Congestion Control and an Analysis of TCP-friendly Rate Control

By Mei Lin
(carine@cs.utexas.edu)

Supervisor: Prof. Chris Edmondson-Yurkanan
(chris@cs.utexas.edu)

Computer Science Honors Thesis
CS379H
Fall 2003

Department of Computer Sciences
The University of Texas at Austin

Congestion Control and an Analysis of TCP-friendly Rate Control

Abstract

This paper presents a history of congestion control research and an analysis of a specific rate-based congestion control protocol, TCP-friendly rate control (TFRC). Two taxonomies of congestion control mechanisms are presented, and later used to classify general end-to-end congestion control schemes including window-based and rate-based methods. The classic TCP congestion control mechanisms are introduced as an instance of window-based congestion control. To address the research in congestion control for UDP flows, we discuss datagram congestion control protocol (DCCP) and the congestion manager (CM). Implemented in DCCP, TFRC is examined in detail for its TCP-friendliness and ability to provide smooth transmission for the applications. The sender's and receiver's protocols are presented using AP notation, and the equations used in the protocols are analyzed for their conduciveness to TCP-friendliness and smooth transmission. Lastly, two empirical studies of TFRC performance are summarized to validate the effectiveness of this protocol.

1. Introduction

In October 1986, the first occurrence of what is called "congestion collapse" spurred the studies in Internet congestion control [JK88]. During this event, the dropping of throughput from 32 Kbps to 40 bps between Lawrence Berkeley Lab (LBL) and UC Berkeley fulfilled Nagle's earlier prediction of such phenomenon [N84]. In fact, research in congestion control and flow control had begun even before the collapse in 1986. Because congestion control is often related to the types and the patterns of Internet traffic, which largely depends on the kinds of applications most commonly used, the continually changing Internet environment has kept the research area of congestion control active even decades later.

Congestion collapse is indicated by near-zero data throughput in a connection. Too many senders sending packets that are too large or too frequently is the cause of network congestion [KR01]. The large data flows of bursty traffic, can quickly saturate the network bandwidth, overflowing the buffers of the intermediate routers. Because packets at the congested routers now have to wait in an extended queue, and some even get dropped due to the lack of buffer space, the delay or the absence of acknowledgements result, which in turn triggers the retransmission timeout at the senders [APS99]. As the duplicates of the dropped or the slowly-transmitted packets enter the already congested network, the congestion condition becomes worse, and the retransmission timeout continues to expire at the senders. Thus, no new packets will be transmitted while the senders keep attempting to retransmit the packets that are dropped or assumed to be dropped. The throughput will eventually reach the minimum. The cycling of the state of low throughput is congestion collapse.

Congestion control is the mechanism for preventing congestion collapse. Its targets are the routers inside the network where congestion collapse occurs. Congestion control can work from the sending source by regulating traffic entering the network, or directly on the router by managing packet arrivals in the router's buffer queue. In this paper, we will focus our discussion on the first approach, end-to-end congestion control. An end-to-end congestion control regulates the amount or the rate at which the data is sent according to the network conditions. It can be viewed as a transport-level congestion control, because the protocol is implemented on the transport layer of the sender and the receiver, and estimates the network congestion level basing on information returned or not returned by the receiver. Congestion control is often implemented together with flow control, the concept of which is different from congestion control. Flow control is the regulation of traffic to prevent buffer overflow at the destination, the receiver, which provides sender with information about its buffer availability. For instance, the TCP congestion control, which is an end-to-end congestion control mechanism, incorporates the feature of TCP flow control as well. The number of segments can be sent is determined by taking the minimum of the congestion window and the receive window (see section 2.3). Here, congestion window is the limit to prevent network congestion, while receive window is the limit to prevent destination buffer overflow. The TCP congestion control mechanism also introduces the notion of congestion avoidance, which refers to the additive increase and multiplicative decrease phase (AIMD) of the protocol (see section 2.3). Congestion avoidance is characterized by the adjustment of window size with a certain degree of reservation. The window size is increased in at a slower rate compared to the slow start phase, and is decreased drastically.

Without the congestion control mechanisms, TCP was vulnerable to congestion collapse. Even though the implementation of TCP congestion control has successfully stabilized the Internet, the studies continued because of the lack of regulation for UDP flows. UDP is a best-effort (unreliable) protocol that provides no connection establishment, error recovery, flow control, or congestion control. It is popular among real-time and streaming media applications, to which reliability is not as important as timeliness, smooth transmission and sending of large amount of data (see section 2.4). When notified of network congestion, while TCP decreases transmission (see section 2.3), UDP flows do not adjust their transmission, thus continue to take advantage of the bandwidth made available by TCP's congestion control. The result is that UDP flows unfairly dominate the bandwidth, and the throughput of TCP flows reaches minimum. Thus, with the congestion control mechanisms, TCP is still vulnerable to low throughput with the presence of unregulated UDP flows.

Today, the affordability and versatility of the Internet and its resources cause a significant increase in traffic, especially that of real-time streaming media applications, which run on top of UDP. For instance, software for online conferencing, streaming video/audio, multi-user games, etc. has become increasingly popular. This situation drew researchers' attention to the lack of congestion control mechanisms in the best-effort protocols, for which the TCP congestion control mechanisms are inappropriate, as we will discuss later.

Some key general congestion control principles that the protocol designs should follow are fairness, robustness, scalability, and feasibility [LB99]. Fairness means concurrent flows of different congestion control protocols should have approximately equal throughput [YKL01]. Fairness is often known as TCP-friendliness, which is the main motivation for congestion control for UDP. Robustness refers to the ability of the congestion control protocol to act against misbehaving users who may try to manipulate parameters used in the congestion control to its own advantage while risking the stability of the network. It is a property of congestion control from the security perspective, and its implementation often involves identity verification [WESSA01]. We will not discuss protocols with robustness in this paper. Lastly, scalability entails the feature of adapting heterogeneous flows to various bandwidth.

Section 2 is a study of past congestion control research in the general-to-specific fashion. We first present two congestion control taxonomies to show how the various implementations of congestion control can be categorized. This will provide a macroscopic view of the congestion control schemes. Then we examine the difference between two streams of end-to-end congestion controls, window-based and rate-based congestion controls, and see where they fit under the two taxonomies. TCP congestion control and its variants, which are examples of window-based schemes, are then discussed. This is important because it is the transport-level congestion control in use, and other congestion controls are designed to be compatible with these schemes. The following section will be an introduction to a few other congestion control schemes for UDP-based applications. This leads to our analysis of a particular rate-based congestion control, TCP-friendly rate control (TFRC). Section 3 is devoted to the description and examination of TFRC to the granularity of equations used and procedures taken in sender and receiver. We will discover how rate-based schemes are designed to be TCP-friendly and to provide application with smoother transmission than TCP congestion control would. Before concluding, we survey two experiments on the performance of TFRC for validating its design goals. In general, as we will see in section 4, the empirical results are good.

2. A History of the Congestion Control Research

2.1 Taxonomies of Congestion Control Schemes

The principles by which the taxonomies differentiate the various schemes demonstrate the different ways to view the common characteristics and distinctions in these congestion control schemes. The taxonomies also provide large pictures encompassing various congestion control mechanisms.

