Figure 15: MARC's self-clocked rate and sending rate when the streaming data rate increases. The fair share is 40 KBps.



(a) Data rate = 30 KBps



(b) Data rate = 40 KBps

Figure 16: TFRC's self-clocked rate and sending rate when the data rate approaches the fair share.

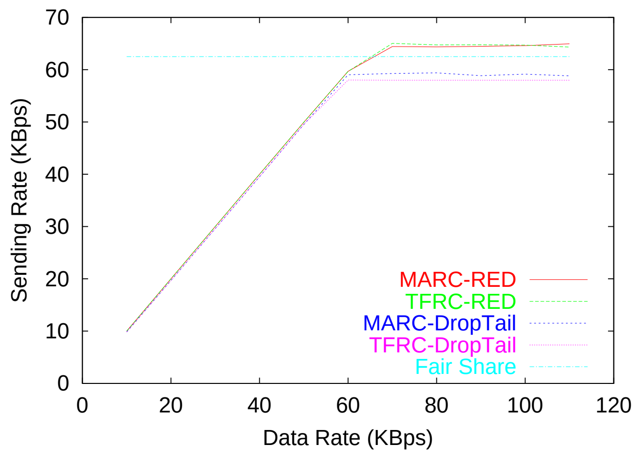


Figure 17: Comparison between MARC and TFRC's sending rate when streaming data rate increases. The fair share is 62.5 KBps.

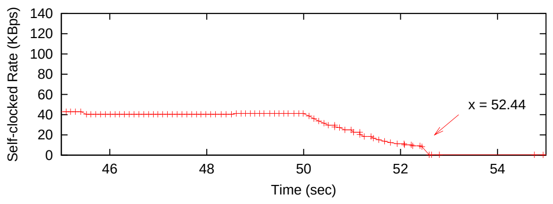
When the application data rate is small compared to its fair share, Figs. 15(a) and (b) show that MARC manages to maintain an ideally smooth sending rate. When the streaming application's data rate approaches fair share, MARC acquires less token during each report interval and as a result, can not guarantee a smooth sending rate, as shown in Fig. 15(c). If the application further increases its data rate, MARC would not be able to accumulate any token and would behave similarly to TFRC. For comparison, we plot the results for TFRC in Fig. 16. In Fig. 15, MARC's sending rates could increase beyond its data rate due to source buffering during congestion. We assume sources have infi nite send buffer.

Since TFRC has been shown to be TCP-friendly [5, 6], to show that MARC is TCP-friendly, we only show that MARC produces similar sending rate to TFRC. Fig. 17 illustrates the average sending rate of 10 tested flows with 10 greedy TCP flows competing for 10 Mbps bottleneck bandwidth. We ran two simulations. One uses MARC as the tested flow and the other uses TFRC. The streaming data rate varies from 10 KBps to 110 KBps. In addition to RED, we also use drop-tail scheme in this simulation. MARC's average sending rate is no more than that of TFRC under both RED and drop-tail gateways.

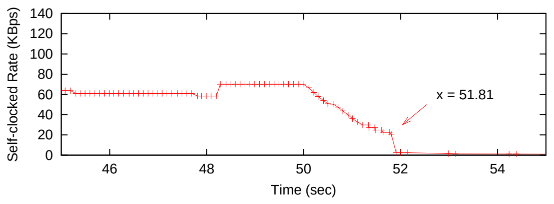
Another observation from Fig. 17 is that both MARC and TFRC achieve higher sending rate with RED. Considering that the fair share of each connection in our simulation is 62.5 KBps, the observation indicates that TFRC and MARC acquire higher throughput than competing TCP connections when RED is used. In [30], the authors have found that, by dropping packets from different flows with the same probability, RED is likely to give higher throughput to less responsive flows. Similar observation has also been made in [31], which shows that RED is not always able to maintain fairness between flows with different characteristics. We further discuss how slowly-responsive congestion control mechanisms such as TFRC and MARC affect bandwidth distribution across different flows in Section 5.4.

5.3 Reducing Sending Rate on Persistent Congestion

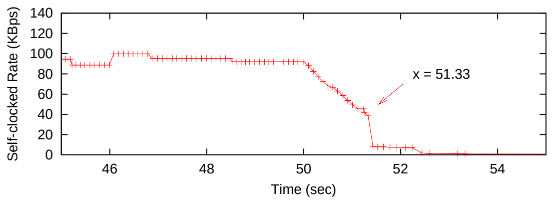
A congestion control mechanism should respond to persistent congestion to avoid congestion collapse [32]. TCP always halves its self-clocked rate upon detecting packet loss. TFRC takes at least $4{*}RTT$ to halves its calculated allowed-rate on persistent congestion [6]. For MARC, when the self-clocked rate is less than $1 - \delta$ times the previous self-clocked rate, instead of adopting the new self-clocked rate, it decreases its current self-clocked rate by a multiple of δ if there are tokens available. In this section we examine the time MARC takes to halves its self-clocked rate.



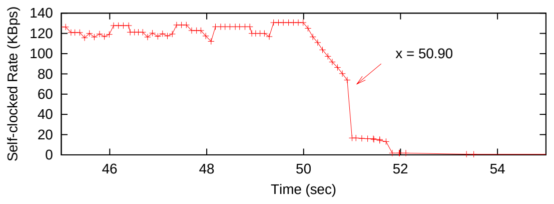
(a) Data rate = 20 KBps



(b) Data rate = 40 KBps



(c) Data rate = 60 KBps



(d) Data rate = 80 KBps

Figure 18: Self-clocked rate for MARC on persistent congestion.

In our simulation, the path RTT is 80 ms. The bottleneck bandwidth is 10 Mbps. RED is deployed on the bottleneck link. The loss history size (N) for both TFRC and MARC is set to 8. This setting is taken from [5], except here we use MARC and CBR sources, with data rates from 20 KBps to 80 KBps. Fig. 18 plots MARC's self-clocked rate while the packet loss rate is changed from 1% to 50% at the 50th second. The figure shows that before the token is exhausted, MARC's self-clocked rate reduction rate, which is determined by δ, is correlated to the streaming data rate. As described in Section 4.1, the reduction rate is determined by δ and the previous self-clocked rate. Further, with self-clocking, the self-clocked rate is proportional to the data rate. As a result, the larger the data rate is, the faster MARC reduces its self-clocked rate. This is illustrated in Figs. 18(a), (b), (c) and (d). When the token is exhausted, MARC reverts back to TFRC and adopts TFRC's self-clocked rate. Since MARC has been reducing its self-clocked rate more slowly than TFRC, there would be an abrupt decrease in MARC's self-clocked rate. We point out such abrupt changes in Fig. 18 with arrows, along with the time when MARC runs out of its token. When larger data rate is used, it takes less time for MARC to exhaust its token and reduce its self-clocked rate. For a given amount of fair share, less token could be accumulated when a larger data rate is used; as a result, MARC runs out of tokens more quickly.

