Design and Analysis for TCP-Friendly Window-based Congestion Control

Soo-Hyun Choi
s.choi@cs.ucl.ac.uk
+44 (0)20 7679 0394
http://www.cs.ucl.ac.uk/staff/S.Choi

Supervisor: Prof. Mark Handley

Keywords: Internet Congestion Control

A document submitted in partial fulfillment
of the requirement for a
PhD Transfer Report
at the **University of London**
Department of Computer Science
University College London

October 10, 2006

Acknowledgment

I would like to give my special thanks to Prof. Mark Handley for his thorough and careful supervision throughout this research. Without his close guidance and numerous discussion, I would have not met this point. I also would like to thank to the members of Networks Research Group in the Department of Computer Science at University College London.

Abstract

The current congestion control mechanisms for the Internet date back to the early 1980's and were primarily designed to stop congestion collapse with the typical traffic of that era. In recent years the amount of traffic generated by real-time multimedia applications has substantially increased, and the existing congestion control often does not opt to those types of applications. By this reason, the Internet can be fall into a uncontrolled system such that the overall throughput oscillates too much by a single flow which in turn can lead a poor application performance. Apart from the network level concerns, those types of applications greatly care of end-to-end delay and smoother throughput in which the conventional congestion control schemes do not suit. In this research, we will investigate improving the state of congestion control for real-time and interactive multimedia applications. The focus of this work is to provide fairness among applications using different types of congestion control mechanisms to get a better link utilization, and to achieve smoother and predictable throughput with suitable end-to-end packet delay.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Area of Research

At the heart of the success of the Internet is its congestion control behavior. The Internet serves flows in a best-effort manner, which does not use bandwidth reservation mechanisms. Therefore, the Internet should be designed carefully in order to prevent applications from making the network congested with undesired work load. So far, the congestion control protocol at an end host has prevented applications from overshooting the network.

Transmission Control Protocol (TCP) [10] has been used as a standard congestion control protocol in the Internet. It uses the Additive-Increase and Multiplicative-Decrease (AIMD) algorithm, which probes the available bandwidth by increasing its sending window size additively, and responds to a congestion signal by decreasing its window size by half. Although TCP has served well to control the general type of Internet data services, it is not sufficient to satisfy applications that require different rate behavior than TCP. For example, TCP's AIMD mechanism may cause too much rate variation for a streaming media application which prefers smoother rate. Moreover, the rapid growth in using streaming media applications over the Internet has draw a great attention to a congestion control mechanism suitable for its preferences. Recently, many congestion control protocols [4, 7, 11, 18, 24] have been proposed, especially for streaming multimedia applications.

1.2 Problem Statement

A congestion control protocol, depending on its manner of estimating the available bandwidth and reacting to congestion signal, produces various rate behavior expecting to achieve the following criteria.

- Average Throughput – what is the bandwidth utilization with a congestion control protocol
- Rate Smoothness – how large the magnitude of rate variations is and how often the rate varies
- Responsiveness – how fast the congestion control protocol responds to changes in network conditions
- Fairness – what the bandwidth share ratio is when it competes with other flows

Typically, real-time multimedia streaming applications prefer a congestion control mechanism that can provide smooth and predictable sending rate with reasonable responsiveness. At the same time, they expect a congestion control to share bandwidth fairly with other flows while still maintaining high average link utilization.

While TCP has successfully controlled the Internet for most applications, TCP-Friendly Rate Control (TFRC) [7] has emerged as an adequate unicast congestion control mechanism for applications such as streaming media. However, it has been known to us that it has some limitations for such applications over certain network conditions [3, 5, 19], and also we have observed some other cases which we will explain as belows.

First, if an end-to-end traffic traverses a network similar to a Digital Subscriber Line (DSL) environment, TFRC does not show protocol fairness in that sources using TFRC can starve the same number of competing sources using TCP. The protocol fairness would be one of the key features that a congestion control protocol has to have across wide range of network conditions (e.g., bottleneck bandwidth, bottleneck queue length, bottleneck queue discipline, the number of source traffic, and so forth).

Second, because of the rate-based feature, TFRC lacks the TCP's Ack-clocking characteristics. The temporal spacing of the data packets depends on the link speed. In steady state, TCP only allows sending a packet when another packet is acknowledged. This feature is an elegant way to limit the number of outstanding packets in the network. This feature will help sending a packet at the link speed (or fair share of the bandwidth when competing with other flows) which will also help fully utilizing the available bandwidth. In this work, we introduce the acknowledgment (ACK) mechanism to bring it for the streaming applications.

Third, TFRC can be easily oscillated in a certain case. If the history update period is fast, and there is an on/off CBR traffic for a short period time in the network, then it is likely that TFRC throughput will be oscillated, especially under a low level of statistical multiplexing. Thus, TFRC introduced a bounding function to calculate the new sending rate (see Equation 3.2). This non-linear control equation may add unwanted long-term oscillation, which is contradictory to the initial design purpose. In this work, we would like to keep the smoothness without introducing any control equation which might add more noise on the throughput.

Finally, TFRC can be hard to implement in a real-world system. If the round-trip time is very small[1], then a processor on a general purpose machine can be hard to process the round-trip time update, which might add another noise on the throughput[2]. In this work, we design and develop a simple protocol by re-introducing TCP-like Ack mechanism to implement on a system in such environment. We will discuss further in Chapter 3.

1.3 Contributions and Scope

The main contribution of this research is to design and develop an Internet congestion control protocol to achieve smooth and predictable throughput for a real-time streaming application without loosing fairness property. Although TFRC has served well around these objectives, we have observed some limitations over *ns-2* [14] simulator: we will discuss further on this in Chapter 3. The main contributions of this work are:

- Smooth and Predictable Throughput Control
- Protocol Fairness
- ACK Clocking
- Protocol Simplicity than the Rate-based Congestion Control

In this work, from the above, we do not specify the criteria whether it is a network centric or a user centric. For example, the network utilization and protocol fairness can be network centric metrics whereas smoothness and responsiveness can be user centric metrics. Also, we do not consider the delay that application will receive in this work. But, after all, the delay is the key performance metric that a user might care. This work does not include an applicability study. We will conduct the application-level experimental evaluation later, but the main objective in this work is not the applicability test. We will mainly focus on the congestion control mechanisms itself whether our idea gives better performance for the various network condition. Therefore, this research basically will focus on the area if the real-time multimedia applications can really adopt a congestion control method that we are proposing.

[1] A typical RTT in LAN environment is normally less than 5 ms.

[2] Because a general purpose machine nowadays has 5 to 10 ms of CPU tick cycle, the RTT sample can be noisy if its value is much less than 5 ms. Recall that RTT estimation in TFRC is important when calculating the sending rate. See [20] for details.

1.4 Structure of the Report

This report is structured as follows. We give a brief introduction to TCP congestion control and TFRC protocol in Chapter 2. Chapter 3 details the problems and motivation. We explain our proposed congestion control protocol and its basic working mechanisms in Chapter 4. Chapter 5 describes the implementation details and protocol validations. Chapter 6 shows the results on the various evaluation for our protocol under wide range of network parameters. Then we conclude this report in Chapter 7.

Chapter 2

Background and Related Work

This chapter reviews two types of congestion control mechanisms in present to use different type of applications: window-based congestion control mechanism and rate-based congestion control mechanism. One of the well-known protocols for the window-based congestion control mechanism is TCP [10]. TCP is a connection-oriented unicast transport protocol, used to transport data for most common Internet applications. The other well-known protocol for the rate-based congestion control mechanisms is TCP-Friendly Rate Control (TFRC) [7, 21]. It is driven by the need of multimedia streaming over the Internet, which requires smooth rate adaptation rather than TCP's *Additive-Increase Multiplicative-Decrease (AIMD)* policy. It models TCP's throughput behavior as an equation [15, 16] to use its rate adaptation mechanism, and maintains its long-term throughput approximately equal to that of a TCP flow under the same network condition. In this chapter, we give an overview of the TCP and TFRC protocol.

2.1 TCP Congestion Control Protocol

Congestion control is imperative in order to allow the network to recover from congestion and operate in a state of low delay and high link utilization. Ideally, end systems need to send as fast as they can in oder to achieve high link utilization. However, if their sending rates exceed the network capacity, data will be accumulated at a bottleneck buffer, which can cause long delay or a packet loss. When the network is overloaded, the buffer at the bottleneck node starts to be filled up, and eventually will be overflowed. The network overloading stage is generally called congestion. If an application does not slow down its sending rate during the network congestion, most of the bandwidth would be used to transmit packets that will be dropped before it gets reached to the receiver. So, the goal of any type of congestion control is to keep end systems sending rate as fast as they can without creating too much network congestion.

2.1.1 TCP Functions

TCP as a transport layer protocol has many functions apart from the congestion control feature. The major functions of TCP are: congestion control, flow control, and reliability control.

- TCP Congestion Control: it prevents a sender from overwhelming the network by detecting congestion signal.
- TCP Flow Control: it prevents a sender from sending data packets faster than the receiver can process.
- TCP Reliability Control: it provides an in-order reliable data transmission.

2.1.2 TCP Overview

In this section, we review the basic mechanisms how TCP detects congestion and limits its rate, and we discuss the algorithm TCP uses for probing and responding to congestion.

Basically, on start-up, TCP performs a slow-start to quickly reach a fair share of the available network capacity without overwhelming the network. When TCP detects congestion, it halves its rate. In case of high congestion, many packets may be lost causing a retransmission timeout. If this happens, TCP goes back to slow start. In the following subsection, we describe how this works.

2.1.3 TCP Acknowledgment

TCP uses acknowledgments to carry feedback information for all three control functions mentioned above. Each time a receiver gets a data packet, it informs the sender of the sequence number that it expects to get. The packet used to inform the sender is called an *acknowledgment (ACK)*. ACKs can be piggybacked on data packets when the receiver has data packets to send back to the sender.

When there are no packet reordering events or losses, the ACK contains the sequence number of the packet following the one that has just arrived. If there is a packet loss, the ACK of a later packet contain the sequence number of the lost packet.

2.1.4 TCP Congestion Window

TCP limits its sending rate by controlling the *congestion window* size, which is the number of packets that may be transmitted in a flow. In general, the time between delivering a packet and receiving its ACK is a round-trip time (RTT). A TCP sender can send up to the congestion window size of data packets during one RTT. Once TCP sends out a window size of data packets, it can send new data packets only after some ACKs are arrived at the sender. Therefore, the average rate of a TCP over one RTT is roughly the window size divided by the RTT.

2.1.5 Congestion Signals

TCP thinks a packet loss as an indication of congestion, and detects packet losses with two mechanisms. The first way is using the *timeout* function. A TCP sender starts out a timer every time it sends a packet. If the timer expires before the sender receives the packet's corresponding ACK, TCP thinks the packet is lost. The retransmission timeout value has a significant impact on the overall TCP performance and is therefore continuously adapted to variations in the TCP's RTT. A detailed study of the effect of various timeout settings is presented in [1].

The second way that TCP detects packet losses is through *duplicate acknowledgments* (DupAck). TCP receiver acknowledges the sequence number the one next to the current packet sequence number. A packet loss causes the receiver to re-acknowledge the sequence number of the lost packet when the next packet arrives. The sender receives the duplicate acknowledgments for the same sequence number. Since packet reordering in the network can also cause duplicated acknowledgments, TCP uses a threshold to avoid treating reordering as packet losses. Typically, TCP sets the threshold to three (it allows 3 DupAcks). Only when a TCP receives three or more duplicated acknowledgments it consider that a packet is lost and thus generates a retransmission. The duplicated acknowledgments may detect packet losses earlier than the timeout timer, thus detecting congestion by duplicated acknowledgments is called *fast retransmission*.

2.1.6 Slow Start and AIMD

Besides congestion signals and how TCP uses a congestion window to limit its rate, the remaining aspect of TCP congestion control is its dynamical window adjustment algorithms. The basic rate control mechanisms are an exponential initialization state called *Slow Start* and an *Additive-Increase Multiplicative-Decrease (AIMD)* stage. Figure 2.1 illustrates this behavior which results in TCP's "sawtooth" behavior.

Since the first TCP implementations, TCP has been improved in several ways. Today, different versions of TCP are in use, the most common being TCP NewReno and TCP Sack [2, 17]. An overview of some of the different TCP behavior and their implications on protocol performance are given in [6].

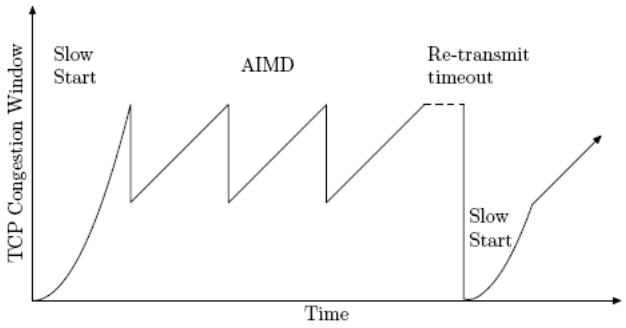


Figure 2.1: TCP Slow Start and AIMD control

2.2 TFRC Congestion Control Protocol

2.2.1 TFRC Overview

TCP-Friendly Rate Control protocol (TFRC) [7, 21] is an equation-based unicast congestion control protocol. The primary goal of equation-based congestion control is not to aggressively find and use available bandwidth, but to maintain a relatively steady sending rate while still being responsive to congestion. Thus, several design principles of equation-based congestion control can be seen different as to that of TCP's behavior.