Gerla and Kleinrock's taxonomy, which dates back to 1980, classifies congestion control as well as flow control mechanisms based on the layer of the network where congestion is to be prevented [GK80]. In this taxonomy, the term "flow control" is used throughout to refer to flow control and congestion control on all levels of the network: *hop level*, *entry-to-exit level*, *network access level*, and *transport level*. On the hop level, the control focuses on avoiding local buffer congestion (e.g. on a router) to ensure transfer between neighboring routers. The schemes on the hop level are in fact congestion control, rather than flow control. The entry-to-exit level flow control schemes works to avoid end router's buffer overflow by limiting amount of transfer from the router at network entry. On the network access level, throttling the incoming traffic is the goal when the internal network is congested. The network access level schemes should also be considered congestion control. And the transport level flow control attempts to ensure reliable delivery of packet from the source process to the destination process by preventing buffer overflow at the destination host. The entry-to-exit level and transport level flow controls are similar in that they are both concerned with the buffer over flow at the destination, thus controlling data flow from the source. The difference is that the entry-to-exit level flow control is only on the two end routers, while the transport level flow control extends a bit further out to the end hosts. Because the taxonomy is based on the locality of the congestion, a robust congestion control scheme is likely to spread over multiple levels. Therefore, a particular flow control scheme can very well be a hybrid scheme according to this taxonomy. For example, the end-to-end congestion control is likely to be network access level and transport level; because it regulates the amount of data transmission based on network congestion as well as receiver's buffer availability.

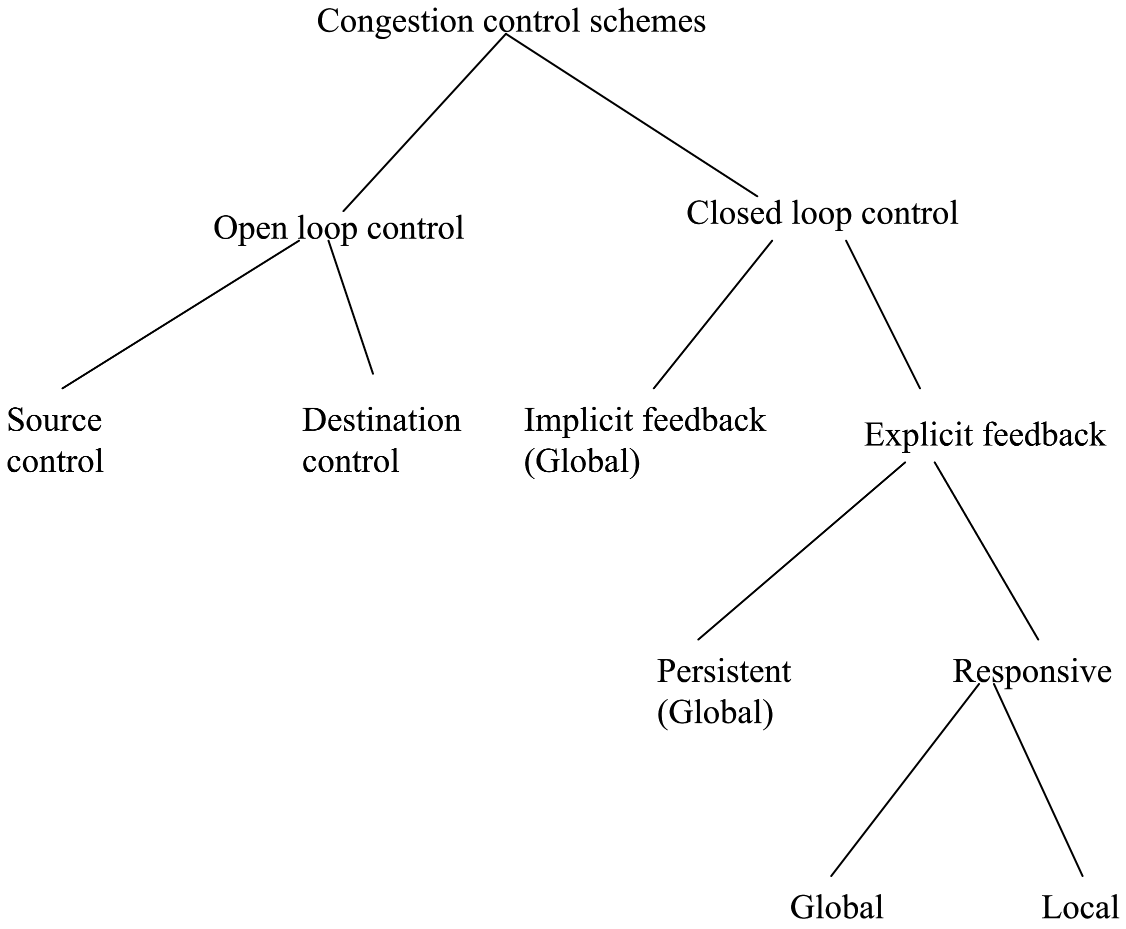

Figure 1: Control theory based taxonomy [YR95]

Yang and Reddy's paper, a more modern taxonomy of congestion control schemes, divides the various schemes into *open-loop* and *closed-loop* controls based on control theory [YR95]. They view the network as a large and complex control system, where control policies are needed in order to maintain stability. Analogously, congestion control schemes are the control policies, and are separated into open-loop and closed-loop categories according to whether the feedback is used. The two categories are further divided into sub-categories (see figure 1). The open-loop congestion control schemes do not provide or use feedback functionality; thus, the regulation is based on local knowledge of the network. Open-loop schemes often involve how to distribute buffer space and bandwidth to arriving packets, how many packets should be admitted into the network, and how to discard packets when buffers are full. This category spreads over hop-level and network access level in Gerla and Kleinrock's taxonomy. The closed-loop congestion control schemes utilize either *implicit* or *explicit feedbacks* and regulate transmission dynamically. The closed-loop control schemes using implicit feedback rely on observation of the network. Usually, the absence of certain events is the trigger for action. The explicit feedback category, on the other hand, receives information regarding the network condition from the feedback messages. Thus, the control is more proactive with the tradeoff of increased network traffic. Moreover, the category of explicit feedback is further divided into, (1) *persistent feedback*, feedback is generated and sent periodically; (2) and *responsive feedback*, feedback is sent only when triggered by certain network conditions. [YR95] points out that open-loop control is generally less robust against all traffic patterns of the network. It is not necessarily so, when open-loop schemes are used in a relatively small network with predictable behaviors, where there are fewer variations in traffic patterns. In fact, open-loop schemes are more efficient than closed-loop schemes, if enough knowledge about the network has been gathered by conducting large number of experiments and statistical analysis. This way, no feedback traffic needs to be generated to the network, and no delay or over-reaction is created. The latter advantage is especially important, because in closed-loop, due to delay of feedback and non-precision in the adjustment process, it often takes some time for the state of the network to reach stability; open-loop schemes do not have this problem, simply due to its non-reactive nature. Therefore, an open-loop scheme with well-research pre-determined parameters and policies can produce more desirable performance than a closed-loop scheme. However, in a network with capricious traffic patterns, appropriate close-loop schemes should be implemented to adapt to the dynamical behaviors that may lead to congestion.

2.2 Window-based versus Rate-based Congestion Control

In this section, we will compare the two main approaches of the end-to-end congestion control, *window-based* and *rate-based* schemes, and apply the network-level based and control theory based taxonomy techniques on these two types of schemes.

Window-based congestion control is also known as credit-based congestion control. For congestion control, the size of the window limits the amount of data that the sender is permitted to send. The window size increases at a certain rate if there are few or no packet losses, and decreases or resets when the network becomes congested. The fine-tuning of the rate of increase while probing for bandwidth and the rate of decrease based on congestion severity is critical for the performance of the protocol.

Window-based methods are often designed for applications that need fast adaptation to traffic change. However, this responsiveness often creates large fluctuation in transmission. Thus, for applications that prefer smoother transmission, rate-based schemes are more suitable. In rate-based schemes, the sender transmits data according to the sending rate, which is periodically recalculated using the changing network parameters (e.g. loss rate, round trip time, etc.). In tradeoff to the smoother transmission, the rate-based methods are often slow to respond to freed-up bandwidth. Although they respond slower to congestion as well, they share bandwidth fairly with other flows with window-based congestion control schemes.