The correlation between self-clocked rate reduction rate and the fair share utilization of the streaming application can be better illustrated with Fig. 19. In this figure, we show the number of round-trip times for MARC to halve its self-clocked rate while the fair share utilization increases from 2% to 2000%. The initial packet loss rate is set to 0.1%. We also show how long it takes TFRC, with and without self-clocking, to halve its self-clocked rate. To illustrate the effect of increasing loss event history size (N), we set N to 128. With these settings, the fair share for the tested flows before the loss rate changes is 547 KBps. When the fair share utilization is small, both TFRC schemes are very responsive to increases in loss rate. We have seen in previous sections that such responsiveness could introduce unnecessary self-clocked rate fluctuation in TFRC. When the fair share utilization increases, self-clocking helps TFRC retain its responsiveness; without self-clocking, TFRC takes a significantly longer period to halve its self-clocked rate when data rate is high. Similar conclusion for bulk data transfer has been reported in [6]. MARC, on the other hand, allows streaming applications with small fair share utilization ($< 50\%$) to reduce their sending rate more slowly on transient congestion while retaining its long term TCP-friendliness.

We next vary initial packet loss rates from 1% to 25% and examine the number of RTTs needed by MARC to halve its sending rate. The streaming data rate is 20 KBps. The result shown in Fig. 20 indicates that it takes around $12 * RTT$ for MARC to halve its sending rate when the initial packet loss rate is small. According to

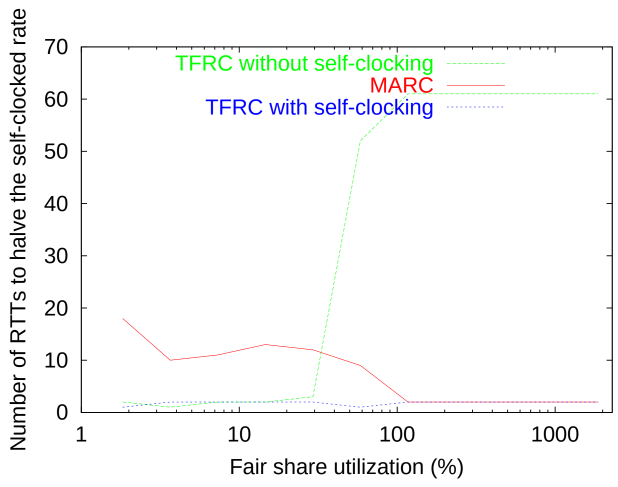


Figure 19: Number of RTTs needed to halve self-clocked rate with varied fair share utilization. The initial loss rate is 0.1%.

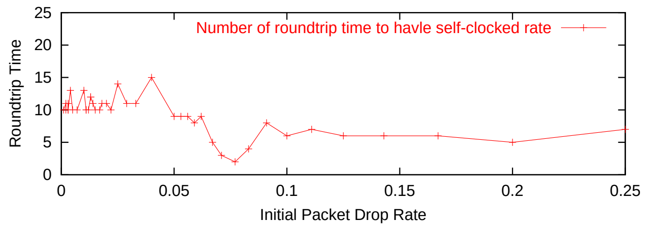


Figure 20: Number of RTT needed for MARC to halve its self-clocked rate. Data rate is 20 KBps.

Eqn. 1, the higher the initial packet loss rate, the less MARC's fair share is. When the initial packet loss rate is so large that MARC's fair share is less than its data rate, no token could be accumulated before the onset of persistent congestion. MARC takes about 4-8 RTTs to halve its self-clocked rate. Again, in this case MARC's performance is similar to that of TFRC. This is shown in Fig. 20 when large initial packet loss rate is present.

5.4 Transient behavior during congestion

To be TCP-friendly, the sending rate of a slowly-responsive congestion control mechanism should be comparable to that of TCP during the steady state. To maintain such compatibility, both MARC and TFRC reduce their self-clocked rate less aggressively when congestion occurs and also increase their self-clocked rate more slowly when congestion disappears. While in the long term such behavior results in a TCP-friendly sending rate, it does not guarantee TCP-friendliness in the short term. By allowing more data to be sent on incipient congestion, it is possible for MARC and TFRC to interfere with the performance of other long-lived connections. In this section we focus on MARC's transient behavior during congestion period.

We use flash crowd as described in Section 5.1 to introduce congestion in our simulations. Other than the flash

Table 1: TFRC's behavior on congestion for bulk data transfer.

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
TFRC	16.17±1.0 (5.11)	13.02±0.92 (4.72)	22.01±1.27 (6.5)
TCP	11.74±1.37 (6.97)	10.55±1.31 (6.7)	17.65±3.2 (16.31)
UDP	104.21±6.03 (30.79)	75.82±3.93 (20.07)	26.39±0.99 (5.06)
Flash	117.41±2.14 (10.93)	86.11±2.71 (13.82)	25.48±1.13 (5.75)

Table 2: MARC's behavior during congestion for bulk data transfer.

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
MARC	21.23±0.91 (4.67)	16.24±0.84 (4.31)	24.52±1.13 (5.77)
TCP	13.12±1.53 (7.79)	11.99±1.5 (7.64)	15.62±2.64 (13.46)
UDP	108.01±6.42 (32.73)	76.84±4.23 (21.6)	27.83±0.93 (4.76)
Flash	113.13±2.37 (12.08)	80.67±2.73 (13.91)	27.62±1.07 (5.47)

Table 3: TFRC's behavior during the entire simulation for bulk data transfer.

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
TFRC	29.66±1.02 (5.2)	28.11±1.01 (5.18)	5.33±0.23 (1.17)
TCP	36.27±1.4 (7.13)	35.11±1.39 (7.11)	3.27±0.19 (0.95)
UDP	107.06±2.94 (15.02)	94.94±2.41 (12.29)	10.18±0.32 (1.64)
Flash	11.18±0.1 (0.49)	9.08±0.01 (0.05)	18.62±0.71 (3.63)

Table 4: MARC's behavior during the entire simulation for bulk data transfer.