- Do not aggressively seek out available bandwidth. That is, increase the sending rate slowly in response to a decrease in the loss event rate.
- Do not reduce the sending rate to half in response to a single loss event. However, do react to congestion in a manner that ensures TCP-compatibility.

In the following section, we give an overview of TFRC protocol and its properties.

2.2.2 TCP Response Function

The basic foundation of TFRC protocol is to keep a flow's sending rate not more than the TCP's rate at a bottleneck using drop-tail queue. For the best-effort traffic competing with TCP in the current Internet, the correct choice of the control equation became an important issue. In [15], it suggested a TCP response function as the below.

$$T = \frac{s}{t_{RTT}\sqrt{\frac{2}{3}p} + t_{RTO}\left(3\sqrt{\frac{3}{8}p}\right)p\left(1+32p^2\right)} \tag{2.1}$$

This gives an upper bound on the sending rate T in bytes/sec, as a function of the packet size s, round-trip time t_{RTT}, steady-state loss event rate p, and the TCP retransmission timeout value t_{RTO}.

In order to determine the sending rate T, it is required to know the following control parameters:

- The parameter t_{RTT} and p have to be determined. The loss event rate p must be calculated at the receiver, while the round-trip time t_{RTT} could be measured at either the sender or the receiver.
- The receiver sends either the parameter p or the calculated sending rate T back to the sender.
- The sender will increase or decrease its sending rate according to the value, T.

2.2.3 Sender Functionality

In order to use the TCP equation (2.1), a TFRC sender has to determine the values for the round-trip time t_{RTT} and retransmission timeout value t_{RTO}. With the measured t_{RTT}, the sender smoothes the t_{RTT} using an exponentially weighted moving average (EWMA). This will determine the responsiveness

of the transmission rate to changes in t_{RTT}. The sender could derive the retransmission timeout value t_{RTO} using the usual TCP algorithm:

$$t_{RTO} = SRTT + 4 * RTT_{var}, \tag{2.2}$$

where RTT_{var} is the variance of RTT, and $SRTT$ is the round-trip time estimate. In practice, the simple empirical heuristic of $t_{RTO} = 4 * t_{RTT}$ works reasonably well to provide fairness with TCP. Unlike TCP, TFRC does not use t_{RTO} to determine whether it is safe to retransmit. Instead, the important parameter is to obtain p in the feedback messages from a receiver. Every time the sender receives the feedback message, it calculates a new value for the allowed sending rate T using Equation (2.1).

Therefore, the critical functionality in a TFRC sender is to calculate the sending rate T using p, and determine whether it can increase or decrease the sending rate by comparing T to T_{actual}[1] at least once per round-trip time.

2.2.4 Receiver Functionality

The TFRC receiver provides two feedback information to the sender: a measured t_{RTT} and the loss event rate p. The calculation of the loss event rate is one of the critical parts of TFRC. The method of calculating the loss event rate has the following guidelines:

- The estimated loss event rate should track relatively smoothly in an environment with a stable steady-state loss event rate.
- The estimated loss event should measure the *loss event rate* rather than the packet loss rate, where a *loss event* can consist of several packet lost within a round-trip time. This was discussed in more detail at [7].
- The estimated loss event should have strong relation to loss events in several successive t_{RTT}.
- The estimated loss event rate should increase only in response to a new loss event.

Apart from the above guidelines, we define a *loss interval* as the number of packets between loss events. The estimated loss event rate should decrease only in response to a new loss interval that is longer than the previously-calculated average, or a sufficiently long interval since the last loss event.

There are a couple of ways to calculate the loss event rate, but TFRC uses the Average Loss Interval method. The Average Loss Interval (ALI) method computes the average loss rate over the last n loss intervals. The detailed description of the ALI method can be found at [7].

We have looked at the basics of current Internet congestion control protocols: TCP and TFRC. In the following chapter, we revisit the detailed motivation how our research find the gap and how it contribute to these research area.

[1] T_{actual} stands for the received rate at the $(n-1)^{th}$ packet transmission whereas T means the calculated rate for the n^{th} packet transmission.

Chapter 3

Motivation: Detailed Version

This chapter details the motivation of our research. We start talking about what TFRC was trying to achieve and how it provided solutions for real-time streaming applications. We discuss a few drawbacks which eventually led us in this research. Moreover, we give an example through *ns-2* [14] simulation providing us how TFRC performed badly in a certain network condition. Finally, we raise research questions which we try to solve them through this work.

3.1 TFRC and Its Defect

TFRC [7, 21] was introduced for real-time multimedia streaming applications, and it has served well in a variety way through many relevant research [3, 7, 12, 13, 15, 16, 21, 23]. The fundamental idea behind TFRC is that we want to produce a very *smooth* and *predictable* sending rate under various network conditions. Certainly, it is not desired for real-time streaming applications to change its sending rate drastically in case of a transient network congestion. TFRC took a rate-based congestion control using the TCP throughput response function as in Equation (2.1). The basics of the equation-based rate control is that the sending rate is mainly controlled by the TCP throughput equation with packet loss history. Although it has achieved some level of smoothness and predictable throughput, it also brought some inherent limitations [5, 19]. We will discuss it in the following section.

3.1.1 What's wrong with TFRC?

TFRC achieves an adequate sending rate using the TCP response function (Equation 2.1), but it really does not have fine-grained congestion avoidance features. In other words, it reduces or increases its sending rate based on a packet loss history information, but it lacks the congestion avoidance features like TCP (i.e., AIMD and slow start). On top of that, because it adjusts the sending rate using loss history information, the sending rate is likely being changed slowly even though t_{RTT} changes rapidly. Particularly, when there are small number of flows in the network, it is easy to take the whole network bandwidth quite quickly if we only use the rate-based control. Therefore, the TFRC sender limits the increase rate no more than the two times of the received rate since last report from the TFRC receiver. This is shown as the below.

$$\mathsf{Max\ Rate} = 2 * \mathsf{Rate\ Since\ Last\ Report} \tag{3.1}$$

In addition, although it has the above limiting factor, it will be easily oscillated. This is due to the fact TFRC will easily make the sending rate to the network limit at a relatively short time and keep overshooting briefly until the history information tells to the sender to cut down the rate. Moreover, the sender will continue to cut down the sending rate until the history information tells to the sender that there are available network resources. If this behavior happens in a longer term, then it would not be much problematic. But if this situation comes in a short period of time, it will generate oscillation in throughput. In order to prevent the throughput oscillation in a short period, the TFRC sender introduced a condition for its sending rate as the below.

$$\text{New Rate} \quad = \quad \frac{\sqrt{t_{RTT}}}{E(\sqrt{t_{RTT}})} * \text{Calculated Rate}, \tag{3.2}$$

where $E(\cdot)$ stands for the expected value of a round-trip time, and the received rate is the rate calculated at a TFRC receiver. This control equation does not have a solid proof whether it is going to work at a wide range of network parameters, but it simply worked for the oscillation case when it is developed. Therefore, there might not be a strong evidence that TFRC will not produce the throughput oscillation under those circumstances with Equation (3.2).

Apart from the two limitations, TFRC can be hard to implement *correctly* in a real world system, especially under a network with a very small round-trip time (e.g., less than 10 ms). If the round-trip time is very small, let say a few milliseconds, then a processor on a general purpose machine is unlikely to process the round-trip time update properly which will result in using a sub-sampled round-trip time, hence, it will eventually add more noise on the calculated throughput. Saurin [20] observed that even the non-conservative TFRC's limiter (which is bounded to twice the received rate) cuts in inappropriately on real systems.

So, the question is: did we actually loose all the benefit just because we used the rate-based throughput control mechanisms? can we get the same benefit (smooth and predictable throughput) but eliminate those limitations that TFRC has? The following section suggests our research idea on this question.

3.2 Window-based Rate Control

Our research objective is to have smooth and predictable throughput with congestion control features for a real-time streaming application, but we do not want to have the limitations that we mentioned in the previous section: throughput oscillation, unfairness, and protocol impracticality.

There are two factors for TFRC which might have caused those protocol limitations. First, TFRC is based on the rate-based control. Second, TFRC is based on the TCP response function. We assume that the TCP response function is reasonably well designed. Then, the rate-based mode became the suspicious part. In this work, we envisage that the limitations presented in TFRC are rooted from the rate-based congestion control, which made us to introduce an Ack-based window-based control mechanism like the normal TCP.

3.2.1 Motivation and Research Question

The motivation is to keep the nice throughput behavior of TFRC, but to remove the limitations. We bring the TCP-like Ack mechanism to control the sending window in packets, which will affect the throughput after all. Because the throughput is controlled by the sending window driven by Ack, we name it as TCP-Friendly Window-based Congestion Control (TFWC). The Ack-based window control is simple and cheap in the following context. First, when an Ack comes in, we increase the sending window by one. If an Ack does not come in, we do not increase (or reduce) the sending window. In this way we can also bring the TCP's Ack-clocking feature. Second, the Ack-based window control is relatively easier to implement than the equation-based rate control mechanism. Therefore, TFWC has at least two advantages against TFRC: cheaper and simpler. The protocol implementation may not be simple under *ns-2*, but it is going to be definitely simpler in a real world implementation.

The research questions that we will solve throughout this work are:

- Does TFWC have same benefit to that of TFRC's?
- Is TFWC better (fairer)?
- Does TFWC have better impulse response? For example, is TFWC less over-reactive?

The ideal solution that we would expect is to satisfy the above questions better than TFRC with the window-based control. If TFWC has the same benefit that TFRC has, if it is fairer than TFRC when it is competing with TCP traffic, and if it is less over-reactive than TFRC, then TFWC would be certainly a better choice than TFRC. The TFRC benefit is it produces much smooth and predictable throughput comparing to that of TCP. According to the original TFRC paper [7], we can say it is fair to TCP in

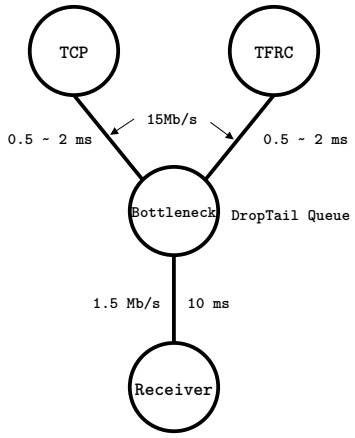


Figure 3.1: DSL-like Network Architecture.

sharing network resources across broad range of network conditions, but not always. It would be better if TFWC presents fairer response in a wide range of network parameters. Suppose there comes a CBR traffic with taking almost network limit, then TFRC is not going to cut down its sending rate right away as it is designed in that way. Once the history information tells to cut down the rate, it will continue reducing the rate until the history information tells to increase the rate. This rate-based mode could cause unnecessary throughput oscillation for a certain amount of period, which can be avoided under a window-based control. We will find a way to make TFWC satisfy all of the above questions through this research. In the following section, we give an example how TFRC performs badly in a certain network condition.

3.3 TFRC under a DSL-like link

An Internet congestion control protocol, like TCP and TFRC, should perform consistently well throughout the end-to-end network path. A congestion control protocol cannot be said to be fair or reasonable if it breaks the fairness in terms of throughput among various types of traffic sources over an end-to-end path. TFRC is specially designed for a real-time multimedia traffic to control the Internet congestion and to gain smooth throughput without breaking the fairness against the conventional TCP traffic. However, we have observed a bad case for TFRC under a DSL-like link using *ns-2* simulator. We have conducted *ns-2* simulation under the network condition as shown in Figure 3.1, and observed that TCP traffic will be starved if it competes with the same number of TFRC sources.

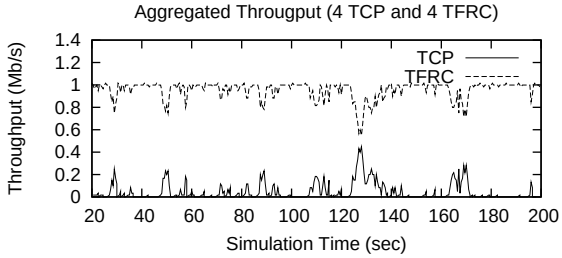
The bottleneck queue size is 5 packets with using drop-tail queue. The bottleneck link speed is 1.0 Mb/s and its link delay is 10 ms. The access link speed is 15 Mb/s and its link delay is randomly chosen between 0.5 ms to 2.0 ms so that the access link delay for TCP and TFRC would be different (to break the simulation phase effect [8]). The bandwidth-delay product (BDP) is about 3 packets[1] assuming that a packet size is 8000 bits (i.e., one packet = 1 KBytes). We ran 200 seconds simulation in total and draw the throughput graph as in Figure 3.2(a).

In this simulation, we had relatively low level of statistical multiplexing (4 TCP and 4 TFRC) with few bottleneck queue size[2] using drop-tail queue, and low link speed with relatively large link delay, which is most likely a DSL link. It showed that TFRC is too much aggressive (greedy) in finding the network bandwidth in that TCP sources are almost starved in terms of getting a useful throughput.

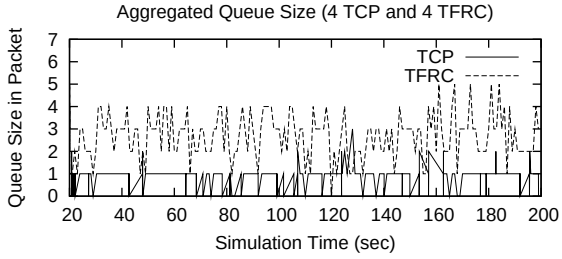
This is due to the fact that the decrease rate of TFRC's throughput is meant to be much slower than that of TCP. In other words, unlike to TCP, TFRC will not halve its sending rate even if it sees a packet drop, which leads the TFRC packets can easily occupy the bottleneck buffer space whereas the TCP packets will get dropped at the bottleneck node. When there is tiny buffer size at the bottleneck,

[1] So, the bottleneck queue size is little more than the BDP size.