Window-based and rate-based congestion control schemes are both on network-access level and transport level in Gerla and Kleinrock's taxonomy. Although one is by the quantity of transmission and the other the rate of transmission, both regulate the data entering the network and to the destination host. These two types of schemes are in the closed-loop category in Yang and Reddy's taxonomy, because it is clear that the congestion control in both are based on feedback information. In addition, implicit and explicit feedbacks are used in both schemes. Usage of implicit feedback is reflected in actions according to the observation of absence of feedback; and usage of explicit feedback is apparent.

2.3 TCP Congestion Control

Following the congestion collapse in 1986, Jacobson proposed the original TCP congestion control flavor, TCP Tahoe [JK88]. It works in two phases: slow-start, and AIMD congestion avoidance. TCP Tahoe and other TCP variants use window-based congestion control schemes. The two variables that indicate the shifting of one phase to another are *cwnd* and *ssthresh*. The variable *cwnd* is the number of segments that the sender is allowed to send at one time. The *cwnd* value increases exponentially in the slow-start phase and linearly in the AIMD phase. The variable *ssthresh* sets the upper bound of *cwnd* in the slow-start phase and before it enters AIMD phase. Once *cwnd* exceeds *ssthresh* as it grows in the slow-start phase, it increases linearly in the AIMD phase.

Not as its name suggests, the slow-start phase grows rather fast, but starts low. *cwnd* is initialized to 1. As long as the acks for the segments sent are received before the corresponding timeouts, *cwnd* is incremented by the number of acks received (the case of acks arriving after timeout will be discussed shortly). Thus, assuming the acks are received before their corresponding timeouts, the sender starts by sending one segment; after the ack of that segment arrives, the sender sends two; after the arrival of those two acks, the sender sends four, and so on. *cwnd* increases exponentially, and eventually reaches *ssthresh*. The AIMD phase now begins. During AIMD, sender is transmitting at a window size larger than the threshold, thus the protocol conservatively probe the available bandwidth while actively cuts back whenever necessary. For every *cwnd* number of acks received before their timeouts, instead of incrementing *cwnd* by the number of acks, *cwnd* is now incremented by 1. As *cwnd* continues to grow beyond *ssthresh*, the network capacity will eventually reach its limit of stability, and become congested or start losing packets. The congestion or the loss of packets is identified by the tardiness or absence of acks. In both cases, the sender will not receive acks before their timeouts. To recover, once an ack does not arrive after its timeout, *ssthresh* is set to half of the current cwnd, and *cwnd* is set to 1 returning to the slow-start phase.

As briefly mentioned in section 1, flow control is incorporated in TCP congestion control by using a variable *rcvwnd*. While *cwnd* guards against congestion inside the network, *rcvwnd* limits the amount of transmission to prevent buffer overflow on the receiver. The receiver attaches its most current value of *rcvwnd* in each segment sent to the sender; and the sender keeps track of the last byte sent and the last byte acked, and makes sure that the difference does not exceed *rcvwnd.* To clearly present the congestion control aspect of TCP congestion control, the description of congestion control in the above paragraph disregards flow control; thus, in integrating flow control, TCP takes the minimum of *cwnd* and *rcvwnd* for the number of segments to send. This allows the transmission to not exceed the limit for preventing network congestion as well as the limit for preventing destination buffer overflow.

TCP Tahoe works effectively against congestion, however shortcomings of this scheme were discovered and studied. Hence, many variants of TCP flavors were proposed to improve efficiency and throughput. TCP SACK is designed to provide information in the acks for the sender to make an intelligent decision on which packet to retransmit [MMFR96]. The fast retransmit algorithm attempts to shorten the time (ack timeout) that the sender needs to wait in order to detect a loss [APS99]. In TCP Reno, fast recovery was proposed to prevent sender from entering the slow-start phase upon the detection of a loss [APS99]. The fast recovery algorithm is further improved in TCP NewReno to achieve a better performance in various scenarios [FH99].

While most other TCP flavors have the same fundamental congestion control mechanisms, TCP Vegas differs from the rest [BOP94]. Rather than simply observing the acks and detecting losses, it keeps track of round trip time (RTT) and throughput rate and change the window size according to the difference between the actual and expected throughput rates. TCP Vegas estimates an RTT by recording the timestamp when a segment is sent and subtract it from the timestamp of the arrival of its ack. The smallest RTT is called BaseRTT, which is used in calculation of the expected throughput, (current window size)/BaseRTT. The actual throughput is calculated measuring the RTT and number of bytes transmitted. For the congestion avoidance scheme, the difference between the expected and the actual throughput, diff, is compared to two parameters to determine if cwnd should be linearly increased or decreased. In TCP Vegas's slow-start, the window size is increased exponentially every other RTT. And when the actual throughput is lower than the expected throughput—indicating there is congestion in the network—congestion avoidance starts. By measuring RTTs and calculating throughput constantly, TCP Vegas gives a better estimate of network condition than the original TCP congestion control mechanisms. However, the recording and sending of timestamps, and the calculation and comparisons of throughputs add overhead to the segments and the sender process.

2.4 Other Congestion Control Mechanisms

TCP congestion control has been successfully implemented on the transport level. And the Internet has been stable ever since. However, the research did not stop there. While TCP congestion control limits the sending of segments of the TCP connections, UDP, the connectionless and unreliable protocol, has no regulation on the transport level. Applications that run on UDP, such as online conference, real media player, multi-user games, etc., typically send large amount of data within a certain amount of time to ensure quality of service. They use the lightweight protocol UDP, because it allows freer and more frequent sending of large amount of data without error recovery such as retransmission. In fact, these applications can tolerate a certain data loss rate, but requires a certain sending rate. Consider a user who is trying to watch a trailer of an upcoming movie. The frame rate requires that the images be sent in a timely manner that the video will not be choppy. If a few images are lost during the transmission, as long as they are reasonably spaced out, the user probably can hardly notice the loss; however, the retransmission of the loss images would be extraneous—the user would not be interested in seeing the frames that are a few seconds late. Many researchers believe that in order to maintain the stability of today's Internet, congestion control mechanisms for UDP need to be implemented as well to achieve the fair sharing of bandwidth with the TCP flows.

Balakrishnan et al. proposed the congestion manager (CM) that works between the application and transport layers. This application- and transport-level independent approach allows cooperation between flows and apportioning bandwidth to different flows [BRS99]. The congestion manager uses a hybrid of window-based and rate-based congestion control mechanism to provide proper regulation with traffic shaping. The CM adaptation API communicates with the applications and the transport layer by passing information about the network congestion and varying bandwidth, so that the applications can make intelligent decision on what to send. The simulation results show that the CM effectively performs congestion control while providing sufficient transmission rate for streaming media applications such as real audio. Although the CM is able to coordinate flows on the same network path, the end-to-end nature of this mechanism poses limitation on flows joining on to the same network path from other sources. Thus, this unique approach works to its best only when widely implemented at the senders and the receiver. Moreover, it appears that the applications need to have the knowledge of the existence and semantics of the CM in order to take advantage of the information received from it. Therefore, the removal of the dependency of congestion control on the application level creates a new inter-dependency of exchanging of data.

Even though many applications running on top of UDP have their own built-in congestion control schemes, congestion control adds too much complexity to the applications, and may be too difficult for the applications to handle properly [KHF03]. Thus, Floyd et al. proposed the transport-level Datagram Congestion Control Protocol (DCCP), which is designed to replace UDP with the additional congestion control features. DCCP allows the application to choose from two congestion control schemes: TCP-like congestion control [FK032] and TCP-friend rate control (TFRC) [FK033]. Due to the different needs of the applications—some prefer a aggressive TCP-like probing scheme, and others prefer a relatively more stable transfer—the choice of the two congestion control schemes provides more flexibility. The choice of congestion control scheme and other features are decided by exchanging a set of values during the negotiation phrase between the sender and the receiver [KHFP03][KHF03].

In the next section, we will pore over one of the congestion control schemes of DCCP, TFRC.