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
MARC	31.38±1.05 (5.35)	29.65±1.06 (5.4)	5.65±0.26 (1.3)
TCP	36.11±1.43 (7.29)	34.97±1.43 (7.3)	3.29±0.19 (0.99)
UDP	106.13±3.18 (16.22)	94.88±2.62 (13.36)	10.38±0.32 (1.64)
Flash	11.35±0.11 (0.54)	9.08±0.02 (0.08)	19.8±0.76 (3.88)

Table 5: TFRC's behavior on congestion for streaming applications (data rate: 20 KBps).

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
TFRC	10.21±0.84 (4.29)	8.01±0.77 (3.92)	24.69±1.72 (8.79)
TCP	15.57±1.83 (9.32)	14.54±1.81 (9.21)	13.36±1.8 (9.17)
UDP	102.44±6.6 (33.67)	74.95±4.5 (22.94)	25.94±0.98 (4.99)
Flash	118.4±2.38 (12.14)	87.83±2.97 (15.16)	24.83±1.24 (6.34)

Table 6: MARC's behavior on congestion for streaming applications (data rate: 20 KBps).

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
MARC	18.67±0.43 (2.18)	14±0.44 (2.25)	25.07±1.21 (6.16)
TCP	15.2±1.56 (7.95)	14.07±1.53 (7.8)	12.79±1.83 (9.33)
UDP	104.54±6.77 (34.53)	74.28±4.19 (21.39)	27.6±1.03 (5.28)
Flash	115.35±2.15 (10.95)	83.22±2.85 (14.54)	26.84±1.32 (6.74)

Table 7: TFRC's behavior during the entire simulation for streaming applications (data rate: 20 KBps).

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
TFRC	19.53±0.2 (1.02)	18.58±0.22 (1.12)	4.85±0.22 (1.11)
TCP	46.33±1.79 (9.15)	45.04±1.8 (9.17)	2.9±0.2 (1.04)
UDP	103.21±2.75 (14.03)	93.59±2.25 (11.45)	9.08±0.33 (1.68)
Flash	11.12±0.11 (0.57)	9.08±0.01 (0.05)	18.1±0.81 (4.12)

Table 8: MARC's behavior during the entire simulation for streaming applications (data rate: 20 KBps).

Flow Type	sending rate (KBps)	throughput (KBps)	loss rate (%)
MARC	19.81±0.12 (0.63)	18.64±0.16 (0.78)	5.91±0.27 (1.36)
TCP	45.5±1.86 (9.49)	44.26±1.87 (9.53)	2.82±0.18 (0.93)
UDP	105.05±2.88 (14.69)	93.95±2.28 (11.63)	9.33±0.39 (1.97)
Flash	11.29±0.12 (0.59)	9.08±0.01 (0.04)	19.33±0.8 (4.09)

crowd, a long-lived TCP connection and five Pareto ON-OFF traffic flows are also competing for the bottleneck bandwidth. The flash crowd consists of 100 short-lived TCP flows. We repeat each simulation 100 times. Each simulation lasts for 100 seconds and the flash crowd starts at the 50th second and lasts for 5 seconds.

We first focus on the 5-second period when the flash crowd exists. Table 1 lists the average value with 95% confidence interval of the sending rate, throughput, and packet loss rate for each flow type. The standard deviation is also shown in parentheses. Comparing Tables 1 and 2 we see that when used with greedy traffic sources, MARC has higher sending rate than TFRC during congestion period. During the congestion period, self-clocking limits the sending rate to be no more than the throughput in the previous cycle. This self-clocked rate could be smaller than the sending rate calculated with Eqn. 1. These tables also show that the long-lived TCP connection has a much lower loss rate than other flows, further confirming the observation made in [23, 30, 31] that less responsive flows would experience higher loss rate on a RED gateway. We also notice that, even though MARC has higher sending rate and throughput than TFRC, it does not come at the cost of the sending rate and throughput of responsive connections such as TCP. Instead, the gain in MARC's throughput comes at the expense of irresponsive flows such as the UDP flows and the flash crowd.

We show the results for the entire simulation in Tables 3 and 4. These results indicate that in the long term MARC sends no more data than TCP. In the long term, both MARC and TFRC have smaller standard deviations in their sending rate and throughput than those of TCP.

To examine the scenario when streaming media is used, we replace the data source in the simulation above with CBR source. The data rate used is 20 KBps. The fair share is 42 KBps. The results are shown in Tables 5, 6, 7, and 8. From these tables, we conclude that MARC can deliver larger throughput than TFRC while maintaining smaller standard deviation in sending rate.

5.5 Performance with ON-OFF flows as background traffic

We now evaluate MARC's smoothness under long-range dependent background traffic. We use 40 ON-OFF Pareto flows and 4 long-lived TCP flows to compete with a single MARC or TFRC connection. The bottleneck bandwidth is 10 Mbps, giving each connection a fair share of 83.3 KBps. The data rate for MARC and TFRC varies from 10 to 120 KBps. We repeat each simulation 20 times. Fig. 21 shows the average, with 95% confidence interval, coefficient of variation in sending rate between MARC and TFRC, for various *fair share utilization*. A connection's fair share utilization is the ratio between its sending rate and its fair share.

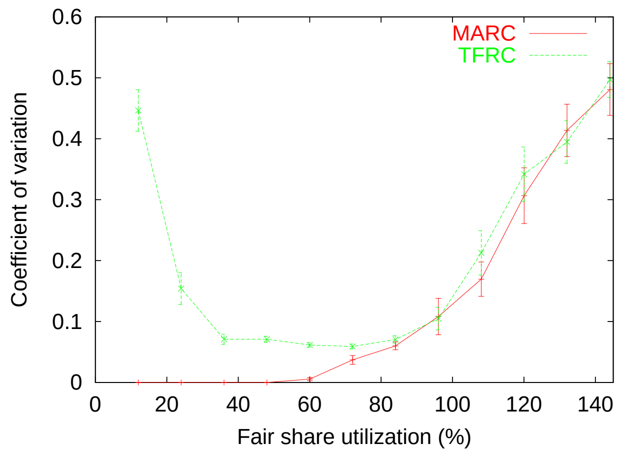
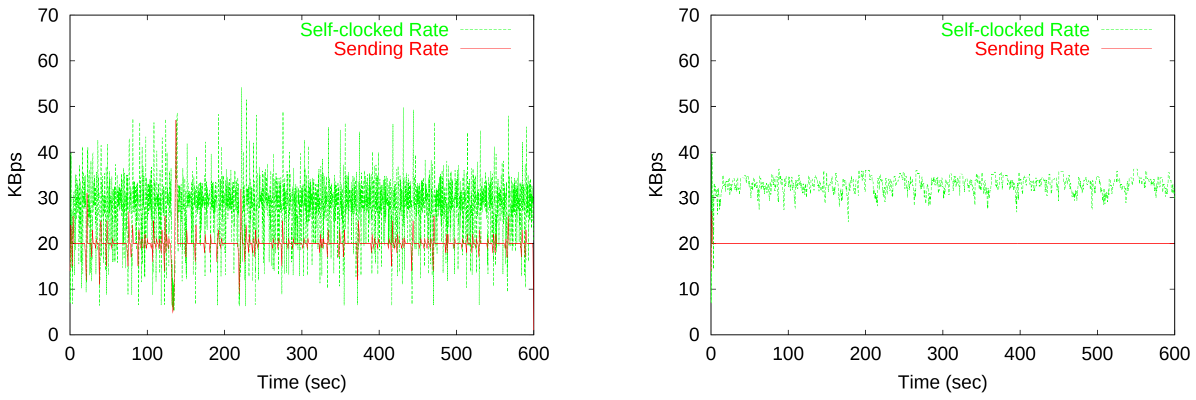