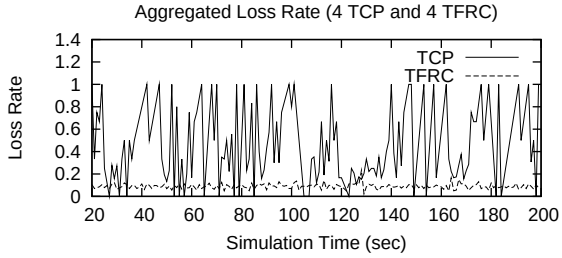
[2] The bottleneck queue size is set to about the delay-bandwidth product value. In this case, the BDP size is around 3 packets, and the bottleneck queue was set to 5 packets to make TCP work properly.



(a) Throughput



(b) Queue Dynamics



(c) Loss Rate

Figure 3.2: Unfairness of TFRC under a DSL-like Network

this behavior will make worse to the TCP's performance when it is competing with the same number of TFRC sources. This is shown in Figure 3.2(b) and Figure 3.2(c). As we have described, TCP packets can hardly occupy the queue space whereas TFRC packets mostly take the bottleneck queue space. This is also illustrated in the loss rate graph. Because the TCP packet cannot get into the queue, the loss rate often become to 1 which in turn results in zero throughput.

3.4 Summary of Motivation

It is certainly not desired if TFRC sources starve TCP sources under a DSL-like environment because a congestion control protocol should not do any harm between any sort of end-to-end network paths. The summary of our motivation is as follows.

- We bring the window-based congestion control instead of using the rate-based control. If we control the network congestion in terms of the sending rate, it is likely that it behaves more aggressively in finding the network resources. The motivation here is not to use the rate-base but to use the window-based control like to the traditional TCP.
- We re-introduce a TCP-like ACK mechanism for limiting the number of outstanding packets. This is an elegant way of controlling the outstanding packets in the network.
- Because of ACK mechanism, we can naturally give a delay loop to control the Internet congestion

opted to a real-time streaming application. In this way, the application performance would be least affected for a subtle change in the network.

- We still use the TCP throughput equation in order to calculate the number of available packets to send. In this way, we can keep the nice behavior that TFRC has.

The important motivation for our research is to achieve the protocol fairness for a real-time streaming application and to control the network congestion without harming the traditional TCP traffic. It can be still of great advantage if our window-based mechanism performs as equal as TFRC because TFWC is already cheaper and simpler to implement than TFRC. We will further investigate in the following chapters.

Chapter 4

TCP-Friendly Window-based Congestion Control

In this chapter, we give a full detail of the TCP-Friendly Window-based Congestion Control protocol (TFWC) in terms of how TFWC operates and the main concept involved in its functionality.

4.1 Introduction

TCP-Friendly Window-based Congestion Control (TFWC) shares a common fact with TFRC such that both protocols are using TCP throughput equation as in Equation (2.1). But, the main difference is one has the rate-based and the other has the window-based packet control. In other words, a TFRC sender computes the sending rate out of a feedback information from a TFRC receiver. If the computed rate is greater than the current sending rate, it increases the sending rate, otherwise decreases. However, a TFWC sender computes the congestion window (*cwnd*) size out of a feedback message from a TFWC receiver. If the computed *cwnd* allows to send more packets, the TFWC sender sends out as many packets as it is allowed, otherwise it waits for another Ack message from the receiver. Another important TFWC characteristic is that it does not retransmit the lost packet as it was designed for a real-time multimedia application. Without question, there is no point in sending the lost packet later for those applications if the retransmitted packet cannot be delivered within the desired end-to-end delay[1].

4.1.1 TCP Throughput Modelling

Before we go into the protocol details, we introduce a simplified version of Equation (2.1). This gives a more complex model of TCP throughput. For most applications it suffices to set t_{RTO} to a multiple of t_{RTT} and to have the fixed size packet. As we have described in section 2.2.3, t_{RTO} can be expressed as $t_{RTO} = 4 \times RTT$. Then, Equation (2.1) can be rewritten as the below.

$$T = \frac{s}{t_{RTT}\left(\sqrt{\frac{2p}{3}} + \left(12\sqrt{\frac{3p}{8}}\right) p\left(1 + 32p^2\right)\right)} \tag{4.1}$$

From Equation (4.1), we can derive the number of available packets to send at a sender. The idea is that if we multiply $\frac{t_{RTT}}{s}$ at the both side of Equation (4.1), then we get the number of bytes, which we can exploit the value to get the current *cwnd* size.

Let,

$$f(p) = \sqrt{\frac{2p}{3}} + \left(12\sqrt{\frac{3p}{8}}\right) p\left(1 + 32p^2\right), \tag{4.2}$$

[1] A typical real-time interactive end-to-end delay should be less than 150 ms. Therefore, the retransmitted packet (this includes the time for detecting the packet loss and the retransmission) cannot be delivered within this delay, there is no incentive of sending it again.

then, T can be expressed in

$$T = \frac{s}{t_{RTT} f(p)}. \tag{4.3}$$

Finally, we get by multiplying $\frac{t_{RTT}}{s}$ at the both side:

$$W = \frac{1}{f(p)}, \tag{4.4}$$

where W is the number of available bytes and p is the packet loss probability.

From Equation (4.4), we can see if we know the packet loss probability, then we can compute the number of available bytes to send at a sender. We brought the *average loss history* mechanism from TFRC to get the packet loss probability, and eventually will use that value to get the number of available bytes. This is the key part of TFWC protocol, and we give full details in the following sections.

4.2 The TFWC Protocol

4.2.1 Slow Start

The initial slow start procedure should be similar to the conventional TCP's slow start procedure where the sender roughly doubles its sending rate in each round-trip time. TFWC's slow start just uses this scheme. Because TCP's ACK clock mechanism provides a limit on the overshoot during slow start, no more than two outgoing packets can be generated for each acknowledged data packet, so TFWC cannot send at more than twice the bottleneck link bandwidth. Up to here (before getting the first lost packet), TFWC behaves exactly same as the conventional TCP does.

When a TFWC sender meets the first lost packet, then it stops slow start and calculate the packet loss probability. This calculated probability is not an actual packet loss probability but a derived probability using Equation (4.4). Because we know the *cwnd* size at the time of the first loss packet (but, we don't know the packet loss probability p yet), we can approximate the packet loss probability from Equation (4.4). Again, we approximate the packet loss probability because we intend to use the value for the average loss history mechanism. Note that the actual packet loss probability (measured packet loss probability) at this moment does not have much meaning as this is the first packet loss. Our intention is to get the packet loss probability using Equation 4.4 as the below.

```
1  On detecting the very first packet loss
2      cwnd = cwnd/2;
3      for pseudo = .00001 to 1 do
4          out = f(pseudo);
5          tempwin = 1/out;
6          if tempwin < cwnd then
7              break;
8          end
9      end
10     p = pseudo;
11 end
```

Algorithm 1: Approximate the packet loss probability

$f(p)$ in the above algorithm represents Equation (4.2). The above algorithm basically scans the p value starting from a very small number, and if the p value gives a larger window size than the current *cwnd*, then we take that p value as the closest approximated packet loss probability: reverse engineering to get the packet loss event rate (p). Once we obtain the approximated packet loss event rate, we can create a pseudo packet loss history, too. Again, this is not an actual packet loss interval, but we reverse-engineer it from the packet loss event rate from Algorithm 1. As [7] defined, we can get the packet loss history as:

$$\text{Packet Loss History} = \frac{1}{\text{Packet Loss Probability}}$$

Now, we obtained the pseudo packet loss probability and the pseudo packet loss history. From this point, TFWC sender does not go through the normal slow start phase, but will go through the main TFWC control loop which will be described in the next sections. The main purpose of having the pseudo loss probability and loss history is to use those value for calculating the average loss history.

4.2.2 ACK Vector

Like TCP and TFRC, TFWC also has an ACK message which carries feedback information from a receiver so that TFWC sender can adjust the window size. Unlike to TCP's ACK, the TFWC's ACK message doesn't have to be accumulative because TFWC does not retransmit the lost packet. What we have to do is to maintain the packet list whether it was successful or not. Therefore, at the receiver side, we build an ACK vector which consists of packet sequence numbers. This ACK vector will grow in size as the receiver gets the packets from the sender. If the sender confirms by sending another ACK of an ACK to the sender to say that it has understood the receiver has gotten the packet sequence number, then the receiver removes the packet number from the ACK vector. In other words, if the sender gets an ACK for a particular packet, then it sends an ACK of an ACK (*aoa*) to the receiver so that the receiver can remove that packet sequence number from the ACK vector.

Moreover, in order to follow the TCP's three duplicative ACK (*DupAck*) rule, we generate a *margin vector* from the ACK vector. Let say, if an ACK vector is (10 9 8 7 6 5 4), then the margin vector is going to be (9 8 7). This margin vector is generated from the head value of the ACK vector; the margin vector is the three consecutive packet sequence numbers from the head value of the ACK vector. The purpose of having the margin vector is the sender doesn't do anything even if a packet sequence number in the margin vector is missing from the ACK vector. For example, if an ACK vector is (10 9 7 6 5 4), then the margin vector would be (9 8 7). We can find that the packet sequence number 8 is missing in the ACK vector, but the sender does *not* think yet that the packet is lost because it is in the margin vector.

4.2.3 Sender Functionality

The TFWC sender functionality is critical for its whole protocol performance as it determines several important parameters. Before we go into much details, we give a big picture how it control the packets. When the sender gets an ACK message from the receiver, it first generates the margin vector as described in Section 4.2.2. Then, it checks the ACK vector whether there is any lost packet or not. If there is no packet loss, it will simply open *cwnd* like the normal AIMD scheme. If there is a packet loss and it is the very first packet loss event, then it halves the current *cwnd* and will get the pseudo packet loss probability and the pseudo packet loss interval as we have explained in Algorithm 1. Once TFWC sender detected the lost packet for the first time and got the pseudo probability/history, TFWC will switch to the following algorithm for computing the available *cwnd* size.

Upon receiving an ACK message, the sender will check the ACK vector to see if there is any lost packet. After that, it updates the loss history information and calculate the average loss interval. From the average loss interval, we could get the packet loss event rate. Using Equation (4.4), it gets the new *cwnd* size. If a packet sequence number of a newly available data is less than the addition of the unacknowledged packet sequence number and the new *cwnd*, then the sender will be able to send the new data packet. This is shown in Figure 4.1. In other words, as long as the sequence number of the new data meets Equation (4.5), then the TFWC sender can send more data.

$$\text{Packet Seqno of New Data} \leq \textit{cwnd} + \text{Unacked Packet Seqno} \tag{4.5}$$

As we have discussed, in order to calculate the *cwnd* size, we use the average loss history mechanism. So, the average loss history update mechanisms is one of the key parts for TFWC to get it work right. Every time the TFWC sender sees a lost packet, it will compare the current time-stamp with t_{RTT}. If the time difference is greater than t_{RTT}, then this loss will start a new loss event. If the time difference

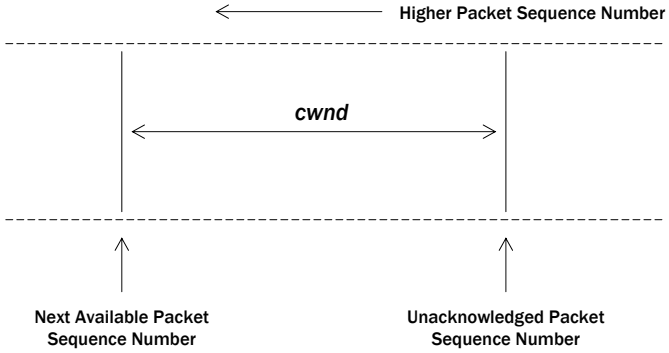


Figure 4.1: TFWC *cwnd* mechanism

is less than t_{RTT}, we do nothing but increase the average loss history by one. If there is no packet loss, it increases the current loss history by one. Once it gets the average loss history information, it calculates the average loss interval which will eventually determine the packet loss probability.

We take the *Exponentially-Weighted Moving Average* (EWMA) method to build the loss history. This method takes a weighted average of the last n intervals, with equal weights for the most recent $n/2$ intervals and smaller weights for older intervals.

To summarize, the TFWC sender will work as in Figure 4.2. We did not cover any of the timer issues in this section. But as we have discussed already, the improper timer management can be drastically influence on the total TFWC performance, so we have to pay attention to deal with it. We will have a separate section on the timer issues later.