3. TCP-friendly Rate Control (TFRC)

We now start exploring in detail a specific rate-based congestion control scheme, TCP-friendly rate control (TFRC). [HFPW03] and [FHPW00] are the main references of most of materials in this section. We start by providing an overview of this protocol, then present the sender and receiver behaviors in AP notation [G98], and finally, analyze the equations and procedures in the sender's and the receiver's protocols.

3.1 Overview of TFRC

TFRC is an end-to-end transport-level congestion control scheme for unicast flows in a best-effort environment. Intended for real-time applications, TFRC is designed to have smoother throughput than TCP while being TCP-compatible. It responds to changes in available bandwidth more slowly than TCP, and adjusts sending rate periodically according to change of loss event rate. In steady-state, TFRC flows use no more bandwidth than TCP flows under the comparable conditions. TFRC achieves this by using similar congestion avoidance algorithms and a simplified Reno TCP throughput equation with parameters used by TCP congestion control algorithms. This will be discussed in detail in section 3.4.1.

TFRC is receiver-based, i.e. most of the calculation is done by the receiver. The values carried by sender's data packets are used by the receiver to calculate the loss event rates, which are sent periodically to the sender in the feedback packets. The sender receives feedback packets from the receiver and adjusts its sending rate accordingly. Receiver-based is a desirable feature, because the sender is likely to be a server handling numerous connections simultaneously; the receiver only receives data packets most of the time, thus is probably less occupied. Assigning the receiver the task of calculation allows more efficient use of receiver's CPU time in the sender-receiver system. Moreover, this feature provides a solid basis for developing congestion control for multicast traffic. In a multicast environment, one sender transmits data to multiple receivers. If the protocol is sender-based, heavy workload will be assigned to the sender computer leaving all the receivers few jobs to do. Hence, receiver-based is the more desirable option.

We should also note that, TFRC adjusts the sending rate of packets of a fixed packet size, while TFRC-PS (specified in a different document) has fixed sending rate and varied packet sizes.

Two types of packets are used in this protocol: the data packets sent by the sender, and the feedback packets sent by the receiver. Each data packet contains the following values:

- Sequence number
- Sender's timestamp when packet is sent
- Sender's current estimate of RTT

And each feedback packet contains,

- Sender's timestamp found in the most recent data packet received
- Time between the receipt of last data packet and the generation of this feedback packet
- The current receiving rate of data packets since the last feedback packet was sent
- Receiver's current estimate of loss event rate

3.2 The sender's behavior in AP notation [G98]

Table 1: TCP Throughput Equation Variables

Var. name	Description
s	packet size (bytes)
R	round trip time (seconds)
p	loss event rate (0 – 1.0) (number of loss events/number of packets sent)
t_RTO	TCP retransmission timeout value (seconds)
b	number of packets acked by one ack message

```
process s

inp q : float {init 0.9, used for EWMA}
inp s, b : float {TCP throughput equation inputs, see table 4.1.1}
inp t_mbi : float {init 64, maximum interpacket backoff interval (see section 4.4)}
inp R, tld, t_RTO  : float {TCP throughput equation inputs, see table 4.1.1}
var X : float {init 1, sending rate}
var nofb : float {init 2, nofeedback timer}
var snd : float {init s/X, sending data packet timer}
var t_recvdata, t_delay, X_recv, p : float {values from feedback packet, see section 4.4}
var R_sample, X_calc: float {RTT sample, sending rate obtained from TCP thpt equation}
var 1fb : Boolean {init true, true if it is the first feedback packet received}
var t_now : float {current timestamp}
var seq : integer {sequence number of outgoing data packets}
def fb_pk(float, float, float, float), dt_pk(integer, float, float) : msg {packet definitions}

begin

rcv fb_pk(t_recvdata, t_delay, X_recv, p) from r ->
        R_sample := (t_now – t_recvdata) – t_delay ;
        if 1fb -> R, 1fb := R_sample, false
        [] ~1fb -> R := q*R + (1-q)*R_sample
        fi ;
        t_RTO := 4*R ;
        if p > 0 -> X_calc := {TCP throughput equation (s, R, b, p, t_RTO)};
                    X := min(X_calc, 2*X_recv) {lower bound = s/t_mbi};
        [] p <= 0 -> if (t_now – tld) >= R -> X := min(2*X, 2*X_recv) {upper bound = s/R);
                                                tld := t_now
                     [] (t_now – tld) < R -> skip
                     fi
        fi;
        nofb := max(4*R, 2*s/X)

[] {nofb expires} -> if {R_sample is not null} ->
                        if X_calc > 2*X_recv -> X_recv := max(X_recv/2, s/(2*t_mbi));
                        [] X_calc <= 2*X_recv -> X_recv := X_calc/4
                        fi;
                        if p > 0 -> X_calc := {TCP thput equation (s, R, b, p, t_RTO)};
                                   X := min(X_calc, 2*X_recv) {lower bound = s/t_mbi};
                        [] p <= 0 -> if (t_now – tld) >= R ->
                                                X := min(2*X, 2*X_recv) {upper bound = s/R);
                                                tld := t_now
                                     [] (t_now – tld) < R -> skip
                                     fi;
                        fi;
                     [] {R_sample is null} -> X := max(X/2, s/t_mbi)
                     fi;
                     nofb := max(4*R, 2*s/X)

[] {snd expires} -> send dt_pk(seq, t_now, R) to r;
                   snd := s/X;
                   seq := seq + 1

end
```

3.3 The receiver's behavior in AP notation [G98]

The receiver protocol is more complex than the sender protocol given that the receiver needs to analyze message loss and message reordering in order to estimate the loss event rate. Thus, we present the AP notation of receiver protocol in two parts: error handling and loss event rate calculation. The error handling protocol shows how the receiver detects lost packets, reorders packets arrived out-of-order, and keeps track of loss events. And the protocol for the loss event rate calculation demonstrates how the receiver uses the values in the data packets along with its own records to estimate the loss event rate.