Figure 21: Average coefficient of variation in sending rate.



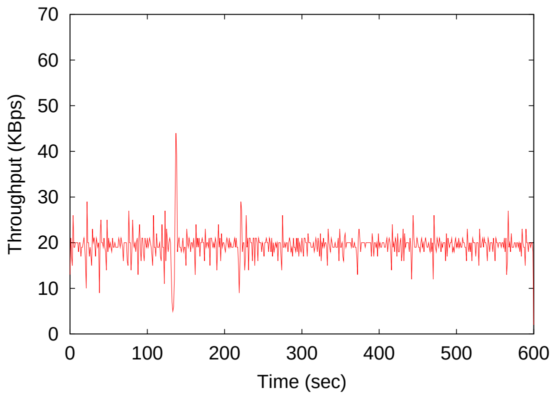
(a) TFRC (b) *MARC*

Figure 22: Self-clocked rate and sending rate variation for TFRC and *MARC*. Data rate is 20 KBps and the fair share is 42 KBps.

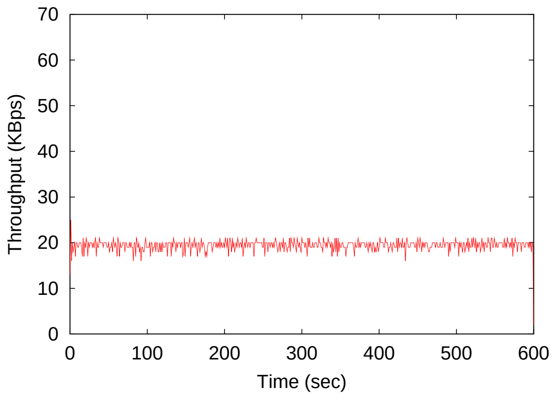
Fig. 21 shows that TFRC has a relatively large coefficient of variation in its sending rate when its fair share utilization is small. This is because TFRC is not media-aware. With bursty background traffic, it suffers from sending rate fluctuation regardless its fair share utilization. In contrast, MARC "remembers" that it has been sending less data than its fair share and decreases its sending rate more slowly on receiving congestion signals. This provides a smoother sending rate profile for streaming media when the data rate is small. With the increase in fair share utilization, the performance difference between MARC and TFRC decreases. To illustrate, we show in Fig. 22 a sample path of the self-clocked rate and the sending rate of TFRC and MARC during one run of our simulations. The data rate is 20 KBps in both cases. We observe fluctuations in TFRC's self-clocked rate due to the bursty background ON-OFF traffic. MARC, as shown in Fig. 22, can support a more stable sending rate.

5.6 Receiver Performance Metrics

In previous studies, performance evaluations of a congestion control mechanism have focused on sender- or network-centric metrics such as sending rate and fairness. For end users, however, metrics which are able to indicate the performance advantages of a mechanism from the user's point of view is more useful. In this section we evaluate MARC's performance in terms of application level performance metrics, in particular, receiver-centric performance metrics.



(a) TFRC



(b) MARC

Figure 23: Receiver perceived throughput corresponding to Fig. 22.

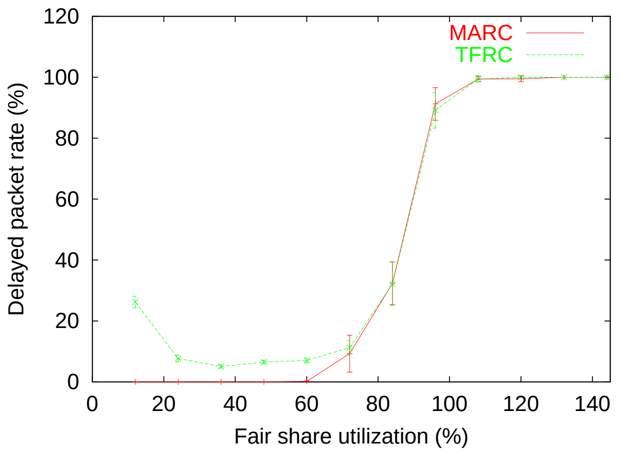


Figure 24: Percentage of delayed packet.

The first receiver-centric metric we look at is the *throughput* of a connection. Fig. 23 shows the throughput for the sending rate seen in Fig. 22. Comparing these two figures, we can see that the sending rate fluctuations under TFRC directly result in throughput fluctuations at the receiver. We next measure the percentage of delayed packets when MARC is used for streaming applications. The authors of [4] found that human detection for synchronization between audio and video is around 100 ms, i.e., if a video packet arrives at the receiver 100 ms after the corresponding audio packet, the user would perceive worse quality. Accordingly, we define an incoming packet as *delayed* if it arrives 100 ms or more after its scheduled playback time at the receiver. This means we must define a playback time for each incoming packet. Applications can reduce the negative impact of delayed packets to some degree by buffering before playback. To simplify our evaluation, we define *smart clients*. A smart client knows *a priori* the *maximum queueing delay* experienced by all packets of a connection and can use it to size its playback buffer accordingly. In reality, without knowledge of queueing delay, a client simply buffers for a pre-defined amount of time. Our definition of smart clients ensures that delayed packets seen by a smart client is solely caused by the congestion control mechanism. We repeat the simulation in Section 5.5 10 times and calculate the average delayed packet rate. The result is shown in Fig. 24 along with the 95% confidence interval. The figure indicates that MARC reduces delayed packet rate when its fair share utilization $< 70\%$.