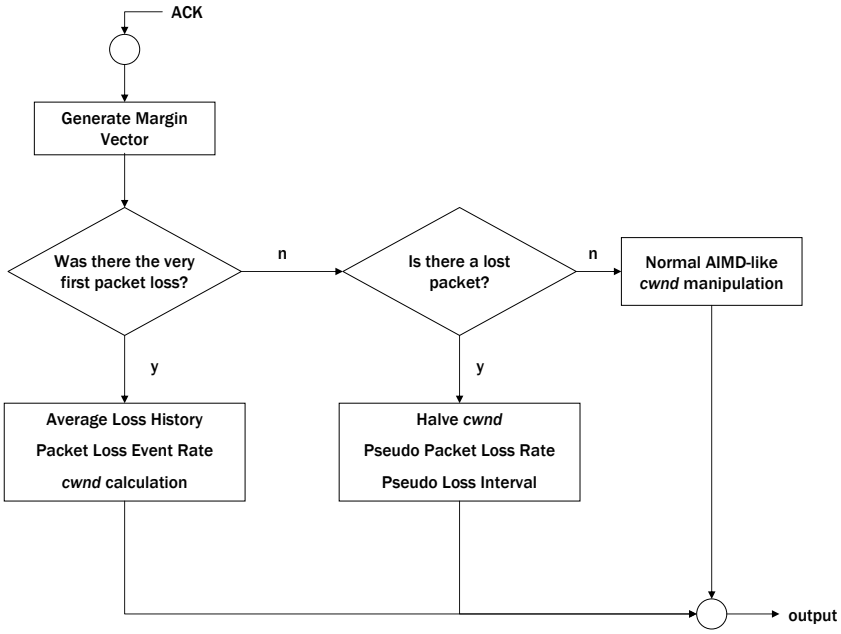


Figure 4.2: TFWC sender functions

4.2.4 Receiver Functionality

The receiver provides feedback to allow sender to measure round-trip time (t_{RTT}). The receiver also appends the packet sequence number in the ACK vector. When it gets an ACK of an ACK from the sender, it removes that packet sequence number from the ACK vector. The complex part of the TFWC protocol is at the sender side, so the TFWC receiver just builds an ACK vector as described above.

4.2.5 TFWC Timer

In order to use Equation (4.4), the sender has to determine the values for the round-trip time (t_{RTT}) and retransmit timeout value (t_{RTO}). The sender and receiver together use time-stamps for measuring the round-trip time. Every time the receiver sends feedback, it echoes the time-stamp from the most recent data packet.

The sender smoothes the measured round-trip time using an exponentially weighted moving average. This weight determines the responsiveness of the transmission rate to changes in round-trip time. The sender could derive the retransmit timeout value t_{RTO} using the usual TCP formula as in Equation (2.2).

Every time a feedback message is received, the sender calculates the new values for the smoothed round-trip time and retransmit timeout value. If timeout occurs, then the sender backs off the current retransmit timeout value and resets the timer with it. After that, the sender will artificially inflate the latest ACK number by one so that it can send out a new data packet.

4.2.6 Summary

In this section, we have explained the key function on the TFWC, and the basic protocol mechanisms. However, there are a few features that we might want to cover in more detail as we explain the protocol implementation issues. We will specifically deal with those matter in the next chapter.

Chapter 5

Implementation

This chapter describes the implementation of TFWC protocol. During this phase, many variants of the required algorithms have been tested and their benefits were evaluated. The most significant problems encountered during the implementation and tuning of the protocol are described in this chapter. The implementation process is not only involved the TFWC specification but also involved in creating more concrete specification.

5.1 Overview

The protocol implementation was developed for the *ns-2* [14] network simulator. There are three main components that has been developed:

1. **TFWC** Agent – This is the TFWC sender.
2. **TFWC Sink** Agent – This is the TFWC receiver.
3. **TFWC ACK Vector** – This manipulates the TFWC ACK vector.

The TFWC Agent is mainly responsible for maintaining the average loss history and calculating the loss event rate. Apart from this, it also updates the various timers such as the round-trip time (t_{RTT}) and the retransmit timeout value (t_{RTO}). Finally, the TFWC Agent adjusts the congestion window size (*cwnd*) based on Equation (4.4).

The Sink Agent is the simplest component of all and will not be described in detail. It simply echoes the time-stamp in its header, and it adds/removes the ACK vector according to the packet sequence number that it has received. This is well enough explained in Section 4.2.4.

The TFWC ACK vector is a singly linked list object. Like a usual linked list, the TFWC ACK vector can be built from the head and tail, and can be deleted in either direction, too. Along with the insert/delete function, it has search and copy function. The implementation of the TFWC ACK vector is quite similar to a usual linked list implementation, we will not cover the details in this section. Throughout this chapter, we only focus on describing the implementation of the TFWC Agent.

5.2 TFWC Main Reception Path

From the functionality point of view, the TFWC Agent will perform the following tasks.

- Normal AIMD-like *cwnd* Control
- TFWC *cwnd* Control
- Timer Update
- Send Packets

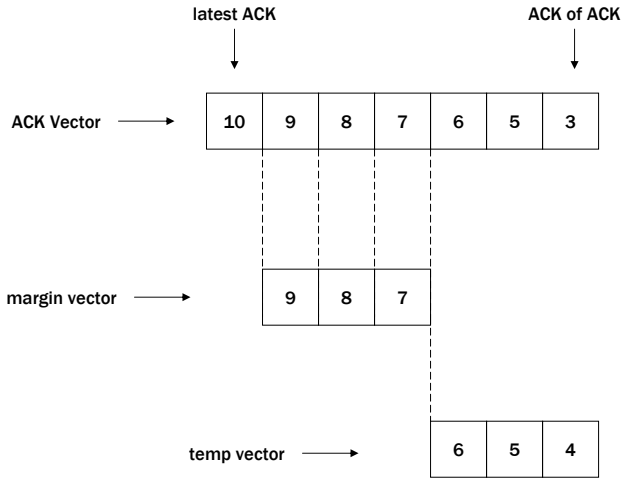


Figure 5.1: TFWC ACK vector, margin vector, and tempvec

We already mentioned about the normal AIMD *cwnd* control mechanism in Chapter 4.2.1. Although the normal AIMD mechanism is trivial, the TFWC *cwnd* control mechanisms involves in many steps and calculation. The TFWC message header contains a packet sequence number, an ACK of an ACK, and a time-stamp. The TFWC ACK header contains the time-stamp echo, and the TFWC ACK vector.

When a new ACK comes in, the sender first stores the head value of the ACK vector (we call it as the latest ACK number), and generates the margin vector using the latest ACK number. This is explained in Section 4.2.2. Then it finds out whether there is a lost packet or not by examining the ACK vector. In order to do that, it first creates a vector sequence ranging from the ACK of ACK to the last element of the margin vector: we call this vector as *tempvec* through the rest of this paper. For example, assume that we have the ACK vector as (10 9 8 7 6 5 3) and the current ACK of ACK is 3, then the marginvec is (9 8 7) and the *tempvec* is (6 5 4). We show how these vectors are generated in Figure 5.1.

From Figure 5.1, we can notice that the packet sequence number 4 is missing (lost) if we compare with the *tempvec* and the actual ACK vector. Upon the very first lost packet event, we halve the *cwnd* and calculate the pseudo packet loss rate and the pseudo packet loss interval. However, until we see the very first lost packet event, we simply follow the normal AIMD-like *cwnd* control. In this chapter, we assume there was the first packet loss already, and every incoming ACK packet is now going to the main TFWC control procedures. In the following sections, we describe the main TFWC control procedures: loss history counting and *cwnd* control.

5.3 Average Loss History

5.3.1 Overview

Before we get into the implementation details, we briefly give an overview of the average loss interval. A TFWC sender maintains the average loss history information. The history information is an n-ary array. Each bucket in the array has the number of packets between loss events; we call this number as the loss interval. After all, the loss history information is the array where the loss interval is stored. Let l_i be the number of packets in the i^{th} interval, and the most recent loss interval l_0 be defined as the interval containing the packets that have arrived since the last packet loss. When a new loss event occurs, the loss interval l_0 now becomes l_1, all of the following loss intervals are shifted down one accordingly, and the new l_0 is initialized. We will ignore l_0 in calculating the average loss interval if it is not enough to change the average. This will allow to track smoothly where the loss event rate is relatively stable. In addition, we give weights depending upon the history information. In other words, we take a weighted average of the last n intervals, with equal weights for the most recent $n/2$ intervals and exponentially smaller weights for older intervals. Thus the average loss interval $\hat{l}$ is computed as follows:

$$\hat{l} = \frac{\sum_{i=1}^{n} w_i l_i}{\sum_{i=1}^{n} w_i}, \tag{5.1}$$

for weights w_i:

$$w_i = 1, \quad 1 \leq i \leq n/2,$$

and

$$w_i = 1 - \frac{1 - n/2}{n/2 + 1}, \quad n/2 < i \leq n.$$

In our implementation, we choose $n = 8$ which gives weights of: $w_1, w_2, w_3, w_4 = 1; w_5 = 0.8; w_6 = 0.6; w_7 = 0.4; w_8 = 0.2$. The average loss interval method that we took also calculates $\hat{l}_{new}$, which takes the intervals from 0 to $n - 1$ instead of choosing 1 to n.

Thus, $\hat{l}_{new}$ can be defined as follows:

$$\hat{l}_{new} = \frac{\sum_{i=0}^{n-1} w_{i+1} l_i}{\sum_{i=1}^{n} w_i}. \tag{5.2}$$

Finally, we determine the average loss interval as:

$$max(\hat{l}, \quad \hat{l}_{new}) \tag{5.3}$$

In this way, we could reduces the sudden changes in the calculated loss rate that could result from unrepresentative loss intervals leaving the set of loss intervals used to calculate the loss rate.

5.3.2 Implementation of Average Loss History

As we can see in Figure 4.2, upon an ACK packet arrival at the sender, it will call the average loss history calculation procedures: assume we had the very first lost packet already. Whenever an ACK message is coming to the sender, it will first check whether there is a lost packet packet by examining the ACK vector, and then it will calculate the average loss history. If there is no packet loss, then it will increase the current loss interval by one. Every time the sender sees a lost packet in the ACK vector, it will compute the time difference between the time-stamp for the last lost packet and the time-stamp for the current lost packet, and compare it with the round-trip time t_{RTT}. If the time difference is greater than t_{RTT}, then this loss will start a new loss event. If the time difference is less than t_{RTT}, then we just increases the current loss interval by one. This is described in Algorithm 2.

In Algorithm 2, numVec stands for the number of element in tempvec. In this way, we build up the average loss history information. Based on the history, we can compute the average loss rate. We have implemented this part as described in Section 5.3.1. In the following section, we give implementation details on the average loss rate calculation.

5.3.3 Implementation of Average Loss Event Rate

As we have discussed in Section 5.3.1, we compute $\hat{l}$ and $\hat{l}_{new}$, and then take the larger value as in Equation (5.3). In our implementation, we have 8 elements in the history array except the current loss information. That is we have 9 elements in the history array in total. This is depicted in Figure 5.2.

As in Equation (5.1) and Equation (5.2), the first eight elements are involved in calculating $\hat{l}$ and the last eight elements are involved in calculating $\hat{l_{new}}$. After computing $\hat{l}$ and $\hat{l_{new}}$, we compare those values and take the larger one as the average loss interval. We use the average loss interval to calculate the loss event rate as follows:

$$\text{Loss Event Rate } (p) \quad = \quad \frac{1}{\text{Average Loss Interval}} \tag{5.4}$$

In the following section, we describe how this value is used to get the current *cwnd* size.

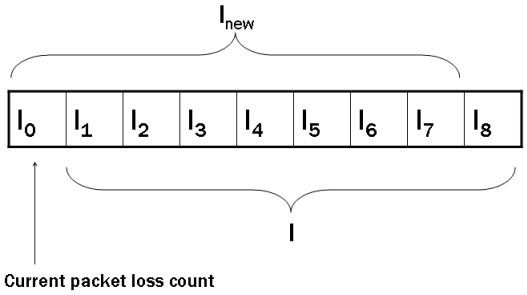


Figure 5.2: Loss History Array

5.4 TFWC *cwnd* Control

We assume that there was already a lost packet, and every incoming ACK message is processed by the TFWC *cwnd* control procedures. The TFWC *cwnd* control procedure is basically calculating the new *cwnd* value using the TCP response function as in Equation (4.4). We use the loss event rate p from Equation (5.4). The TFWC *cwnd* control mechanism works in the following way. Note that we change the *cwnd* controlling method when the calculated window value is less than 2. This is because the window-based *cwnd* control can cause the application's throughput going to almost zero for a certain period of time. We will describe this more in detail in the following section.

```
1  Upon an ACK vector arrival at the sender
2     for i = 0 to NumVec do
          /* search AckVec to find any missing packet                    */
3         if AckVecSearch(tempvec[i]) then
4            isGap = true;
5         else
6            isGap = false;
          /* compare timestamp values                                    */
7         if timestamp(tempvec[i]) - timestamp(last loss) > t_RTT then
8            isNewEvent = true;
9         else
10           isNewEvent = false;
          /* determine if this is a new loss event or not                */
11        if isGap and isNewEvent then
             /* this is a new loss event                                 */
12           shift history information;
13           record lost packet's timestamp;
14           history [0]=1;
15        else
             /* this is not a new loss event                             */
             /* thus just increase current history information            */
16           history [0]++;
17    end
18 end
```

Algorithm 2: TFWC Average Loss Interval Calculation

```
   input: Average Loss Interval (avgInterval)
1  Upon an ACK vector arrival
2      p = 1/avgInterval;
3      out = f(p);
4      floatWin = 1/out;
       /* convert floating point window to an integer                    */
5      cwnd = (int)(floatWin + .5);
       /* if window is less than 2.0,                                    */
       /* TFWC is driven by rate-based timer mode                        */
6      if floatWin < 2.0 then
7          isRateDriven = true;
8      else
9          isRateDriven = false;
10 end
```

Algorithm 3: TFWC *cwnd* Procedures

5.4.1 Extended TFWC *cwnd* control

The TFWC's window-based *cwnd* control can sometimes cause the throughput being almost zero when the loss rate is large. For example, if the loss event rate is large enough to make the outcome of $f(p)$ be greater than 0.5, then the calculated *cwnd* will be less than 2, which in turn the other type of application packets can occupy the bottleneck queue space more easily. In this case, TFWC traffic may be starved for a certain amount of period.