3.3.1 The Error Handling Protocol

```
process r

inp n : integer {init 8, number of loss intervals used in estimating p}
var p : float {init 0, loss event rate}
var p_prev, fb, t_sent, R, I_tot0, I_tot1, I_tot, W_tot, I_mean, t_now, t_recv : float
var X_recv : float {init 0}
var seq : integer {nonnegative, sequence number in the data packet}
var init : Boolean {init true, first data packet received}
var I : array [integer] {an array of loss event intervals}
var w : array [integer] of float
var i : integer {nonnegative, index}
var recv_data : Boolean {init false}
var nr : integer {sequence number of next data packet expected, init. 0}
var lpk : array [0..2, integer] of integer
```

```
{log of lost pk, each elem: lost pk seq num (init –1), lost pk ts, subseq pk cnt (init 0), loss event
num (init –1)}
var lpk_s : integer {init. 0}
var lpk_e : integer {init. 0}
var levnt_t : array [integer] of float {timestamp of the first lost pk of the lost events}
var levnt_n : array [integer] of integer {seq num of the first lost pk of the lost events}
var levnt_e : integer {init. 0}
var ni : integer {init. 0}
var found : Boolean
var k, j : integer
var s_bf : integer
var s_aft : integer
var s_loss : integer
var t_bf : float
var t_aft : float
var t_loss : float

begin

rcv data_pk(seq, t_sent, R_i) from s ->
    k := lpk_s;
    {count the number of pks arriving after the lost pk}
    do k < lpk_e ->
        if (lpk[k][0] < seq) ^ (lpk[k][2] < 3) ->
            lpk[k][2] := lpk[k][2] + 1;
            if lpk[k][2] = 3 ->
                lpk_s := k+1;
                if (levnt_e = 0) v (lpk[k][1] – levnt_t[levnt_e-1] > R_i) ->
                    levnt_t[levnt_e] := lpk[k][1];
                    levnt_n[levnt_e] := lpk[k][0];
                    lpk[lpk_e-1][3] := levnt_e;
                    I[ni] := lpk[k][0] – levnt_n[levnt_e – 1];
                    ni := ni + 1 ;
                    levnt_e := levnt_e + 1
                [] (levnt_e > 0) ^ (t_loss – levnt[levnt_e][0] <= R_i) ->
                    lpk[lpk_e-1][3] := levnt_e - 1
                fi;
            [] lpk[k][2] < 3 -> skip
            fi
        [] (lpk[k][0] >= seq) v (lpk[k][2] = 3) -> skip
        fi;
        k := k+1
    od;
    s_aft, t_aft := seq, t_now;
    if seq = nr -> nr := nr + 1
    [] seq < nr -> {erase a previously lost packet}
                  found, k := false, 0;
                  do (~found ^ k<lpk_e)  ->
                   if lpk[k][0] = seq -> found := true;
                                         lpk[k][0] := -1;
                                         if lpk[k][2] = 3 ->
                                           if levnt_n[lpk[k][3]] = lpk[k][0] ->
                                             j := k + 1;
```

```
                               {update loss event array and I}
                               do j < lpk_e ->
                                   if lpk[j][2]=3 ^ lpk[j][3]=lpk[k][3]
                                        -> levnt_n[lpk[k][3]]:=lpk[j][0];
                                          levnt_t[lpk[k][3]]:=lpk[j][1];
                                          I[lpk[k][3]-1]:=
                                   levnt_n[lpk[k][3]]-levnt_n[lpk[k][3]-1];
                                          I[lpk[k][3]]:=
                                   levnt_n[lpk[k][3]+1]-levnt_n[lpk[k][3]]
                                   [] lpk[j][2]~=3 ^ lpk[j][3]~=lpk[k][3];
                                          j:=lpk_e
                                        -> skip
                                   fi; j:=j+1
                               od
                              [] levnt_n[lpk[k][3]] ~= lpk[k][0] -> skip
                              fi
                            [] lpk[k][2] < 3 -> skip
                            fi;
                            lpk[k][2] := 0
               [] l_pk[k][0] ~= seq -> k := k+1
               fi
              od;
     [] seq > nr -> {log a lost packet}
              k := 0;
              do k < seq – nr ->
                    s_loss := nr + k;
                    t_loss := t_bf + ((t_aft - t_bf)*(s_loss – s_bf)/(s_after – s_bf));
                    lpk[lpk_e][0] := s_loss;
                    lpk[lpk_e][1] := t_loss;
                    lpk[lpk_e][2] := 1;
                    lpk_e := lpk_e + 1;
                    k := k+1
               od;
     fi;
     s_bf, t_bf := seq, t_now;
     {calculation the loss event rate p}
     if p > p_prev -> {cause fb to expire}
     [] p <= p_prev -> skip
     fi;

[] {fb expires} -> {prepare and send fb pk}

end
```

3.3.2 The Procotol for the Loss Event Rate Calculation

```
process r

{same variables as the error handling protocol}
begin

rcv data_pk(seq, t_sent, R_i) from s ->
     recv_data, t_recv := true, t_now;
```

```
        if init -> fb, p, X_recv, init:= R_i, 0, 0, false;
                  {cause fb to expire}
        [] ~init -> skip
        fi;
        {add the data packet to the packet history};
        p_prev, i := p, 0;
        do i < n ->
        if i < n/2 -> w[i] := 1
        [] i >= n/2 -> w[i] := 1–(i–(n/2–1))/(n/2+1)
        fi; i := i + 1
        od;
        I_tot0, I_tot1, W_tot, i := 0, 0, 0, 0;
        do i < n -> I_tot0 := I_tot0 + (I[i]*w[i]);
                    W_tot, i := W_tot + w[i], i + 1;
        od;
        i := 1;
        do i <= n -> I_tot1, i := I_tot1 + (I[i]*w[i-1]), i + 1;
        od;
        I_tot := max(I_tot0, I_tot1);
        I_mean := I_tot/W_tot;
        p := 1/I_mean;
        if p > p_prev -> {cause fb to expire}
        [] p <= p_prev -> skip
        fi;

[] {fb expires} -> if recv_data -> i := 0;
                                   do i < n -> if i < n/2 -> w[i] := 1
                                               [] i >= n/2 -> w[i] := 1–(i–(n/2–1))/(n/2+1)
                                               fi; i := i + 1
                                   od;
                                   I_tot0, I_tot1, W_tot, i := 0, 0, 0, 0;
                                   do i < min(n, ni) ->
                                          I_tot0 := I_tot0 + (I[ni – i - 1]*w[i]);
                                          W_tot, i := W_tot + w[i], i + 1;
                                   od;
                                   i := 1;
                                   do i <= n -> I_tot1, i := I_tot1 + (I[i]*w[i-1]), i + 1;
                                   od;
                                   I_tot := max(I_tot0, I_tot1);
                                   I_mean := I_tot/W_tot;
                                   p := 1/I_mean;
                                   X_recv:= {number of packets received in last R_i seconds}/R_i;
                                   send fb_pk(t_sent, t_now-t_recv, X_recv, p) to s;
                                   recv_data := false
                    [] ~recv_data -> skip
                    fi;
                    fb := R_i;

end
```

3.4 Analysis of TFRC Equations

In this section, we explore the equations and procedures used by the sender and the receiver, and analyze how they contribute to the smoother throughput and TCP-friendly features of TFRC.

3.4.1 Analysis of Sender Equations and Procedures

Sender's actions depend on whether a feedback packet is received or nofeedback timer expires. Initially, the sending rate is set to 1 packet per second, and the nofeedback timer is set to 2 seconds.

When a feedback packet is received, the sender performs the following 5 steps: calculate a most recent sample of RTT, estimate a new smoothed RTT, calculate the TCP retransmission timeout value, adjust the sending rate, and reset the nofeedback timer.

1. *Calculate a sample of RTT*

```
R_sample := (t_now – t_recvdata) – t_delay ;                    (equ. 1)
```

The sender calculates a new RTT sample every time a feedback packet is received. Then it recalculates the RTT estimate based on the past values and the new RTT sample. t_recvdata is the sending time of the last data packet received by the receiver upon the generation of this feedback packet, and t_delay is the time elapsed from the receipt of the last data packet to the generation of this feedback packet on the receiver (see figure 2). Thus, equation 1 gives the most recent sample of RTT.

