Congestion Control for Real-time Interactive Multimedia Streams

Soo-Hyun Choi

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

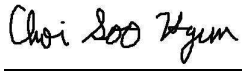
University College London

Department of Computer Science

University College London

September 30, 2010

I, Soo-Hyun Choi, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.



To my parents,

Young-Il Choi and Kil-Ja Shim

Abstract

The Internet is getting richer, and so the services. The richer the services, the more the users demand. The more they demand, the more *we* guarantee[1].

This thesis investigates the congestion control mechanisms for interactive multimedia streaming applications. We start by raising a question as to why the congestion control schemes are not widely deployed in real-world applications, and study what options are available at present. We then discuss and show some of the good reasonings that might have made the control mechanism, specifically speaking the rate-based congestion control mechanism, not so attractive.

In an effort to address the problems, we identify the existing problems from which the rate-based congestion control protocol cannot easily escape. We therefore propose a simple but novel *window*-based congestion control protocol that can retain smooth throughput property while being fair when competing with TCP, yet still being responsive to the network changes.

Through the extensive *ns-2* simulations and the real-world experiments, we evaluate TFWC, our proposed mechanisms, and TFRC, the proposed IETF standard, in terms of network-oriented metrics (fairness, smoothness, stability, and responsive), and end-user oriented metrics (PSNR and MOS) to throughly study the protocol's behaviors. We then discuss and conclude the options of the evaluated protocols for the real application.

[1] We as congestion control mechanisms in the Internet.

Acknowledgment

First of all, I am deeply indebted to my supervisor, Prof. Mark Handley, for giving me invaluable guidance, encouragement, and help on countless occasions throughout my PhD study. His intellect, rigor, enthusiasm and humor have always inspired me in various ways.

Although a doctoral thesis is the product of largely one's own work, I have been fortunate to have collaborative works with outstanding people to whom I express my gratitude: Piers O'Hanlon and Adam Greenhalgh at UCL Computer Science Department. Specially, numerous discussions with Piers O'Hanlon have been extremely helpful with developing and improving the Vic application. He has also helped with the Vic development through acting as a project mentor during the course of Google Summer of Code in 2008. Adam Greenhalgh has spent his time and effort to set up the lab facilities for the experimental stages. I also thank him for giving me useful feedback in the early stages of my thesis draft.

I would like to thank my thesis committee members, Colin Perkins and Miguel Rio, for their time and effort in reviewing the draft and providing valuable suggestions to better shape this thesis.

I would like to acknowledge the financial support during my PhD studies by the British Chevening Scholarship that enabled this research in the first place, and the UCL Departmental Research Studentship that funded the remainder of my PhD study. The development of the Vic application was partially supported through the Google's Summer of Code (GSoC) program in 2008.

It has been a pleasure to get to know and to work with so many wonderful colleagues at Networks Research Group of UCL. Brad Karp always showed us his passion and shared research ideas with us in a number of different occasions. I admire him for his presentation skill not only in research seminars but also in general discussions. Peter Kirstein introduced the UCL Networks Research Group to me while I was working for ETRI, Korea, which eventually led me into joining. Angela Sasse provided a research desk to me when finalizing the thesis. Also, I would like to thank to the members of UCL Networks Research Group, Socrates Varakliotis, Costin Raiciu, Vladimir Dyo, Mohamed Ahmed, Felipe Huici, Andrea Bittau, Petr Marchenko, Daniele Quercia, Yangcheng Huang, Jie Xiong, Ran Atkinson and Saleem Bhatti, for their various help and thoughtful discussions. I also benefited from numerous email discussions with Bob Briscoe at BT Research and Jon Crowcroft at Cambridge.

I would not miss thanking the members of UCL Computer Science Technical Support Group (TSG), Denis Timm, Neil, Daeche, John Andrews, Adil Patel, Nick Turpin, and Dave Twisleton, in providing excellent support in terms of my software and hardware needs. Their timely support has been invaluable

when conducting experiments and setting up research environment.

I was privileged to work with Christopher Clak and John Dowell as their teaching assistant during 2004 – 2008 in UCL Computer Science Department. Their enthusiasm influenced and helped me to establish teaching philosophy.

I have been delighted to have many Korean friends in London who spent their time for sharing fine beer and wine with me, not to mention black coffees. I thank to Yoonjae Lee, KJ Yoo, Sang-Hyuk Lee, Ho Bae, Min-Hyouk Park, and many others whom I cannot write all. I would also like to give my special thanks to Grace Park who made her enormous commitment to improving my written English in the later stages of my thesis manuscript.