Loosing throughput all of a sudden can degrade the application's overall performance, hence, it is not what we desired. It could be an expected behavior to shut down the application's sending window under a severe network congestion, but we want to keep shooting the packets at least rather than cutting down the sending rate completely. The solution is to switch back to the rate-based mode if this situation occurs. That is we switch back to the rate-based timer mode when the calculated *cwnd* is less than 2. The reasoning behind this idea is we have seen that the rate-based mode can react more aggressively when there is a small buffer space with relatively low link speed. Equation (2.1) basically means the number of available bytes to send in seconds. Therefore, if we take the denominator as the retransmit timeout value (t_{RTO}), then we could adopt the rate-based mode inside TFWC. This means if a TFWC sender does not send any packet during t_{RTO}, the sender calls the timer-out procedure and artificially send the next available packet. In other words, when the calculated *cwnd* is greater than 2, t_{RTO} can be computed by Equation (2.2). If the calculated *cwnd* is less than 2, then t_{RTO} will be computed by Equation (5.5); it is the denominator of Equation (2.1).

$$t_{RTO} = t_{RTT}\sqrt{\frac{2}{3}p} + t_0(3\sqrt{\frac{3}{8}p})p(1+32p^2), \tag{5.5}$$

where

$$t_0 \quad = \quad SRTT + 4 * RTT_{var}, \tag{5.6}$$

and

$$SRTT \quad = \quad (t_{RTT} - SRTT)/8 + SRTT, \tag{5.7}$$

and

$$RTT_{var} \quad = \quad \mid (t_{RTT} - SRTT)/4 \mid + RTT_{var}. \tag{5.8}$$

We did a simple simulation to show how it works on *ns-2* network simulator. There are one TCP, one TFRC, and one TFWC traffic over a dumbbell topology as in Figure 5.3. The bottleneck bandwidth is 1.5 Mbp/s and the bottleneck queue is drop-tail with 5 packets. This is shown in Figure 5.3.

We ran 500 ms of simulation before we implement the extended *cwnd* control (rate-based timer mode), and after we put it. We can see some of the transient period for TFWC application which shows poor throughput performance roughly between 160 seconds to 180 seconds in Figure 5.4(a). However, if we

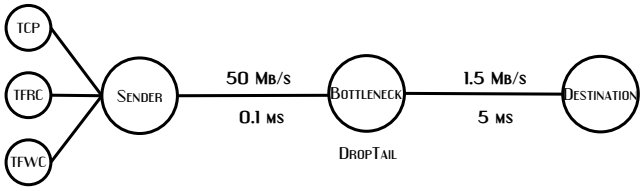
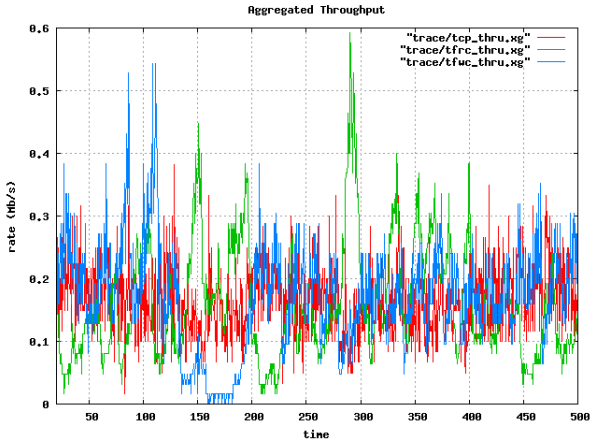
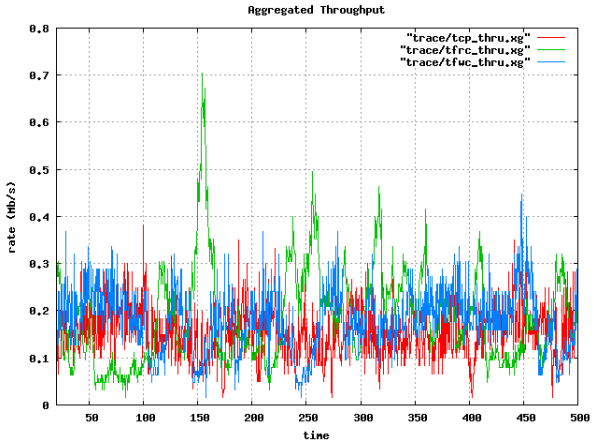


Figure 5.3: Simulation Topology for an Extend TFWC *cwnd* Control

implement the rate-based timer mode, we can see this bad performance can be notably removed as in Figure 5.4(b).



(a) without rate-based timer mode



(b) with rate-based timer mode

Figure 5.4: Comparison for the Extended TFWC *cwnd* Control

In summary, only when the calculated *cwnd* goes below 2, we switch back to the rate-based timer mode so that the TFWC sender can call the timer-out procedure more often to artificially send the next available packet. Consequently, the main reception path should be changed as in Figure 5.5 instead of Figure 4.2. In this figure, we added the timer update and the rate-based mode depending upon the calculated *cwnd* value.

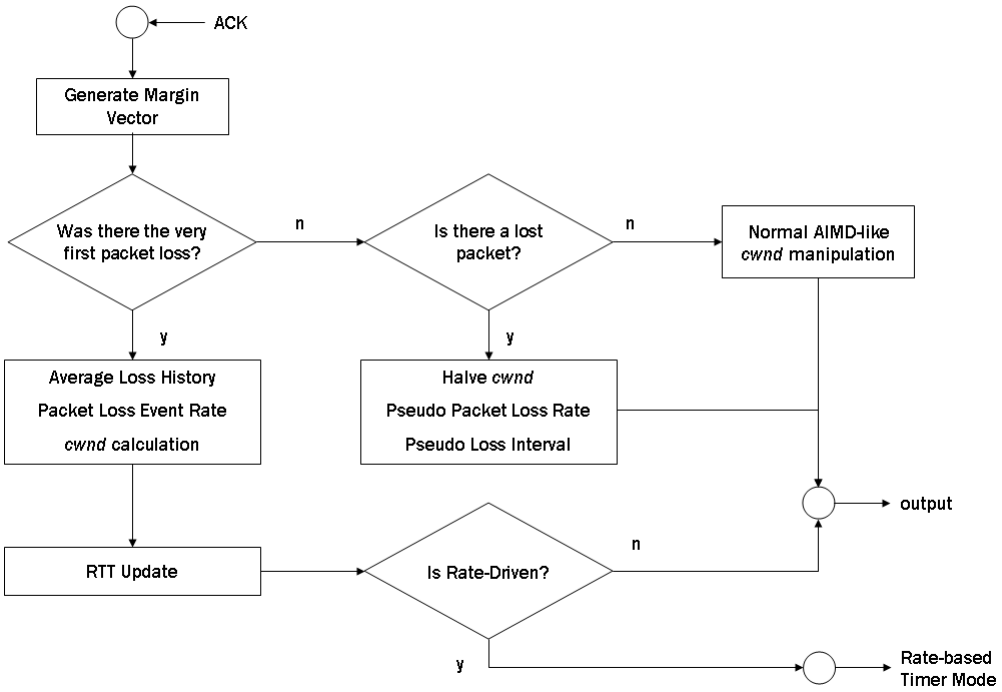


Figure 5.5: Extended TFWC Sender Functions

5.5 TFWC Timer

Every time an ACK message comes into the TFWC sender, it will update the round-trip time (t_{RTT})and the retransmit timeout (t_{RTO}) value. With the calculated t_{RTO}, it sets the retransmission timer every time it sends out a new packet. If the retransmission timer expires, it backs off the the timer by setting t_{RTO} double, and sets the retransmission timer again. Finally, it artificially inflates the last ACK number (this ACK is still unacknowledged) in order to send the next packet sequence number. Note if it is under the rate-based timer mode (i.e., the *cwnd* is less than 2), then we do not back off the timer but reset the timer with t_{RTO} calculated from Equation (5.5). This is shown in Algorithm 4. By the way, TFWC's the round-trip time update follows the conventional TCP and TFRC update mechanisms.

```
1  On the TFWC timer expires
2      if isRateDriven = true then
3          setRtxTimer(rto);
4      else
5          backoffTimer();
6          setRtxTimer(rto);
7      if isRateDriven = false then
8          lastACK++;
       /* call send method                                          */
9      send(seqno);
10 end
```

Algorithm 4: TFWC Timer Functions

Chapter 6

Evaluation

This chapter describes the detailed results on the protocol's performance evaluation using *ns-2* [14] simulation. We first verify/validate the protocol's functionality and then we compare the protocol's performance with TCP and TFRC.

6.1 Protocol Validation

In this section, we validate the protocol's functionality in three-fold: ACK vector, average loss history mechanism, and *cwnd* mechanism. The simulation topology for the validation process is composed of one bottleneck node and a TFWC source and a destination node. We assume that the application traffic is always available, which means a packet is always ready to be sent out from the source. The simulation topology is shown in Figure 6.1. Because we want to test the protocol's functionality, we set the bottleneck link speed and the bottleneck queue size is reasonably high enough so that those parameters don't affect the test results.[1] The purpose of the protocol validation is to make sure that the TFWC is behaving correctly as we have designed before we have the detailed comparison work.

6.1.1 ACK Vector

The purpose of this test is to verify whether the TFWC ACK vector is reacting correctly in case of a packet loss. We would like to see that the sequence number of the lost packet is not showing up in the received ACK vector so that the sender can detect the packet loss. We artificially drop a specific packet sequence number and see if it is reflected in the received ACK vector. Also, we drop a series of packets in the same window and check the ACK vector. Finally, we check the ACK of ACK functionality. This is itemized as the below.

- ACK vector build-up process
- ACK vector update mechanism
- ACK of ACK

In order to check the ACK vector functionality, we can directly analyze the *ns-2* output trace file generated at the bottleneck node. However, this process is inefficient such that we will have to explicitly drop a certain packet number and check it at the output trace file. Although we can still do this way, there is a more elegant way to validate the ACK vector functionality. The other way is to check ALI. This is efficient in that if we can prove ALI works correctly, then we can say the ACK vector functionality and ALI mechanism both are working correctly. If the ACK vector mechanism is not properly working, then ALI will not work correctly accordingly. In the following section, we validate the Average Loss History mechanism.

[1] We want to validate protocol's functionality in a network condition where there is no *arbitrary* packet loss occurred by network parameters (e.g., bottleneck queue size, BDP, etc). However, in order to test ALI and *cwnd* mechanisms, we manipulate the packet loss explicitly as we wanted (this is a constant packet loss). Having the explicit and *constant* packet loss rate control, we know what we can expect to get ALI values accordingly. Practically speaking, for example, if we want to set $p = 0.01$, then we drop one packet intentionally out of a hundred packet transmission.

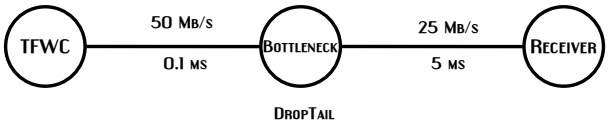


Figure 6.1: Simulation Topology for the Protocol Validation

6.1.2 Average Loss History

Case 1: Constant Packet Loss Case

The Average Loss History (ALI) mechanism is one of the core parts to make TFWC working correctly. For example, if we have a constant packet loss event rate, then it should be reflected to ALI accordingly. Suppose we drop the very first packet every time out of one hundred packet transmission periodically, then the loss event rate generated from ALI should indicate 0.01. In this test, we artificially drop the very first packet periodically out of 100 packet transmission, out of 20 packet transmission, and out of 10 packet transmission, respectively. The results are shown in Figure 6.2.

As we can see in Figure 6.2, ALI is quickly adjusted to what we can expect. These graphs show that ALI is 100, 20, and 10 according to the loss event rate of 0.1, 0.05, and 0.1, respectively. We could say that ALI mechanism works correctly under a constant loss event rate with a fixed-location packet drop (first packet drop among such packet transmission). Now, we want to check more complicated cases in the following subsections.

Case 2: Multiple Packet Losses in a Single Window

In this subsection, we look into some cases what happens if there are multiple packet losses in a same window. According to Algorithm 2, we should not start a new loss event if multiple losses occur in the same window. That is it is expected that the latest loss history information is just increased by one. We would like to extend our case as follows:

- Two back to back losses in a same window, Figure 6.3(a)
- One packet loss after one successful packet transmission, Figure 6.3(b)

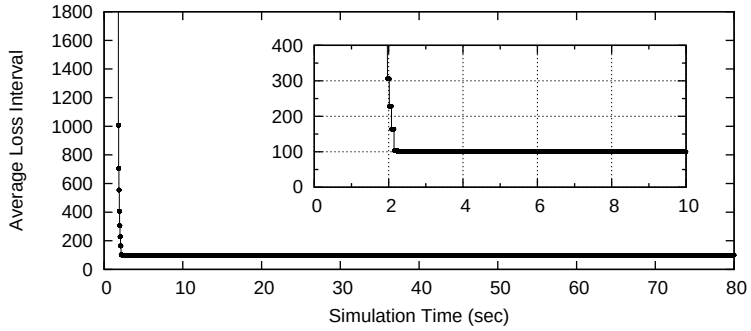
For the back-to-back packet loss case, we artificially set the *cwnd* size equal to 100, which means we intentionally drop the back-to-back packets out of 100 packet transmission, and check the ALI dynamics whether it indicates 100. As for the bi-packet loss case, we drop a packet right after one successful packet transmission.