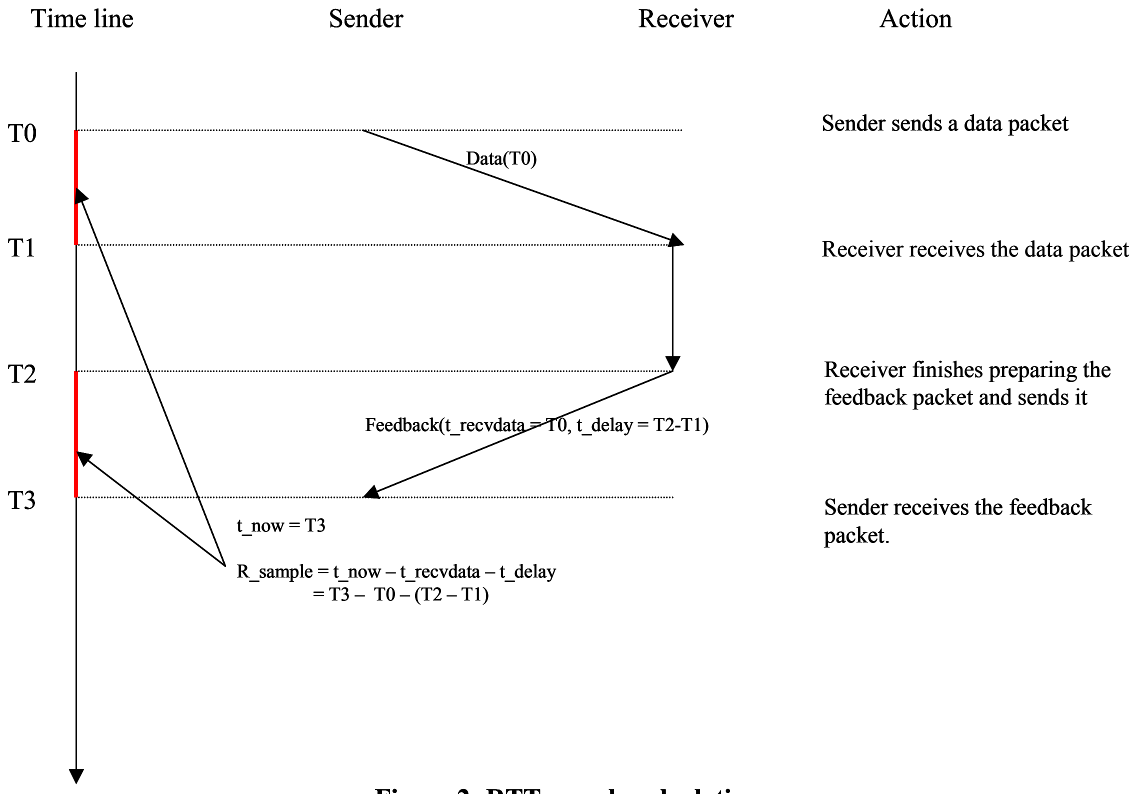
Time line
Sender
Receiver
Action
T0
T1
T2
T3
Data(T0)
Feedback(t_recvdata = T0, t_delay = T2-T1)
t_now = T3
R_sample = t_now – t_recvdata – t_delay
= T3 – T0 – (T2 – T1)
Sender sends a data packet
Receiver receives the data packet
Receiver finishes preparing the feedback packet and sends it
Sender receives the feedback packet.

Figure 2: RTT sample calculation

2. *Estimate RTT*

```
if 1fb -> R, 1fb := R_sample, false
[] ~1fb -> R := q*R + (1-q)*R_sample                                    (equ. 2)
fi ;
```

Equation 2 comes directly from TCP's estimation of RTT [JK88]. Thus, when the RTT value is used in the TCP throughput equation, a good estimate of TCP sending rate will be obtained. This equation yields the exponential weighted moving average (EWMA) of RTT. By using a q value close to 1, the equation puts more weight on the more recent RTT samples. Consider the EWMA of a 5th estimate of RTT, R_5. Let the first estimate of RTT be R_0 = R_sample$_0$.

$$R_5 = q^{5\,*}R\_sample_0 + q^4 * (1\text{-}q) * R\_sample_1 + \ldots + q^0 * (1\text{-}q) * R\_sample_5.$$

So,

$$R_n = q^n * R\_sample_0 \sum_{i=1}^{n} q^{n-i} * (1-q) * R\_sample_i \;.$$

The first RTT sample is weighted the most. Then, the coefficient is the smallest for the second RTT sample, and increases gradually toward the most recent RTT sample. Usually, q is set to be 0.9; the performance of TFRC is not affected by the exact value of q.

3. *Calculate the TCP retransmission timeout value*

```
t_RTO := 4*R ;                                                          (equ. 3)
```

The TCP throughput equation—which will be discussed shortly—that is used by TFRC to estimate the sending rate needs a retransmit timeout value t_RTO. In TCP algorithm, it is estimated as,

t_RTO = R + 4*R_var, where R_var is the variance of RTT.

However, it is difficult to accurately model the TCP retransmit timeout value, because the various TCP flavors use drastically different clock granularities to measure this value. Further, TFRC does not rely on this value to determine the retransmission time. Thus, a rough estimate of it does not result considerable inaccuracies. Through experiments, equation 3 evolved as an acceptable heuristic estimate.

4. *Adjust the sending rate*

```
if p > 0 -> X_calc := {TCP throughput equation (s, R, b, p, t_RTO)};
            X := min(X_calc, 2*X_recv) {lower bound = s/t_mbi};          (equ. 4)
[] p <= 0 -> if (t_now – tld) >= R -> X := min(2*X, 2*X_recv) {lower bound = s/R);
                                                                         (equ. 5)
                                      tld := t_now
             [] (t_now – tld) < R -> skip
             fi
fi;
```

The TCP throughput equation used to calculate X_calc is

$$\text{X\_calc} = \frac{s}{R * \sqrt{\frac{2bp}{3}} + t\_RTO * (3 * \sqrt{\frac{3bp}{8}} * p * (1 + 32p^2))} \qquad \text{(equ. 6)}$$

TFRC follows the same general principles for adjusting the sending rate. When loss event rate, p, is greater than zero, it recalculates the sending rate using the TCP throughput equation; otherwise, the rate is doubled. The TCP throughput equation is the simplified modeling result of Padhye et al [PFTK98]. using TCP Reno, which is the most widely implemented version of TCP in the Internet. The original equation is,

$$\text{X} \approx \frac{1}{RTT\sqrt{\frac{2bp}{3}} + t\_RTO \min(1,3\sqrt{\frac{3bp}{8}})p(1 + 32p^2)}$$

The empirical result of the work of Padhye shows that this equation accurately models the TCP throughput with a wide range of loss rates.

The result of the sending rate is then compared to twice the receiving rate measured by the receiver, and the lesser of the two values is chosen. Also, a lower bound of s/t_mbi is put on the adjusted sending rate. t_mbi is the maximum inter-packet backoff interval, and is set to 64 seconds. Thus, if the sending rate returned by the TCP throughput equation falls below the lower bound, it ensures that the sending rate is at least one packet per every 64 seconds.

When the loss event rate is zero, TFRC doubles the sending rate as TCP would. A variable named "time last doubled" (tld) is used to keep track of the timestamp when sending rate was most recently doubled; so only when it has been at least one RTT since tld, does the sending rate get doubled. Equation 5 doubles the sending rate conservatively by comparing it with twice the receiving rate measured by the receiver, and picking the lower value. Symmetric with adjusting the sending rate, equation 5 also puts a lower bound, s/R. So the sending rate is bumped up to at least one packet per RTT, if it was still very low.

5. *Reset the nofeedback timer*

```
nofb := max(4*R, 2*s/X)                                   (equ. 7)
```

Equation 7 updates the nofeedback timer to the greater of the two values: 4 times RTT and amount of time allowed for sending two packets.

When no feedback packet is received for an extended period of time, nofeedback timer will expire; and the sender performs these three actions: update the receiving rate last calculated by the receiver, adjust sending rate, and reset nofeedback timer.

1. *Update the receiving rate last calculated by the receiver*

```
if X_calc > 2*X_recv -> X_recv := max(X_recv/2, s/(2*t_mbi));      (equ. 8)
[] X_calc <= 2*X_recv -> X_recv := X_calc/4                        (equ. 9)
fi;
```

If no feedback packet is received after the nofeedback timer expires, the sender decreases the value of supposedly receiver-measured receiving rate, and recalculates the sending rate. This is also halving of the sending rate. Equation 8 is used when 2*X_recv was last chosen as the sending rate, thus, X_recv is halved with a lower bound of s/(2*t_mbi), meaning one packet every 128 seconds. Equation 9 is used with the condition that the sending rate calculated from the TCP equation was chosen, so X_recv is assigned X_calc/4, so that 2*X_recv = X_calc/2 will be used in the following step.

2. *Adjust sending rate*

If there has been feedback packets received since the establishment of the current connection, the same procedure as step 4 above is performed; Otherwise,

$$X := \max(X/2,\ s/t\_mbi). \qquad \text{(equ. 10)}$$

The initial sending rate of one packet/second is halved, and will continue getting halved if the absence of feedback persists.

3. *Reset nofeedback timer*

   And the nofeedback timer restarts as above.