Another receiver-centric metric we examine is *jitter*. In [1], the authors define jitter as the standard deviation of the inter-packet arrival time measured at the receiver. Their survey shows that jitter is closely correlated to perceived video quality. We measured the jitter in the above simulation and plot the result in Fig. 25. The figure

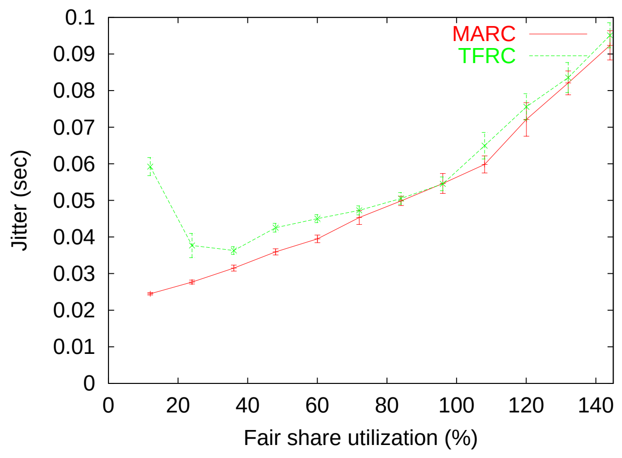


Figure 25: Jitter

shows that when the fair share utilization of a streaming application is $< 70\%$, MARC could reduce up to 60% of the jitter compared to TFRC. However, the performance difference between TFRC and MARC shrinks quickly with the increase in fair share utilization.

6 Summary and Conclusions

In this paper we evaluate TFRC's ability to provide streaming applications with smooth sending rate. In time scales larger than path RTT, TFRC fails to prevent abrupt sending rate reduction during transient workload increases. Increasing loss event history size or removing self-clocking in TFRC can only give TFRC slight resistance to such transient changes. We propose MARC, a TCP-friendly, media-aware congestion control mechanism for streaming media transfer. When application data rate is less than its fair share, MARC provides streaming applications with additional protection from transient congestion by reducing its self-clocked rate less aggressively than does TFRC. For bulk data transfer, MARC's behavior is similar to that of TFRC. We have carried out a large number of simulations to evaluate MARC's performance. We show that MARC is capable of tolerating bursty network workload while maintaining *TCP-friendliness*. Previous slowly-responsive congestion control mechanisms including TFRC has excluded some important traffi c characteristics possessed by streaming media from their studies. We argue that one should take into consideration the characteristics of streaming applications when developing congestion control mechanisms for them. This allows a congestion control mechanism to be *media-friendly* as well as *TCP-friendly*. MARC is a fi rst step towards a media-aware congestion control mechanism for streaming applications.

7 Future Work

Variable bit rate (VBR) coding is likely to be widely deployed in the Internet for media content. One direction of our ongoing work is to accommodate VBR encoded streaming media. A main difference between CBR coding and VBR coding is that VBR encoded media content requires peak sending rate that can be much higher than its average rate. With the use of such coding schemes, using CBR to approximate media content is no longer valid. Correspondingly, a constant sending rate may no longer be a meaningful goal for a congestion control mechanism designed for such applications. The credit accumulation scheme of MARC may also become less effective if the trough between data peaks in the VBR stream is relatively short. We will look into adaptive coding mechanisms that can preferentially change the codes used depending on the importance of the scenes depicted. Such coding mechanism could be asked to reduce its data rate when MARC is running short of tokens, for example.

References

[1] M. Claypool Y. Wang and Z. Zuo, "An empirical study of realvideo performance accross the internet," *Proceedings of the ACM SIGCOMM Internet Measurement Workshop'01*, 2001.

[2] S. Banerjee, J. Brassil, and et. al. A. C. Dalal, "Rich media from the masses," Tech. Rep. HPL-2002-63, Hewlett-Packard Laboratories, Palo Alto, CA, 2002.

[3] D. Wu, Y. Hou, W. Zhu, Y. Zhang, and J. Peha, "Streaming video over the internet: Approaches and directions," *IEEE Transactions on Circuits and Systems for Video Technology'01*, vol. 11, no. 1, pp. 1–20, 2001.

[4] R. Steinmetz, "Human perception of jitter and media synchronization," *IEEE Journal of Selected Areas in Communication'96*, vol. 14, no. 1, January 1996.

[5] Sally Floyd, Mark Handley, Jitendra Padhye, and Jorg Widmer, "Equation-based congestion control for unicast applications," *Proc. of ACM SIGCOMM '00*, pp. 43–56, Aug. 2000.

[6] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker, "Dynamic behavior of slowly-responsive congestion control algorithms," *Proc. of ACM SIGCOMM '01*, Aug. 2001.

[7] Deepak Bansal and Hari Balakrishnan, "Binomial congestion control algorithms," *Proc. of IEEE INFOCOM '01*, pp. 631–640, 2001.

[8] Reza Rejaie, Mark Handley, and Deborah Estrin, "RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet," *Proc. of IEEE INFOCOM '99*, pp. 1337–1345, 1999.

[9] V. Jacobson and M. J. Karels, "Congestion Avoidance and Control," *Proc. of ACM SIGCOMM '88*, vol. 18, 4, pp. 314–329, 1988.

[10] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for TCP," *Proc. of ACM SIGCOMM '96*, vol. 26,4, pp. 270–280, 1996.

[11] L.S. Brakmo and L.L. Peterson, "TCP vegas: End to end congestion avoidance on a global internet," *IEEE Journal of Selected Areas in Communication*, vol. 13, no. 8, pp. 1465–1480, Oct. 1995.

[12] Kevin Fall and Sally Floyd, "Simulation-based comparisons of Tahoe, Reno and SACK TCP," *Computer Communication Review*, vol. 26, no. 3, pp. 5–21, July 1996.

[13] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan, "Evaluation of TCP vegas: Emulation and experiment," *Proc. of ACM SIGCOMM '95*, pp. 185–205, 1995.

[14] Volkan Ozdemir Injong Rhee and Yung Yi, "TEAR: TCP emulation at receivers - flow control for multimedia streaming," Tech. Rep., NCSU, Apr. 2000.

[15] C. Shannon, D. Moore, and K. Claffy, "Beyond folklore: Observations on fragmented traffi c," *IEEE Journal of Selected Areas in Communication'02*, 2002.