Figure 6.4(a) shows that the back-to-back packet losses in the same window (*cwnd* size is 100) has the identical outcome with the constant loss rate ($p = 0.01$) case as in Figure 6.2(a). According to Algorithm 2, we simply increase the current history information by one if the packet loss doesn't start a new loss event. The back-to-back packet loss in the same widow falls in this category, thus it increases the current loss history information. Because of this reason, we get the same ALI value with the back-to-back packet loss case and one packet loss case among one hundred packet transmission.[2] In terms of the *loss event rate* derived from ALI, these two case will have same value . Therefore, Figure 6.4(a) is just the expected result and we can confirm that the ALI works correctly where multiple packet losses in a single window. Recall that we compute ALI as in Equation (5.4), we expect to have ALI value as 2 if a packet is lost one another, meaning that ALI should indicate 2 in this bi-packet loss case. After all, the loss event rate will become 0.5 in this case. Figure 6.4(b) shows that it behaved as we expect.

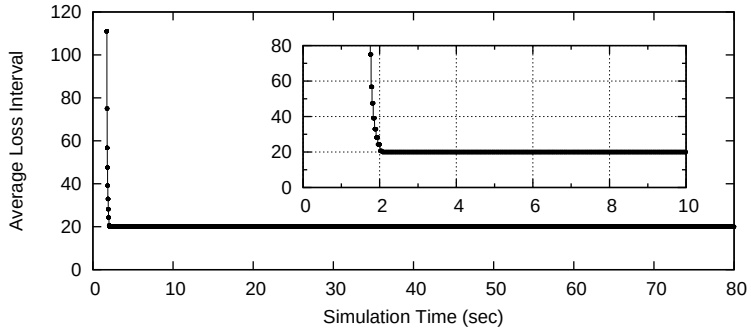
Case 3: Constant Packet Loss with Random Packet Drop

In this subsection, we will have a constant packet loss event rate but drop a packet randomly in the window like Case 1. For example, in case of $p = 0.01$, we drop a packet *randomly* out of hundred packets in the same window.

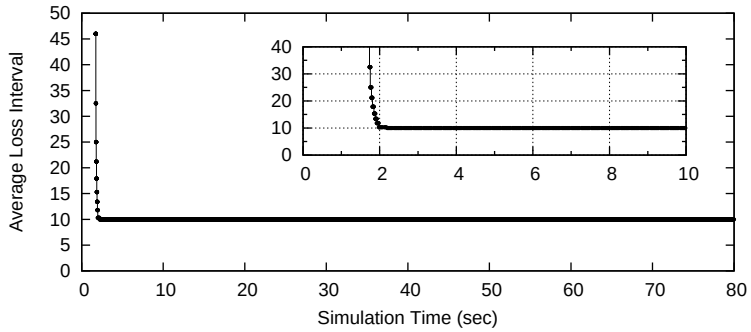
[2]The *actual* packet loss probability will be 0.02 for the former case and 0.01 for the latter case.



(a) Loss Event Rate = 0.01



(b) Loss Event Rate = 0.05



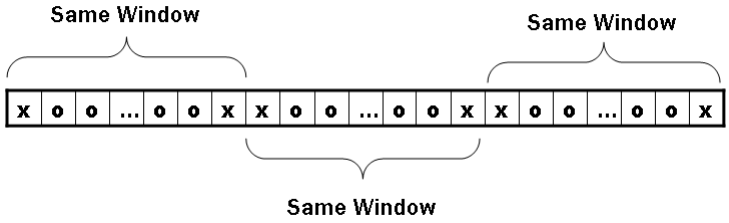
(c) Loss Event Rate = 0.1

Figure 6.2: TFWC ALI Validation

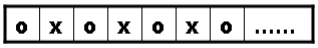
As we can see in Figure 6.5, we see the results are identical to the Case 1. This will verify that ALI mechanism works fine under the random packet loss with a constant loss event rate. In summary, we have dealt various cases to check the ALI functionalities. In the following section, we will check whether *cwnd* dynamics under the same conditions that we tested in this section. In theory, if ALI doesn't work, then we can't really say that *cwnd* works fine. In other words, if ALI mechanism works as we have designed, then we could possibly say that the TFWC's *cwnd* control mechanism works fine, too. We will have the validation test in the following section.

6.1.3 *cwnd* Validation

In this section, we will have the validation test for the TFWC's *cwnd* mechanisms. Because *cwnd* is determined by the ALI value with Equation (4.4), we first validated the ALI functionality. Assuming the ALI mechanisms work correctly, calculating *cwnd* value only involves applying the TCP throughput equation. From Equation (4.4), we could roughly say that $cwnd = \frac{1}{\sqrt{p}}$. For example, if the loss event

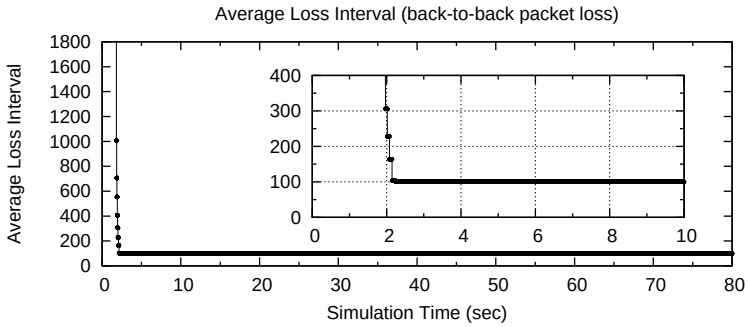


(a) Two Back-to-back Packet Loss

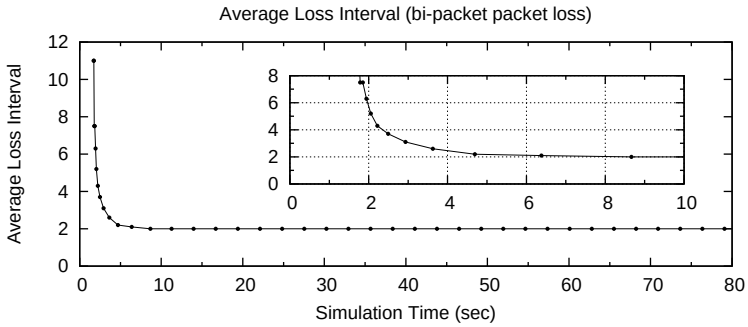


(b) Bi-Packet Loss

Figure 6.3: ALI Test with Various Packet Loss Pattern



(a) Back-to-back Packet Loss



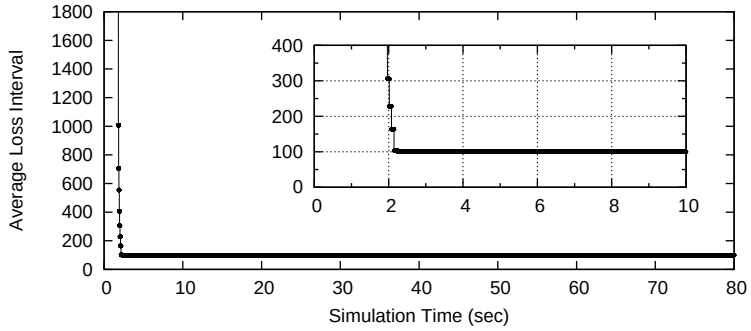
(b) Bi-Packet Loss

Figure 6.4: Additional ALI Validation Test

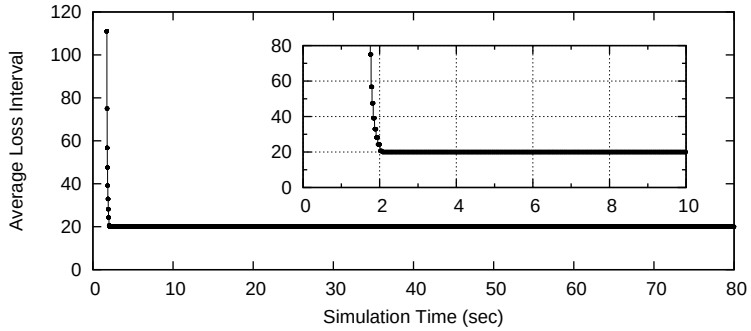
rate is 0.01, then *cwnd* is going to be 10 roughly.[3] Note there may occur *cwnd* value out by one packet. This is due to we initialize the first loss history value as 1 instead of 0. Suppose we have the case as in Figure 6.3(b). If we initialize the history value to 0 when a new loss event starts, then the average loss interval in this case would be 1, which results in the loss event rate to 1 instead of 1/2.[4] This is nonsense

[3]The dominant term from Equation (4.2) is $\sqrt{\frac{2p}{3}}$. Then, using Equation (4.4), it results in $\mathsf{W} \cong \frac{1}{\sqrt{\frac{2p}{3}}} = \sqrt{\frac{3}{2}}\frac{1}{\sqrt{p}}$. Thus, TFWC's *cwnd* (W) will be roughly $\frac{1}{\sqrt{p}} < \mathsf{W} < \frac{1.225}{\sqrt{p}}$. For example, when $p = 0.05$, we get $f(0.05) \cong 0.271$ using Equation (4.2). So, then, $\mathsf{W} = 1/f(0.05) = 1/0.271 \cong 3.7$, which reflects our results as in Figure 6.6(b). We have rounded up this value to the nearest integer because W can only be an integer value.

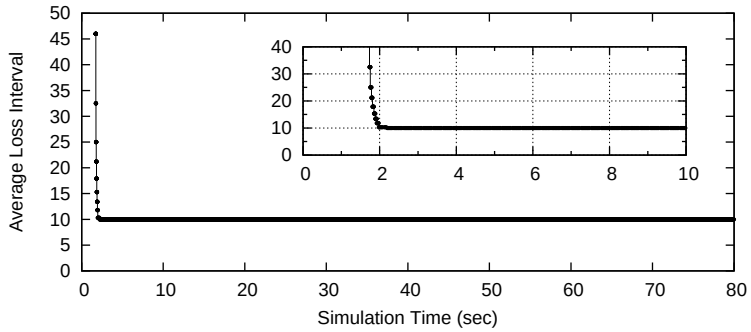
[4]The fact that the loss event rate becomes to 1 means that W will become 1. However, we expect that W converges to 2 in the bi-packet loss case.



(a) Loss Event Rate = 0.01



(b) Loss Event Rate = 0.05

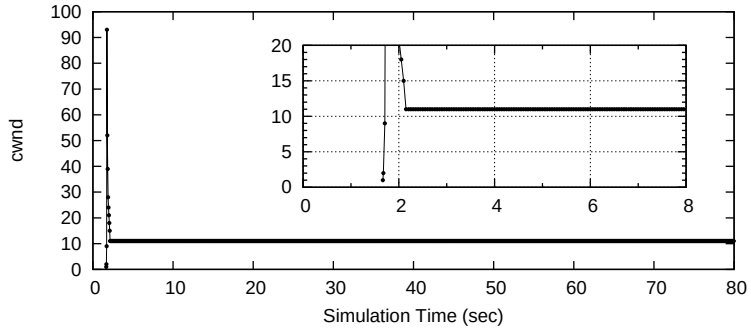


(c) Loss Event Rate = 0.1

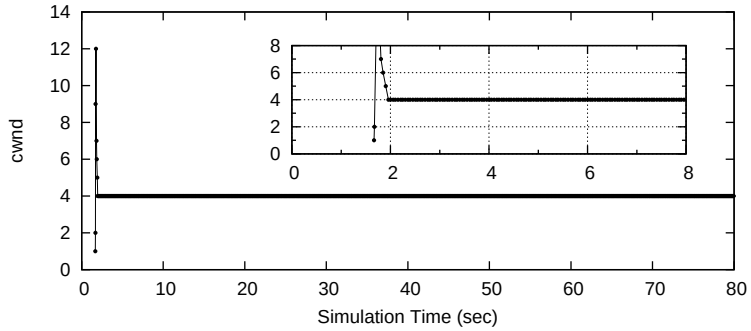
Figure 6.5: TFWC ALI Validation: random packet drop in the same window

as the loss event rate in this case should be 1/2. For this initialization problem, we may get *cwnd* value out by a packet for the other cases. However, this is not a problem from the protocol's performance point of view as we will round up or down the value anyway depending upon the calculated *cwnd* value (floating point): *cwnd* will be chosen to the nearest integer value given by Equation (4.4). We conducted the same simulation at Section 6.1.2. The results are shown in Figure 6.6

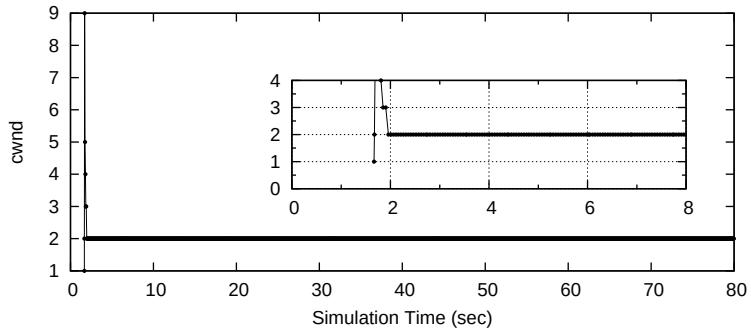
Figure 6.6(a) shows *cwnd* is stabilized to 11. In this case, the loss event rate is 0.01, and this will give $cwnd \cong 10$ using $cwnd \cong 1/\sqrt{p}$ rule. The next case, $cwnd \cong 4.47$ using the same rule with $p = 0.05$, and $cwnd \cong 3.16$ with $p = 0.1$. These are all in the error range that we can tolerate. In a longer term, this would not affect the overall protocol performance. In summary, we have validated that the every piece of TFWC works correctly in Section 6.1. Now, we will move on the protocol's performance evaluation in a various way.