3.4.2 Analysis of Receiver Equations and Procedures

The receiver is responsible for receiving data packets, calculating the loss event rate, and sending feedback packets. It performs a sequence of actions depending on whether a data packet is being received or the feedback timer expires.

The calculation of loss event rate is one of the core components of this protocol. The change of sending rate is partially based on the loss event rate (see section 3.4.1). Before discussing the loss event rate, it is necessary to first understand the detection of a loss, a loss event, and a loss interval. A data packet is considered lost if three packets of greater sequence numbers are received; and its late arrival can erase the packet loss in the history. A *loss event* contains one or more losses occurred during one round trip time. A packet is considered part of an existing loss event, if its timestamp is no more than one RTT larger than the timestamp of the first packet of the loss event; otherwise, this packet becomes the first packet of a new loss event (see figure 3). The *loss interval* is the number of packets within a loss event; it is obtained by subtracting the sequence number of the first lost packet in a loss event from the sequence number of the first lost packet in the subsequent loss event.

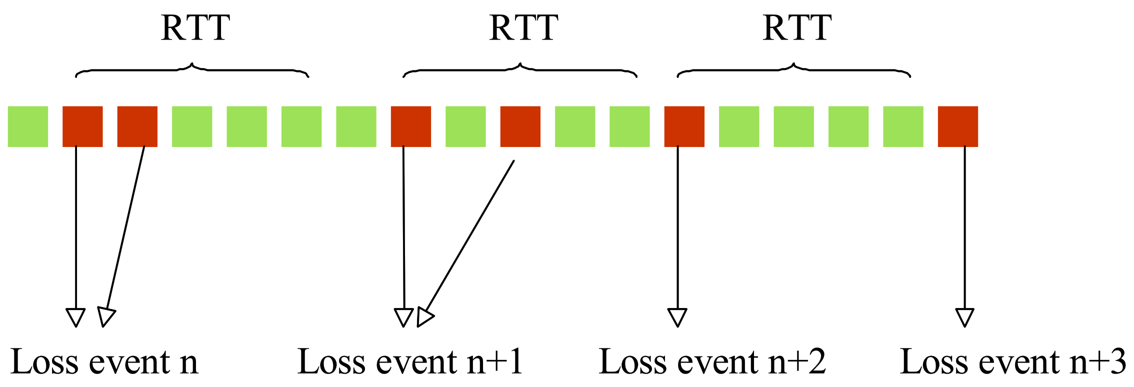

Figure 3: Loss events

Loss event rate is the ratio of the number of loss event and the total number of lost packets. Although it is seems sensible to estimate loss event rate as number of lost packet over total number of packets transmitted, this estimate is not an accurate representation of general TCP loss event rate due to the varied implementation of different flavors of TCP. Because different TCP variants halve the congestion window differently in response to several losses in a window of data, and different queue management schemes in the routers cause different packet loss patterns, by ignoring losses after one loss in a round trip time, TFRC's loss event estimate closely reflects the loss condition under most TCP variants.

The detailed calculation of the loss event rate is as follows:

```
I_tot0, I_tot1, W_tot, i := 0, 0, 0, 0;
do i < n -> I_tot0 := I_tot0 + (I[i]*w[i]);
            W_tot, i := W_tot + w[i], i + 1;
od;
i := 1;
do i <= n -> I_tot1, i := I_tot1 + (I[i]*w[i-1]), i + 1;
od;
I_tot := max(I_tot0, I_tot1);
I_mean := I_tot/W_tot;
p := 1/I_mean;
```

The loss event rate is measured over n most recent loss intervals. n is specified to be less than 8 so that the loss event rate can sufficiently reflect recent changes in network congestion level. The larger the value of n, the less the sender will adjust the sending rate, thus the more slowly the sender responds to severe packet loss. Moreover, n weights are used on the n most recent intervals, so that the more weights are given to the more recent intervals. The weighted total number of packets is divided by the sum of the weights to yield the average loss interval I_mean. And the reciprocal of I_mean is the loss event rate. The decision on whether to include the packet losses since the most recent loss event can effect the value of the loss event rate. The n intervals are shifted to include those packet losses if the resulting average loss interval is larger.

The frequency of the sending of feedback packets also determines the sender's responsiveness to network congestion. Receiving feedback packets more frequently allows sender to have the more current information about the network status; however, if the interval between the receipts of feedback packet is too short, the difference made by the extra packets may not justify the increased traffic in the network and increased workload on the receiver's CPU. Considering the tradeoffs, feedback packets should be sent at least once per RTT or for every data packet sent if the sending rate is less than one packet per RTT, and when a new loss event is detected.

The feedback packets contain a value, X_recv, the receiving rate within the previous RTT calculated by the receiver. It is simply the number of packets received in the past RTT divided by the RTT. This is an empirical value that is used by the sender to compare with the calculated sending rate. This value is chosen over the calculated sending rate when the theoretic value is inflated. See section 3.4.1 for more details.

4. Discussion of TFRC Empirical Performance Results

Two main goals of TFRC are (1) TCP-friendliness, where UDP flows share bandwidth fairly with TCP flows by responding appropriately to changing network conditions, and (2) smoothness, where UDP packets are transmitted at a sending rate with less abrupt rises and falls while being regulated.

To demonstrate that TFRC is a well-designed protocol, [FHPW00] tested TFRC on the public Internet, the Dummynet network [Riz98], and in the ns network simulator, with various numbers of TCP flows and bandwidth parameters. In a network with utilization greater than 90% at all times, the same number of TFRC flows and TCP flows are sharing a common bottleneck. The means of the throughputs of two types of flows measured over a number of small intervals were quite close. This observation shows that in a busy network, a number of TCP flows competing with a number of TFRC flows has a similar result as if all flows were TCP. And this observation is consistent with the protocol design, because TFRC conservatively adjusts its sending rate based on several network parameters using the TCP throughput equation (see section 3.4.1). This prevents TFRC flows to have throughput that deviates too far from that of the TCP flows under the same network conditions. However, the experiment did show considerable deviation of the throughputs of TFRC flows from the throughputs of TCP flows under certain conditions. The possible factors that contribute to the deviations are the measure of the loss rate, the TFRC receiver's calculation of the actual receiving rate, and the frequency of updating the round trip time and the sending rate. Thus, in a network of rapidly changing traffic patterns, TFRC, which does not respond as quickly as TCP, can have a throughput quite different than that of TCP. The variances of the throughputs of TFRC and TCP flows were also measured in the experiment. As we expected, the TFRC flows show a smaller variance than those of TCP flows [FHPW00]. This can be explained by the smooth sending rates of TFRC flows. Since the TFRC flows do not experience the halving in number of packets sent like the TCP flows do, and the reduction of sending amount is reflected by the sending rate over a period of time, the throughputs measured over the intervals for TFRC flows fluctuate less than those of TCP flows. Hence, the smoothness of TFRC is evident.