Finally, my parents, Young-il Choi and Kil-Ja Shim, have always been the sources of comfort, refreshment, and encouragement. Without their endless love, I would not have come this far. I dedicate this thesis to my beloved parents.

Contents

List of Figures

List of Algorithms

List of Tables

Acronyms

Ack Acknowledgment

AckVec `Ack`-Vector

ANSI American National Standards Institute

AoA `Ack` of `Ack`

AIMD Additive-Increase Multiplicative-Decrease

ALI Average Loss Interval

BDP Bandwidth-Delay Product

CBR Constant Bit Rate

CIF Common Intermediate Format

CoV Coefficient of Variation

cwnd congestion window

DCCP Datagram Congestion Control Protocol

DCT Discrete Cosine Transform

DSL Digital Subscriber Line

DupAck Duplicate `Ack`

ECN Explicit Congestion Notification

EWMA Exponentially-Weighted Moving Average

FTP File Transfer Protocol

fps frames per second

HEN Heterogeneous Experimental Network

IETF Internet Engineering Task Force

ISDN Integrated Services Digital Network

ISP Internet Service Provider

ITU International Telecommunication Union

IRTF Internet Research Task Force

JPEG Joint Photographic Experts Group

LAN Local Area Network

MOS Mean Opinion Score

MSE Mean Square Error

MPEG Moving Picture Experts Group

NAT Network Address Translation

P2P peer-to-peer

PID Proportional-Integral-Derivative

PSNR Peak Signal-to-Noise Ratio

QCIF Quarter CIF

RED Random Early Detection

REM Random Exponential Marking

RTCP RTP Control Protocol

RTO retransmission timeout

RTP Real-Time Transport Protocol

RTT round-trip time

RTTVAR RTT Variance

SNR Signal-to-Noise Ratio

TCP Transmission Control Protocol

TFRC TCP-Friendly Rate Control

TFWC TCP-Friendly Window-based Control

UDP User Datagram Protocol

Vic Video Interactive Conferencing

RTCP XR RTCP Extended Reports

Chapter 1

Introduction

Today's Internet is different from yesterday's. It has been uncannily growing in its size and speed over the last few decades, so will be different from tomorrow's. Owing to this prosperity, functions and services of the Internet have developed from a simple text email exchange into a complex business task which are all inseparable from our daily lives. At the heart of the Internet providing these services without fail lies its congestion control mechanism. While the mechanism, mostly known as the Transmission Control Protocol (TCP), has been extremely successful in allocating and sharing the bandwidth as a distributed manner, it is widely admitted that the single control method does not accord with all kinds of applications and environments that have far different requirements in today's Internet. Among countless categories of applications and environments, this thesis addresses the issues around the congestion control mechanisms for real-time interactive multimedia applications.

Recently, the Internet is rapidly evolving to become an adequate platform to convey high-quality multimedia contents that often desire more bandwidth than the normal web browsing or email services. As the Internet infrastructure can afford the demand due to its growth, the usage of real-time multimedia applications has eminently proliferated over the last few years. Although TCP has served remarkably well for reliable packet delivery for time-insensitive elastic type of data transfer, its functions are beyond the requirements of those multimedia applications. This increasing diversity and disparate nature of applications have spurred recent interest in re-designing transport protocols, in which TCP-Friendly Rate Control (TFRC) [30, 31, 36] has emerged as the most adroit control mechanism, and has yet to be widely used in the real world. The reasons are many.

1.1 Problem Statement

Typically, real-time multimedia streaming applications prefer a congestion control mechanism that can provide smooth and predictable sending rate while being responsive fast enough to adapt the network changes reasonably well. For these requirements, TCP is generally not considered suitable for those types of applications, mainly due to its variability and reliability feature. This has led to the development of TFRC, the proposed standard for multimedia streaming applications in Internet Engineering Task Force (IETF), which produces a smoother sending rate while being fair when competing with the standard TCP flows, and also being responsive to the network changes. Shortly after, there have been

a number of literatures that evaluated the protocol's performance, and discussed the possibilities and limitations for the use of the protocol in real situations [23, 25, 56, 57, 62]. All of these works have questioned the initial design intent whether it actually certify the offered features (i.e., smoothness, fairness, and responsiveness), which have known to be *"not always"* [1]. We elaborate them as follows.