(a) Loss Event Rate = 0.01



(b) Loss Event Rate = 0.05



(c) Loss Event Rate = 0.1

Figure 6.6: TFWC *cwnd* Validation

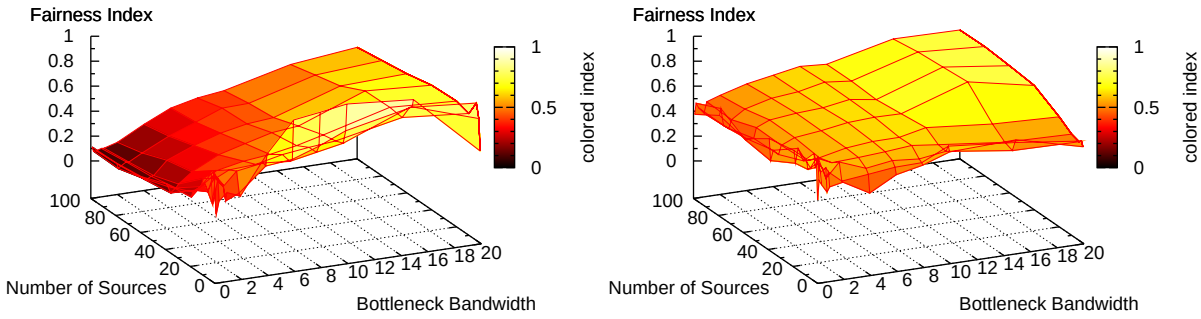
6.2 Performance Evaluation

This section presents the results of the TFWC protocol performance comparing to TCP and TFRC under various network environment. We will primarily focus on the protocol's fairness issues with the drop-tail queue at the bottleneck node. We will also deal with Random-Early Detection (RED) [9] scheme at the bottleneck node. Finally, we will present a result regarding the protocol sensitivity depending upon a queue parameter for RED queue case.

6.2.1 Fairness

In this section, we use simulation topology as in Figure 6.1. The objective of this simulation study is to have a clear idea on how much TFWC is fairer comparing to TFRC when it is competing with the same number of TCP sources[5]. We used the simulation topology as described in Figure 6.1, but we vary the network parameters. First of all, the number of sources vary from 1 to 100 per simulation and used the

[5]In *ns-2* simulation, we have used TCP/Sack1 Agent.



(a) Fairness Comparison (TCP and TFRC)

(b) Fairness Comparison (TCP and TFWC)

Figure 6.7: Protocol Fairness Comparison using Drop-Tail queue where $t_{RTT} \cong 22.0$ ms

same number of competing TCP and TFRC sources and similarly competing TCP and TFWC sources. The bottleneck bandwidth varies from 0.1 Mb/s to 20 Mb/s with a 10ms link delay. The access link delay was chosen randomly between 0.2 ms to 2.0 ms. There are reverse path TCP traffic in the simulation in order to break the simulation phase effect and exacerbate any issues due to TFWC Ack-compression. We used the drop-tail queue at the bottleneck and its size was set to a BDP to allow TCP to perform properly. In this scenario, the approximate average t_{RTT} is roughly 22 msec - a fairly typical DSL t_{RTT} to the local ISPs. We draw the protocol fairness graphs as follows. Let the protocol fairness be θ, then the fairness indicator can be defined as:

$$\text{Fairness Index } (\theta) = \frac{\sum_{i=1}^{n} T_{tcp_i}}{\sum_{i=1}^{n} T_{tcp_i} + \sum_{j=1}^{n} T_j}, \tag{6.1}$$

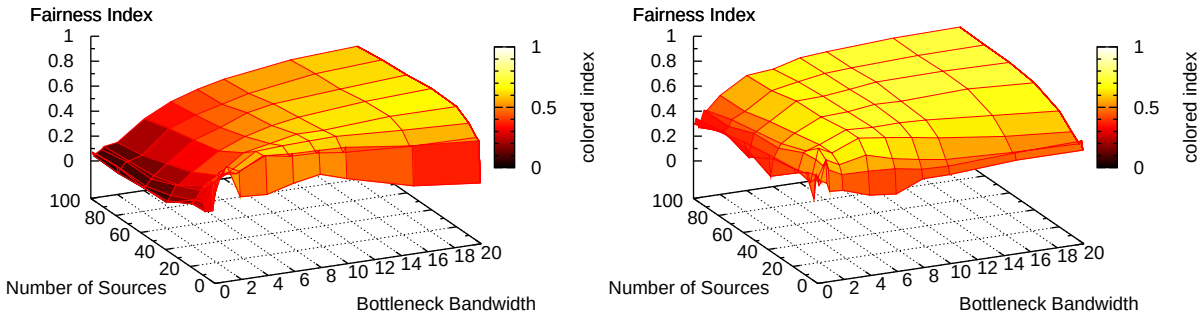
where T_{tcp_i} is the throughput of the i^{th} TCP source, and T_j is the throughput of the j^{th} TFRC (or TFWC) source. If θ is 0.5, then it indicates TCP perfectly shares the link with TFRC (or TFWC). As the *fairness index* (θ) goes to 1, it means that TFRC (or TFWC) sources are starved by the TCP sources. In other words, if θ goes close to 0, it means that TFRC (or TFWC) sources are dominant in the network. Figure 6.8(a) and Figure 6.8(b) show the protocol fairness for TCP with TFRC and TCP with TFWC, respectively.

The x-axis shows the bottleneck bandwidth, and the y-axis gives the number of flows sharing the bottleneck. The z-axis stands for the fairness index (θ). The TFRC protocol fairness is largely dependent upon the bottleneck bandwidth. If the bottleneck bandwidth is less than 1 Mb/s, then we can hardly say the TFRC is fair with TCP traffic no matter what the number of sources: $\theta \cong 0.1$. This is exactly corresponding to our early simulation in Figure 3.2(a). Note that the one flow case also shows little bit of unfairness. If the bottleneck bandwidth is large (e.g., over 5 Mb/s), we can roughly say that TFRC is reasonably fair. Likewise, Figure 6.8 shows the fairness when RED queue is presented at the bottleneck.

It showed us that no matter what the number of sources, TFRC sources starved much of the TCP sources when the bottleneck bandwidth is less than 2 Mb/s. Also, when there are low level of statistical multiplexed sources, TFRC starved TCP in all cases. However, using TFWC, it showed us it is reasonably fair in all cases: low level of multiplexed sources and low bottleneck bandwidth. In short, TFWC wins in all case against TFRC in terms of protocol fairness across wide range of network parameter settings using *ns-2* simulation.

6.2.2 Protocol Sensitivity

In this section, we describe the sensitivity of the protocol by altering w_q on RED [9] queue. The main purpose of using RED queue at the bottleneck is to keep throughput high but average queue sizes low. The RED gateway has two separate algorithms. The algorithm for computing the average queue size determines the degree of burstiness whereas the algorithm for calculating the packet-marking probability



(a) Fairness Comparison (TCP and TFRC)

(b) Fairness Comparison (TCP and TFWC)

Figure 6.8: Protocol Fairness Comparison using RED queue where $t_{RTT} \cong 22.0$ ms

determines how frequently the gateway marks packets. As described in [9], w_q can determine the average RED queue size in EWMA fashion[6]. In *ns-2*, if $w_q = 0$, then it sets to a reasonable value of $1-exp(-1/C)$, where C is a link capacity. This corresponds to choosing w_q to be of that value for which the packet time constant $-1/ln(1-w_q)$ per default t_{RTT}. If $w_q = -1$, then the queue weight is set to be a function of the bandwidth and the link propagation delay. In particular, the default t_{RTT} is assumed to be three times the link delay and transmission delay, if this gives a default t_{RTT} greater than 100 ms. If $w_q = -2$, set it to a reasonable value of $1 - exp(-10/C)$. In our simulation, we set w_q as a function of the bandwidth and the link propagation delay as belows:

$$t_{RTT} = 3.0 * (\text{link delay} + \frac{1}{\text{ptc}}), \tag{6.2}$$

where

$$\text{ptc} = \text{packet time constant}^{7}$$

Thus,

$$\text{ptc} = \frac{\text{link bandwidth}}{8 * \text{packet size}}. \tag{6.3}$$

Then, w_q can be computed as in the following function:

$$w_q = 1.0 - exp(\frac{-1.0}{10 * t_{RTT} * \text{ptc}}) \tag{6.4}$$

So, w_q can be dynamically computed using Equation (6.4) depending on the link capacity and its propagation delay. In general, w_q should be fixed per a simulation even though the link capacity or the link delay changes during the course of the simulation[8]. For max_p[9] setting, it follows the rule described in [9]. For setting max_{th} and min_{th}:

$$max_{th} = 3 * min_{th}, \tag{6.5}$$

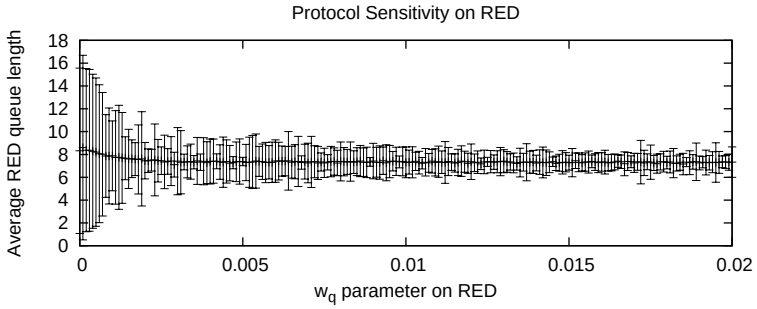
where

[6] $\text{avg} \leftarrow (1 - w_q)\text{avg} + w_q\text{q}$, where w_q is the queue weight factor, avg is the average queue length, and q is the instantaneous queue length. So, as w_q decreases, the importance of avg at the $(n-1)^{th}$ interval increases when calculating the avg at the n^{th} interval. If w_q is too large, then the averaging mechanism will not filter out transient congestion at the gateway. If w_q is set too low, then avg responds too slowly to changes in the actual queue size. In this case, the gateway is unable to detect the initial congestion signal.

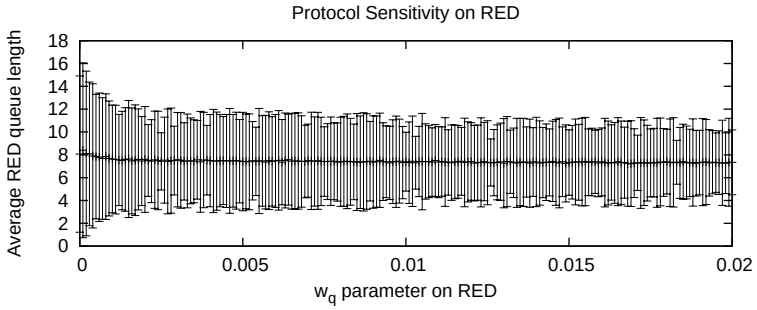
[7] The "ptc" is the max number of average packets per second which can be placed on the link.

[8] We did not change the link capacity and its delay during the simulation.

[9] max_p is the max probability of dropping packet in RED queue.



(a) TFRC Protocol Sensitivity



(b) TFWC Protocol Sensitivity

Figure 6.9: Protocol Sensitivity

$$min_{th} = \frac{\mathsf{target\ queue}}{2}.^{10} \tag{6.6}$$

The target queue can be computed by target delay $*$ ptc[11]. These are all default values set in *ns-2* [14]. When we had simulations on RED and for the use max_p parameter, we calculated an estimated loss rate on that link using Algorithm 1.

Figure 6.9 shows the protocol's sensitivity depending on choosing different w_q values. It shows the minimum/maximum of the average queue length and the averaged average queue length in the middle using RED queue per simulation. So, the y-axis stands for the average RED queue length, and the x-axis stands for the different w_q values. In case of TFWC, the average RED queue length showed very stable no matter what we select w_q values whereas TFRC's average RED queue length was varied slightly more than TFWC depending on w_q values that we choose, especially when w_q is chosen to be very small. The RED queue dynamics of TFWC showed a little bit bulky than that of TFRC. This is because of the TFWC's Ack mechanism; when the queue drains, TFWC sources start filling it up whereas when the RED queue marks packets with the probability p, then TFWC sources start back-off packet transmission. But, after all, what's important in RED is the average queue length.

In this chapter, we have seen that TFWC is robust in various network parameters on both drop-tail and RED gateway, even with different RED parameters. If an application designer has an option between TFRC and TFWC, then TFWC can be certainly better choice on various reasons.

[10] In *ns-2*, if $min_{th} < 5$, then we round it up to 5, so the minimum value of min_{th} is set to 5 packets.
[11] In *ns-2*, the default target delay is set to 5 ms.

Chapter 7

Conclusion

This chapter will summarize the key results obtained while designing and developing the protocol, and whether it has solved the research question raised in Chapter 3. We also mention about the future work briefly, and will conclude the report.