[16] D. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, vol. 17, pp. 1–14, 1989.

[17] Yang R. Yang and Simon S. Lam, "General aimd congestion control," *Proceedings of the International Conference on Network Protocols'00*, Nov. 2000.

[18] Dorgham Sisalem and Henning Schulzrinne, "The loss-delay based adjustment algorithm: A TCP-friendly adaptation scheme," *Proc. of the Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video'98*, July 1998.

[19] Minseok Kwon and Sonia Fahmy, "TCP increase/decrease behavior with explicit congestion notification (ECN)," *IEEE International Conference on Communications'02*, vol. 4, pp. 2335–2340, Apr. 2002.

[20] S. Floyd, "TCP and explicit congestion notification," *ACM Computer Communication Review*, vol. 24, no. 6, pp. 10–23, 1994.

[21] K. Ramakrishnan and S. Floyd, "A proposal to add explicit congestion notification (ECN) to IP," *RFC 2481*, Jan. 1999.

[22] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: A simple model and its empirical validation," *Proc. of ACM SIGCOMM '98*, pp. 303–314, 1998.

[23] Milan Vojnovic and Jean-Yves L. Boudec, "On the Long-Run behavior of Equation-Base rate control," *Proc. of ACM SIGCOMM '02*, 2002.

[24] Yang Richard Yang, Min Sik Kim, and Simon S. Lam, "Transient behaviors of TCP-friendly congestion control protocols," *Proc. of IEEE INFOCOM '01*, pp. 1716–1725, 2001.

[25] Jitendra Padhye, "Model-based approach to TCP-friendly congestion control," 2000.

[26] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *ACM/IEEE Transactions on Networking*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[27] W. Willinger, M.S. Taqqu, R. Sherman, and D.V. Wilson, "Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level," *Proc. of ACM SIGCOMM '95*, pp. 100–113, Aug. 1995.

[28] W.E. Leland, M.S. Taqqu, W. Willinger, and D.V. Wilson, "On the Self-Similar Nature of Ethernet Traffic (Extended Version)," *ACM/IEEE Transactions on Networking*, vol. 2, no. 1, pp. 1–15, Feb. 1994.

[29] A. Bouch, G. Wilson, and M. A. Sasse, "A 3-Dimensional approach to assessing end-user quality of service," *Proc. of the London Communications Symposium*, pp. 47–50, 2001.

[30] D. Lin and R. Morris, "Dynamics of random early detection," *Proc. of ACM SIGCOMM '97*, pp. 127–137, September 1997.

[31] M. May, J. Bolot, C. Diot, and B. Lyles, "Reasons not to deploy RED," *Proc. of 7th. International Workshop on Quality of Service (IWQoS'99)*, pp. 260–262, 1999.

[32] R. Jain and K. K. Ramakrishnan, "Congestion avoidance in computer networks with a connectionless network layer: Concepts," *Proceedings of the Computer Networking Symposium88*, pp. 134–143, 1988.

[33] National Institute of Standards and Technology, "Nistnet, a network emulation package," .

[34] M. Li, M. Claypool, and R. Kinicki, "MediaPlayer versus RealPlayer – a comparison of network turbulence," *Proceedings of the ACM SIGCOMM Internet Measurement Workshop'02*, 2002.

[35] A. C. Dalal and E. Perry, "An architecture for client-side streaming media quality assessment," Tech. Rep. HPL-2002-90, Hewlett-Packard Laboratories, Palo Alto, CA, 2002.

A Alternative approaches to reduce sending rate fluctuation

Besides MARC, we have also explored other alternatives to reduce sending rate fluctuation without using fair share utilization history.

A.1 Reaction with Probability

First we consider an alternative where a TFRC receiver reacts to congestion signal with probability α, where $0 \leq \alpha \leq 1$. With this approach, the loss event rate calculated by the receiver will reduce less aggressively than TFRC. The advantage of this scheme is it only modifi es TFRC minimally and the modifi cation only involves receivers. However, given that a receiver responds to a congestion signal with probability α and there are N congestion signals within a RTT, the probability for a sender not to be notifi ed of the congestion will be $(1-\alpha)^N$. This leaves the sender a chance to continue increasing its sending rate even under congestion. Also, since TFRC's loss calculation is already based on the *loss event rate*, which counts multiple congestion signals within a RTT as a single loss event, as long as TFRC reacts to one of the congestion signals within an RTT, the calculation will result in the same packet loss even rate. This would prevent us from achieving a traffi c profi le free from abrupt rate reduction, even though the probability of such abrupt changes is smaller than that in TFRC.

A.2 Constrained Rate Changes

To reduce the rate reduction magnitude in TFRC, another scheme is to simply limit the rate changes between two consecutive RTTs. In [19], the authors proposed a mechanism to reduce TCP's congestion window fluctuation by differentiating ECN signals from packet loss in TCP's congestion window calculation. We adopt a similar approach. In our scheme, TFRC's self-clocked rate reduction on congestion is proportional to the number of ECN signals detected during the previous feedback interval if there is no packet loss. If there is packet loss, our scheme behaves the same as TFRC. More specifi cally, we modifi ed TFRC as follows:

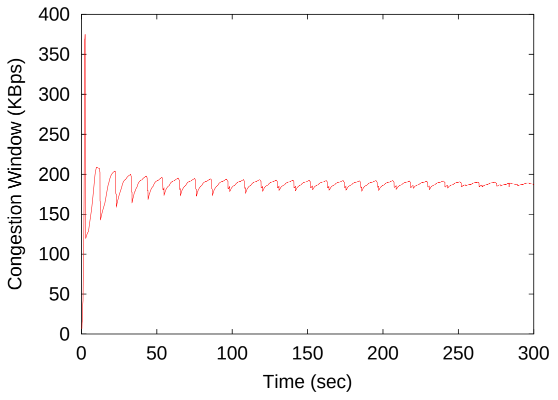
1. The receiver calculates the percentage of the ECN signals for each interval. We denoted this percentage as R_{CE}; If neither ECN signal or packet loss is detected, we have $R_{CE} = 0$; If there is packet loss, we set $R_{CE} = 1$.