In [YKL01], four congestion control protocols are studied: TCP, GAIMD [YL00], TFRC, and TEAR [ROY00]. The studies of the protocols are based on four measurements, defined as follows: (1) fairness, small variations of sending rate compared with that of the competing flows; (2) smoothness, small variations of sending rate of one flow over time in a stationary environment; (3) responsiveness, the quickness of reduction of sending rate in response to congestion; (4) aggressiveness, the quickness of increase in sending rate for higher utilization when the network recovers from congestion and more bandwidth becomes available. Fairness and smoothness are measured in a stationary network environment with fluctuations. Responsiveness and aggressiveness are studied under increase in network congestion and increase in available bandwidth, respectively. [YKL01] measures and calculates the practical and theoretical sending rate coefficient of variation (CoV) in each study case. The value of practical CoV indicates the performance of the protocol. The calculated CoV is then compared with the observed CoV to check for consistency. In most cases, the results match the calculations. In the stationary environment, it was concluded that for all four protocols, smoothness and fairness are positively correlated. This echoes the earlier discussion about the speed of the changing of network traffic pattern and the difference in the mean throughputs of TCP and TFRC. This paper also agrees that smooth traffic patterns promote fairness between flows of different protocols. Thus, at a low loss rate, TFRC appears to perform well in smoothness and fairness; while as the loss rate reaches 20%, its smoothness and fairness were observed to be the worst among the four protocols. Although the numerical result does not seem optimistic, by examining the lines of TFRC and TCP in figure 5 carefully, we can see that the TFRC sending rate falls at appropriate places relative to the TCP sending rates, considering that TFRC is supposed to be smoothed in contrast to the fluctuations in TCP sending rates. In network environment with increased congestion, TFRC is shown to have relatively slower responding speed compared to TCP and GAIMD. This is expected for rate-based protocol as discussed previously. Interestingly, it was noticed that since TCP over-reacts to congestion, the amount of time for it to reach the stable state was actually approximately as long as those of slower-responding protocols. In the environment with increased bandwidth, although rate-based protocols are expected to have slower response, TFRC performed reasonably well with its history-discounting feature turned on. The history-discounting feature, which is not studied in section 3, is an optional feature that allows more weights to be put on the more recent intervals in calculating the loss event rate [HFPW03]. The motivation of this feature is to allow quicker response to the sudden disappearance of congestion. [YKL01] shows TFRC performs well in most conditions except when the loss rate is high. However, it seems to us that the TFRC sending rate is reasonable, although extremely low, relative to the TCP sending rate in the same condition.

According to the empirical studies, TFRC performs well in that it shares bandwidth fairly with the TCP flows, and that it yields a smooth sending rate for its application. However, TFRC can be slow in responding to changes in the network. With the aid of the history discounting feature, it can still achieve a reasonable quickness in response to available bandwidth.

5. Conclusion

We have explored the field of congestion control by studying the history of research in this area and a specific rate-based congestion control, TFRC.

The history of congestion control is presented from a macroscopic view gradually down to specific details. Using Gerla and Kleinrock's survey of congestion control and flow control mechanisms based on the targeted network layer of congestion, we classify end-to-end congestion control mechanisms, window-based as well as rate-based, to be on both entry-to-exit level and transport-level. Yang and Reddy's taxonomy of congestion control schemes takes a totally different approach by using control theory. The end-to-end congestion control mechanisms are closed loop control using both implicit and explicit feedbacks. Having generalized the characteristics of window-based and rate-based congestion control schemes, we introduce the classic TCP congestion control mechanism with slow start and AIMD congestion avoidance, and provide a brief description of other TCP variants. Differing drastically from most TCP flavors, TCP Vegas is pointed out for its usage of round trip time—instead of simply the acks--to adjust the congestion window. Due to the increasing popularity of applications running on top of UDP, such as real time and streaming media applications, the absence of regulation for UDP flows threatens the current stability of the Internet. To address the research on congestion control for best-effort protocols, we evaluate DCCP that works on transport level, and the congestion manager, which works between the transport and the application levels.

The main contribution of the paper is the detailed examination of TCP-friendly rate control (TFRC) in section 3, where the sender and receiver protocols are presented in AP notation, and the procedures and equations in the sender's and the receiver's protocols are studied for their originating source and their conduciveness to TCP-friendliness and transmission smoothness. To evaluate TFRC, two empirical studies of this protocol are presented for validating the TCP-friendly and smooth transmission features of TFRC, and with comparison to other window-based and rate-based congestion control protocols. From the experiment results shown in the papers, we conclude that when loss rate is high, the performance suffers. However, with a reasonably low loss rate, TFRC performs well as a congestion control mechanism for best-effort flows sharing bandwidth fairly with TCP flows without creating an abrupt change in the sending rate.

References

[APS99] Allman, M., V. Paxson, and W. Stevens. "TCP Congestion Control," IETF RFC2581, Apr. 1999.

[BRS99] Balakrishnan, H., H. Rahul, and S. Seshan, "An Integrated Congestion Management Architecture for Internet Hosts," Proc. ACM SIGCOMM, Cambridge, MA, Sept. 1999.

[BOP94] Brakmo, L., S. O'Malley, and L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance," In *Proceedings of the ACM SIGCOMM*, pp. 24-35, August 1994.

[FH99] Floyd, S. and T. Henderson, “The NewReno Modification to TCP’s Fast Recovery Algorithm.” IETF TFRC 2582, Apr. 1999.

[FK032] Floyd, S., and E. Kohler, “Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control.” Internet-Draft draft-ietf-dccp-ccid2-02, IETF, May 2003. Work in progress.

[FK033] Floyd, S., and E. Kohler, “Profile for DCCP Congestion Control ID 3: TFRC Congestion Control.” Internet-Draft draft-ietf-dccp-ccid3-02, IETF, May 2003. Work in progress.

[FHPW00] Floyd, S., M. Handley, J. Padhye, J. Widmer, "Equation-Based Congestion Control for Unicast Applications: the Extended Version," ICSI tech report TR-00-03, March 2000.

[G98] Gouda, M. G., *Elements of Network Protocol Design*, John Wiley & Sons, 1998.

[GK80] Gerla, M. and L. Kleinrock, "Flow Control: A Comparative Survey," *IEEE Transactions on Communications*, April 1980, pp. 553-574.

[HFPW03] Handley, M., S. Floyd, J. Padhye, and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification," RFC3448, Jan. 2003.

[JK88] Jacobson, V., and M. Karels, "Congestion Avoidance and Control," ACM SIGCOMM 88.

[KHFP03] Kohler, E., M. Handley, S. Floyd, and J. Padhye. “Datagram Control Protocol (DCCP).” draft-ietf-dccp-spec-05.txt, internet-draft, work in progress, October 2003.

[KHF03] Kohler, E., M. Handley, and S. Floyd, “Designing DCCP: Congestion Control Without Reliability.” Under submission, May 2003.

[KR01] Kurose, J. and K. Ross, *Computer Networking – A Top-Down Approach Featuring the Internet*, Addison Wesley Longman, Inc., 2001.

[LB99] Legout, A. and E. W. Biersack, "Beyond TCP-Friendliness: a New Paradigm for End to End Congestion Control," Technical Report, Institute Eurecom, June 1999.

[MMFR96] Mathis, M, J. Mahdavi, S. Floyd, and A. Romanow, “TCP Selective Acknowledgement Options.” RFC 2018, Oct. 1996.

[N84] Nagle, J., “Congestion Control in IP/TCP Internetworks.” IETF RFC896, Jan.1984.

[PFTK98] Padhye, J., V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," Proc. ACM SIGCOMM 1998.

[ROY00] Rhee, I., V. Ozdemir, and Y. Yi, “TEAR: TCP Emulation at Receivers – Flow Control for Multimedia Streaming,” Tech. Rep., Department of Computer Science, North Carolina State University, Raleigh, North Carolina, U.S.A., Apr. 2000.

[Riz98] Rizzo, L., “Dummynet and Forward Error Correction.” In *Proc. Freenix* 98, 1998.

[WESSA01] Wetherall, D., D. Ely, N. Spring, S. Savage, and T. Anderson, "Robust Congestion Signaling," IEEE International Conference on Network Protocols, Nov. 2001.

[YKL01] Yang, Y.R., M.S. Kim, S.S. Lam, "Transient Behaviors of TCP-friendly Congestion Control Protocols," *Computer Networks*, Volume 41, Issue 2, pages 193-210, February 2003.

[YL00] Yang, Y.R. and S. Lam, “General AIMD Congestion Control.” in *Proceedings ICNP 2000*, Osaka, Japan, November 2000.

[YR95] Yang, C. and A. Reddy, "A Taxonomy for Congestion Control Algorithms in Packet Switching Networks," *IEEE Network Magazine*, Jul./Aug. 1995, pp. 34-45.