7.1 Summary of Results

TFWC protocol performed better than TFRC over wide range of network parameter settings. The meaning of *"performed better"* is that TFWC is fairer than TFRC across various types of network environment and it is less parameter sensitive. Moreover, because TFWC uses the TCP-like Ack mechanism, it is much cheaper and simpler to implement in a real world system.

The best scenario for TFWC is a network with several reverse path traffic with a drop-tail (and RED) queue regardless of the number of sources, bottleneck bandwidth, round-trip time, and bottleneck queue size. We have seen that as the round-trip time and/or the bottleneck queue size increase, TFWC performed better than TFRC. Both the round-trip time and the bottleneck queue can increase the feedback loop. The increased feedback loop delay will help for the TFWC's history mechanisms which in turn result in the better performance.

The worst scenario for TFWC *at this moment* is a network with no reverse path traffic with drop-tail queue at the bottleneck. In this case, TFWC and TFRC both performed badly, especially with a low level of statistical multiplexing. That is when there are small number of flows (e.g., particularly one or two sources), both protocols performed badly. We assume there is a strong phase effect around this network parameter settings, but unfortunately we do not have clear evidence at the moment. The other assumption might be this is an inherent limitation of using the TCP throughput equation. Overall, TFWC performed much better for the following account.

- It is much simple and cheap to implement in a real world system for real-time streaming applications. In general, the TCP-like Ack mechanisms are simpler than the rate-based control: one Ack comes in, then one packet goes out.
- TFWC is much fairer than TFRC across wide range of network parameters. TFRC performed badly (killed almost all TCP sources) when the bottleneck link speed is relatively low. For this reason, an application using TFRC under a DSL-like Internet might potentially starve other type of applications.
- TFWC is much stable than TFRC. Although there are not many places using RED queue, TFRC is sensitive for the RED parameter settings (susceptible to w_q value) whereas TFWC performed quite stable. This will prevent the overall application's performance after all.

Assuming that TFWC performs just as good as TFRC, TFWC still win the game as it is still cheaper and simpler. In the following section, we describe some of the future work.

7.2 Future Work

First of all, we did not conduct any of real world experiment yet. Although our major goal is not to have an applicability study in this work, it is crucial to check in a real word system. This work would be included for the rest of my PhD, and will be conducted in a few months.

Secondly, we did not pay attention to the multicast, wireless, or anything like that kind of network environment. We have focused on a fixed network environment. Once we prove whether TFWC works fine in a real world (fixed net), it is encouraged to move on to one of those areas.

Finally, we assumed that the packet size is fixed. There might be another chance to look into in more detail for a variable packet size control, which we haven't thought of yet. Apart from it, we might want to optimize the TFWC code further, which is not of our concern yet.

Bibliography

[1] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *In Proceedings of ACP SIGCOMM '99*, pages 263–274, 1999.

[2] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP Congestion Control. `http://www.ietf.org/rfc/rfc2581`.

[3] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly Responsive Congestion Control Algorithm. In *Proc. of SIGCOMM*, pages 263–274, 2000.

[4] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of ACM SIGCOMM '94*, pages 24–35, 1994.

[5] K. Chen and K. Nahrstedt. Limitations of Equation-based Congestion Control in Mobile Ad hoc Networks. *International Journal of Wireless and Mobile Computing (IJWMC), Special Issue on Wireless Ad Hoc Networking*, 1:2, 2005.

[6] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, pages 5–21, July 1996.

[7] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *Proceedings of ACM SIGCOMM '00*, 2000.

[8] S. Floyd and V. Jacobson. Traffic Phase Effects in Packet-switched Gateways. *Journal of Internetworking: Practice and Experience*, pages 115–156, September 1992.

[9] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1:397–413, August 1993.

[10] V. Jacobson and M. J. Karels. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, 1988.

[11] S. Jin, L. Guo, I. Matta, and A. Bestavros. A Spectrum of TCP-friendly Window-based Congestion Control Algorithms. *Technical Report, Computer Science Department, Boston University*, July 2002.

[12] K. Li, M. H. Shor, and J. Walpole. Modeling the Effect of Short-term Rate Variations on TCP-Friendly Congestion Behavior. In *Proc. of American Control Conference*, volume 4, pages 3006–3012, 2001.

[13] M. Mathis, J. Semke, and J. Mahdavi. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM Computer Communication Review*, 27:67–82, 1997.

[14] Network Simulator, Version 2.28. `http://www.isi.edu/nsnam/ns/`.

[15] J. Padhye, V. Firoiu, D. Towsley, and J. Kurouse. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proc. of SIGCOMM*, pages 303–314, 1998.

[16] J. Padhye, J. Kurouse, D. Towsleyi, and R. Koodli. A model-based TCP-Friendly Rate Control Protocol. In *Proc. of the 9th NOSSDAV*, pages 137–151, 1999.

[17] J. Postel. RFC 793: Transmission Control Protocol. `http://www.ietf.org/rfc/rfc793`.

[18] R. Rejaie, M. Handley, and D. Estrin. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proceedings of IEEE INFOCOM '99*, pages 1337–1345, 1999.

[19] I. Rhee and L. Xu. Limitations of Equation-based Congestion Control. In *Proceedings of ACM SIGCOMM '05*, pages 49–60, 2005.

[20] A. Saurin. Congestion Control for Video-conferencing Applications. *MSc Thesis, University of Glasgow*, December 2006.

[21] TCP-Friendly Congestion Control (TFRC) Protocol Homepage. `http://www.icir.org/tfrc/`.

[22] Ultragrid. `http://ultragrid.east.isi.edu/`.

[23] M. Vojnovic and J.-Y. L. Boudec. On the Long-Run Behavior of Equation-Based Rate Control. *IEEE/ACM Transactions on Networking*, 13:568–581, June 2005.

[24] Y. Yang and S. Lam. General AIMD Congestion Control. In *Proc. of ICNP*, 2000.

Appendix A

Proposed Thesis Outline

1. **Introduction**
   - Problem Statement
   - Contributions
   - Structure of the Thesis
2. **Overview of Internet Congestion Control**
   - TCP Congestion Control
     - Overview
     - ACK
     - Congestion Window
     - Protocol Description
   - TCP-Friendly Rate-based Congestion Control
     - Overview
     - TCP Throughput Modelling
     - Loss Event Rate
     - Average Loss Interval
3. **TCP-Friendly Window-based Congestion Control**
   - Introduction
   - Detailed Motivation
   - The TFWC Protocol
     - Slow Start
     - ACK Vector
     - TFWC Timer
     - Sender Function
     - Receiver Function
   - Discussion
4. **Experimental Evaluation**
   - Introduction
   - Protocol Validation
   - Fairness
     - TCP vs. TFRC

  - TCP vs. TFWC
  - TFRC vs. TFWC
  - Mixed Traffic
- Queue Dynamics
- Responsiveness Test
- Discussion

5. **Application-level Evaluation**
   - Introduction
   - Applicability Study
   - Experimental Study
   - Discussion
6. **Related Work**
   - TFRC
     - Equation-based Congestion Control
     - Some other variants
   - DCCP
     - CCID 2
     - CCID 3
   - XCP
     - XCP
     - Some other variants
7. **Conclusions and Future Work**
   - Summary of Contributions
   - Further Work
8. **Appendix**
9. **References**

Appendix B

PhD Thesis Work Plan

This section provides an outline schedule for the completion of the thesis.

1. **Introduction**
   This section will describe the research statement, contributions, and the structure of the thesis. This task has been partially completed. The research contributions will be modified at the final state of this research.

   Estimated time to complete: 2 weeks

2. **Overview of Internet Congestion Control**
   This work gives a wide background study on the Internet congestion control, mainly TCP and TFRC. This chapter will describe a good foundation on the general congestion control concept. This work has been partially completed.

   Estimated time to complete: 2 weeks

3. **TCP-Friendly Window-based Congestion Control**
   This section will provide the protocol's basics and its working mechanisms: the core design concept of the TCP-Friendly Window-based Congestion Control (TFWC) protocol.

   TFWC Introduction This section will cover the background of TFWC.

   ACK Vector This section explains the TFWC ACK Vector

   TFWC Timer This section describes the TFWC retransmission timer.

   Sender Functionality This section describes the TFWC sender's functionality. This includes the computation of the average loss interval, ACK vector maintenance, the average loss history update mechanism. Also, we will describe how TFWC sender determines a packet loss from the ACK vector.

   Receiver Functionality This section deals with the TFWC receiver functionality.

   The protocol is already implemented over *ns-2* network simulator, and the write-up is partially completed.

   Estimated time to complete: 2 weeks

4. **Experimental Evaluation**
   This section involves the various *ns-2* simulation study about the protocol validation. It will also deal with an extensive comparison study with other types of congestion control mechanisms.

   - Protocol Validation – Check average loss history update mechanism and verify the *cwnd* dynamics.
   - Protocol Comparison – Compare the protocol performance with TCP and TFRC.
   - Queue Dynamics – Study on DropTail/RED queue dynamics and compare with other types of protocols.

- Responsiveness Test – Compare the protocol's responsiveness depending upon the level of congestion.

The majority simulation has been done, but the review process and write-up are still in progress.

Estimated time to complete: 4 weeks

5. **Application-level Evaluation**
   This section provides an applicability study inside the UltraGrid [22] project. This work has not yet been started, but it will be followed by the completion of "Experimental Evaluation."

   Estimated time to complete: 12 weeks

6. **Related Work**
   This section reviews the related work for the real-time multimedia application. It also gives the state-of-art in the relevant research area. This section will include the following research activities.

   - TFRC
   - DCCP
   - XCP

   This work has bee partially completed.

   Estimated time to complete: 3 weeks.

7. **Conclusions and Future Work**
   This section summarizes the research and its outcomes. Also it provides the contributions to the related research area.

   Estimated time to complete: 1 week.

8. **Thesis Write-up**
   The completion of thesis write-up and integration of final results.

   Estimated time to complete: 20 weeks.

Appendix C

Detailed Descriptions of the Plan

C.1 Detailed Descriptions of the Plan

This section describes the detailed descriptions of the PhD research plan.

C.1.1 Introduction

We use *ns-2* network simulator for the protocol validation test. The simulation topology is composed of one bottleneck node and TFWC source and destination nodes. We assume that the application traffic is always available, which means a packet is always ready to be sent out from the sources.

C.1.2 Protocol Validation

1. `ACK` Vector
   The purpose of this test is to verify whether the TFWC `ACK` vector is reacting correctly in case of a packet loss. We artificially drop a certain packet number and see if it is reflected to the `ACK` vector. Also, we drop a series of packets in a window and check the `ACK` vector. So, the core part is as follows.

   - `ACK` vector build-up process
   - ACK of ACK
   - `ACK` vector update mechanism

   This work is completed and the protocol is working fine as we have expected.

2. Average Loss History Update
   The purpose of this test to verify whether the TFWC loss history mechanism is behaving correctly in case of a packet loss. We set various periodic packet drop rate and see if the loss history is updated as the protocol specification. The core check items are described as the below.

   - Set packet drop rate to 0.1 (periodic packet drop rate), and check if the loss history is filled with 10 in all fields.
   - Set packet drop rate to 0.02 (periodic packet drop rate), and check if the loss history is filled with 50 in all fields.
   - Set packet drop rate to 0.01 (periodic packet drop rate), and check if the loss history is filled with 100 in all fields.
   - Set packet drop pattern as [OX...O] out of 100 packets, and check if the loss history is same as the one in $p = 0.1$ case.

- Set packet drop pattern as [OXOX...O] out of 100 packets, and check if the loss history is same as the one in $p = 0.02$ case.
- Set packet drop pattern as [XO...OX] out of 100 packets, and check if the loss history is same as the one in $p = 0.02$ case.

This work is completed and the protocol is behaving as we have expected.

3. `CWND` Validation
   The purpose of this test is to check whether TFWC's congestion window increases/decreases its size in a right manner. The core items are in the below.

   - Check `CWND` is doubling its size per an `RTT` before a packet loss
   - Check `CWND` is increasing 0.1 packet per an `RTT` after a packet loss
   - Check `CWND` size reflects the packet drop rate for a flow
   - Check `CWND` is corresponding to ALI

   This work is completed and the protocol is behaving as we have expected.

4. Phase Effect
   The purpose of this text is to break phase effect so that the validation test is actually effective. We take various methods to break it as described in the below.

   - Send a group of packets with random probability even though there is enough `CWND` size available. In other words, do not send a packet as soon as it available, but send the packet an RTT later or two.
   - Use RED router. This is rather complex to deal with per simulation scenarios, but RED router will effectively break the phase effect.

   This work is completed.

C.1.3 Protocol Comparison

1. Throughput

   - 1 source and destination pair for TCP, TFRC, TFWC
   - 4 source and destination pair for TCP, TFRC, TFWC
   - Many source and destination pair for TCP, TFRC, TFWC

   This work is partially completed.

2. Queue

   - DropTail
   - RED

   This work is partially completed.

3. Fairness

   - Generate 3-D graphs
     From 1 source for each to 100 or 200 sources
     From a very small queue size to a very large queue size

   This work is completed. (We need to verify that some DropTail queue cases if it is a phase effect or not. We strongly think it is the phase effect, but do not have a clear clue yet.)

C.1.4 Application-level Evaluation

1. Introduction
    - Introduction to Real-time Multimedia
    - Video Streaming Codecs
2. Applicability Study
    - TFWC with Real-time Streaming Application
    - UltraGrid Project run by USC-ISI and U. of Glasgow
      Modular software for different types of codecs and congestion control
3. Simulation Study
    - Performance Evaluation
4. Discussion