2. R_{CE} will be included in the feedback message sent to the sender.

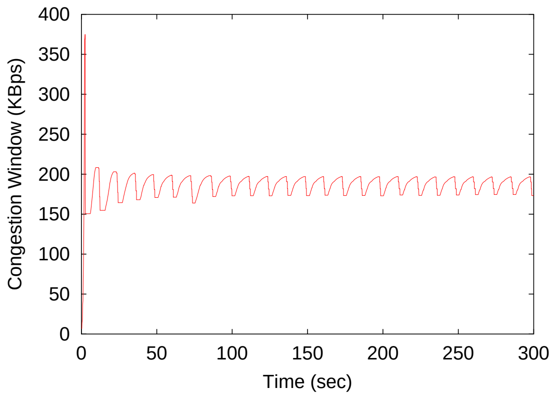
3. On receiving a feedback message, the sender calculates a new sending rate T'. Should $T' < T$, where T is the current sending rate, The sender reduces its sending rate to $T - R_{CE} * (T - T')$ instead of T'.

Similar to our discussion in Section 4.1, it is possible that the sending rate will decrease when a congested interval is followed by a non-congested period. Hence we keep the sending rate unchanged when no congestion signal is detected. It ensures a non-decreasing sending rate when no congestion signal is detected. Compared to MARC, a side effect of simply limiting the sending rate changes is that it could violate TCP-friendliness, i.e., a connection will send more traffi c than the current TFRC under the same network condition.

Fig. 26 compares the sending rates between the modifi ed scheme and TFRC. During the early stages of the connection, the modifi ed scheme gives a more stable traffi c throughput due to the constrains. However, throughout the 300-second simulation period, the sending rate of the modifi ed scheme is not able to converge. We also experimented with longer simulation time and we made the same conclusion. We expect such behavior is to be caused by the slower response to the congestion signals. To confi rm our conclusion, we repeat our simulation with fi xed R_{CE} in the feedback. Our experiment result shows that when R_{CE} is smaller than a threshold value, the modifi ed scheme always fails to converge.



(a) TFRC sending rate



(b) TFRC sending rate with constrained rate change

Figure 26: TFRC without competing traffi c

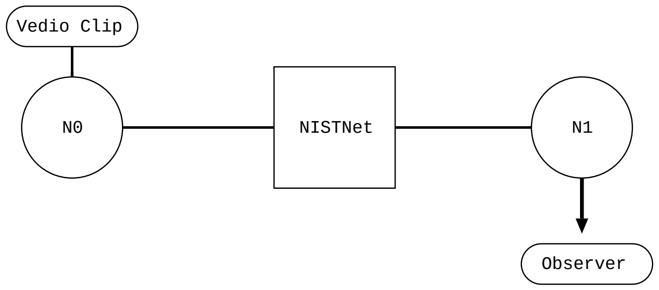


Figure 27: Emulation setup

B Perceived Streaming Quality with Different Congestion Control Schemes

We have designed a simple experiment running on a testbed to demonstrate MARC's performance improvement over TFRC. The goal of this experiment is to examine how transient sending rate changes in a congestion control mechanism affect the perceived quality of streaming applications. A direct approach for our evaluation is to use different congestion control mechanisms to support streaming applications. However, it requires implementations of these congestion control mechanisms in our experiment environment. It also requires modifi cation to the streaming applications, whose source code is not available to us. We decided to carry out our evaluation by simulating the congestion window or self-clocked rate changes in these mechanisms.

We obtain the traces for TFRC and MARC's self-clocked rates and TCP's congestion window using the *ns* simulator. The traces are collected when these congestion control mechanisms are used for a streaming video session. According to these traces, we then emulate the changes in congestion window or self-clocked rate with *NISTNet* [33]. NISTNet is a network emulation package that allows a machine to be set up as a router to simulate a wide variety of network conditions such as bandwidth and delay. In our experiments, through changing the bandwidth on NISTNet, we emulate congestion window changes in different congestion control mechanisms.

Our experiment setup is shown in Fig. 27. Hosts $N0$ and $N1$ are holding a *Windows Media Player* session through a middle box where NISTNet [33] is running. Node $N0$ is the media server and node $N1$ is the client. We use Windows Media Player in order to obtain a streaming data rate that can be more closely approximated by constant bit rate traffi c [34]. However, we note that Window Media Player's initial buffering mechanism would result in some degree of tolerance towards transient congestion. This implies that the perceived quality in our demonstration will be slightly better than those in the cases where initial buffering is not preferred, e.g., real time conferencing. With this setting, by adjusting the available bandwidth on NISTNet, we are able to emulate the sending rate of TCP, TFRC, and MARC. All the components are located in an isolated network. Suffi cient

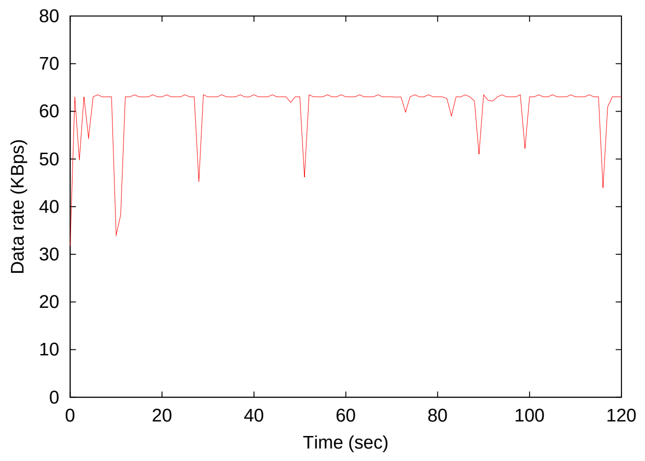


Figure 28: Data rate for the Clip S

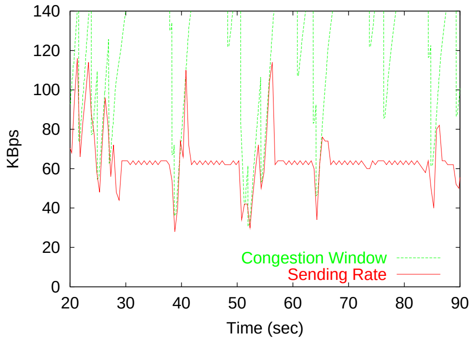
bandwidth are provided between $N0$ and $N1$ when NISTNet is not activated. During all the experiments, we use the default settings in Window Media Player except that we set the initial buffering time to 1 second.

We use Clip S from [35] in our experiment. Its data rate captured by `tcpdump` is shown in Fig. 28. The average sending rate is 60.47 KBps. To incorporate the sending rate into our simulation, we feed the sampled data rate into `ns` as the data source.

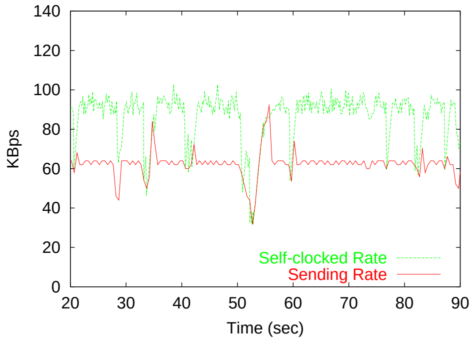
In our demonstration, we emulate the scenario where there is transient background traffi c. To introduce transient background traffi c, similar to our experiments in Section 5.1, we introduce a flash crowd consisting of N short-lived TCP flows. The settings of these short-lived flows follow those in Section 5.1. Flash crowd is introduced at the 50th second and lasts for 5 seconds. The bottleneck bandwidth is 1.5 Mbps. We then adjust the bandwidth on the NISTNet gateway according to the self-clocked rate from the simulation to emulate different congestion control mechanisms. By varying N, we can simulate different magnitudes of the transient flows. We show the self-clocked rate as well as the sending rate for TCP, TFRC and MARC in Fig. 29 to Fig. 31, with N ranges from 50 to 200.

These fi gures clearly shows that when the transient background traffi c is small, both TFRC and MARC are able to maintain steady sending rate. However, RTT fluctuation introduced by queueing effect introduce slight sending rate fluctuation to MARC. With the increasing amount of transient background traffi c, TFRC responses by reducing its self-clocked rate sharply, which causes the sending rate to drop responsively. In this case, MARC reduces its sending rate more gracefully. We notice that the connection's sending rate is larger than its data rate after the congestion period. This owes to the packets buffered at the bottleneck gateway during the congestion period.

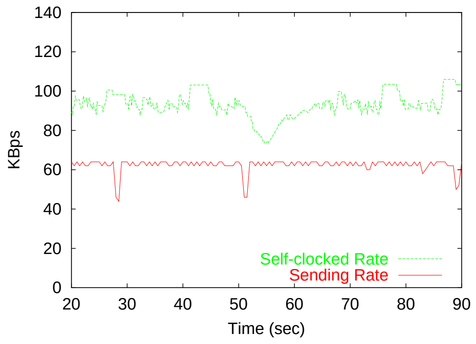
For practical reasons, we can only give an overview on the perceived video quality during the demonstration here. When $N = 50$, TFRC's sending rate has a slight drop while the visual quality under MARC is not affected. After increasing N to 100, MARC's self-clocked rate decreases slightly but it doesn't change the sending rate. Under the same scenario, there is signifi cant self-clocked rate decrease for TFRC and the visual quality degrades signifi cantly. After we further increase N to 200, the streaming session using TFRC experiences large amount of frame losses. Under the same background workload, the streaming session using MARC only sees some degree of quality degradation for a short period.



(a) TCP

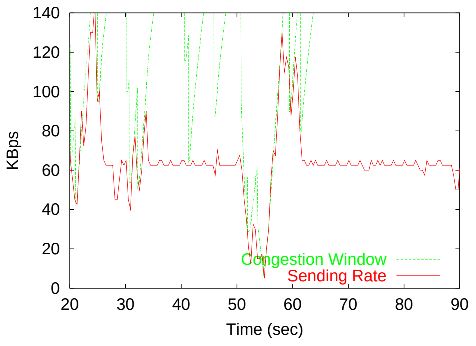


(b) TFRC

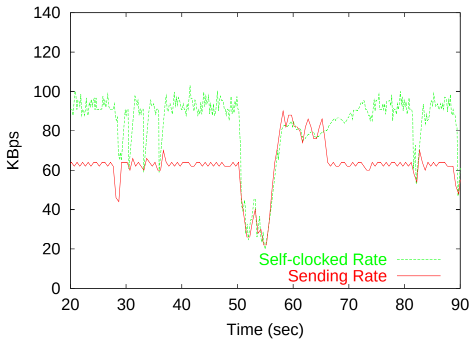


(c) MARC

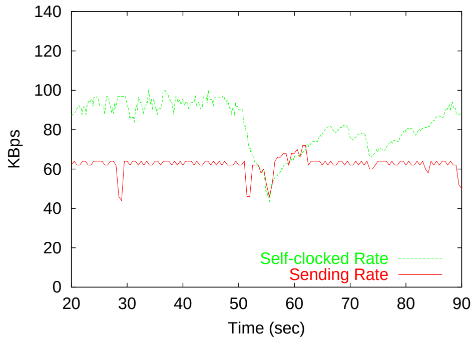
Figure 29: Self-clocked used in our demonstration: N=50



(a) TCP

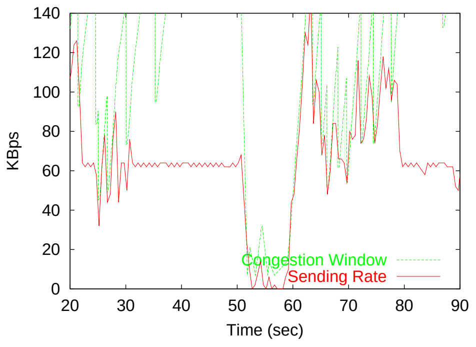


(b) TFRC

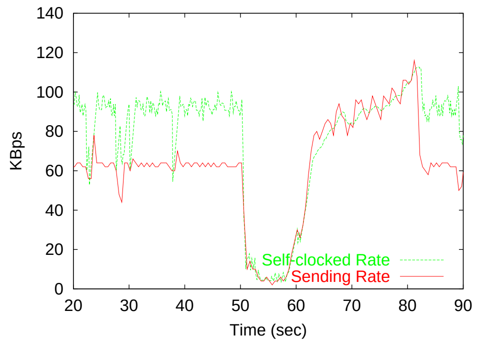


(c) MARC

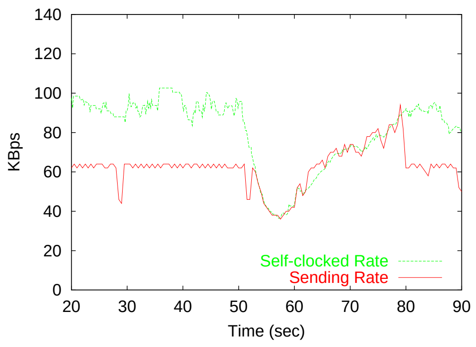
Figure 30: Self-clocked rate used in our demonstration: N=100



(a) TCP



(b) TFRC



(c) MARC

Figure 31: Self-clocked rate used in our demonstration: N=200