- ***Limitations of the Equation:***

  The authors of [62] discussed that the equation[2] itself that TFRC has used bear inherent limitations, resulting in deteriorating the potential benefits that the protocol might have brought. The authors showed the limitation originally came from the convexity property of the equation $(1/f(1/x))$, where x is the loss event rate and $f(\cdot)$ is the TCP equation. This essentially sets a tight upper bound on the actual TFRC sending rate, such that in some cases those throughput would become largely different to the rate it originally should have produced.

  On the other hand, the authors in [56] identified that the different scheme used for TFRC and TCP, when measuring round-trip time (RTT) and retransmission timeout (RTO), would drive in different behaviors between them, especially during the slow-start phase. With shorter link delay, there is an inclination that TFRC sets smaller RTO values than TCP. This initial gap in the sending rate can incite to have different loss rate between TFRC and TCP, which may eventually lead into unwanted or uncontrolled situation. Several studies also showed similar results [23, 62].

  The question is then why they had to introduce the *equation* from the beginning. The answer is palpable: because it can generate smooth throughput using the average filter between loss intervals, giving a transmission *rate*. Assuming the smoothness throughput property is a paramount benefit to the multimedia applications, the equation itself does not appear to affect the final choice of the protocol. We will come back to these issues in Chapter 2.

- ***Limitations of being Rate-based:***

  The author of [57] revealed a practical limitation being a rate-based control, such that when RTT is so small (much smaller than the host OS's interrupt timer granularity), there would be a period that the sender produces zero throughput in an extreme case. This is specially true when the sender has to send a packet later than it should do (due to the OS's interrupt timer granularity), the receiver might meet a situation where it did not receive the packet in that particular RTT, hence ill-reporting the under-estimated rate. The computed rate at the sender is then bounded by $2 * \tilde{X}_{recv}$, where $\tilde{X}_{recv}$ is the under-estimated received rate in the previous RTT interval. In addition, a relatively large network jitter can affect the process of measuring the received rate: the jitter can help generate the under-estimated received rate. In consequence, the TFRC sending rate can be tightly bounded in these circumstances, resulting in being *excessively* conservative mode. The issues here are clear:

[1] Here, *"not always"* means some have worked and others have not. Therefore, it is rather a loose term.

[2] We give the full explanation of the *equation* in Chapter 2 and Chapter 3. For the sake of an argument, we simply use the term in this chapter without further elaboration.

  - It is difficult to build the sending rate accurately due to the clocking issue on the host.
  - It is difficult to measure the received rate correctly in a real system, especially at low sending rate, due to late arriving packets.

  The above mentioned issues are painful when the rate itself breaks, and thus potentially harms other competing flows, not to mention its own traffic. This could certainly serve the reason why the rate-based TFRC has not been widely deployed yet. Moreover, TFRC added a delay-based back-off mechanism to avoid short-term oscillation, thus the question of being a rate-based or not turns out to be an interesting question to ask.

Although it does not look particularly daunting at a glance, the benefit of using the rate-based congestion control mechanism in multimedia applications appears to be unsure yet: promising and unattainable features go together. Despite this question, there are numerous attempts that have tried to address the usefulness of congestion control mechanisms for multimedia applications [43, 46, 60, 61, 63, 66]. All of these works are based on simulation results, lacking a real-world implementation and integration. On the other hand, there are related works [26, 27] which analyzed Skype [8] congestion control schemes: one of the most used application for voice/video chat in the world at present. The authors in [27] identified that Skype seemed to apply some sort of congestion control mechanisms for the video calls, but the mechanism itself showed to be unrefined control with the following findings:

1. They are extremely slow in reaching a fair share of the link.
2. Their utilization never reaches its full capacity.

All of the above facts urge a question of what is the real capacity of congestion control mechanisms for the interactive multimedia applications (i.e., is it really beneficial? Or is it merely causing troubles?). The previous literatures, based on simulation results, consistently educate us it can be quite useful to deliver a better media quality for such an application. Then, was the rate-based controller the source of the troubles? If so, can we develop a window-based version that can retain similar throughput property while still effectively control the rate? (e.g., in a smooth, predictable, fair, and responsive manner)

There is an old proverb, *"Curiosity killed the cat"*. This proverb applied to me, in which the above *"curiosity"* drove me into this research topic to bother myself for some years.

In summary, this thesis explores answers for the following questions:

- Can any kind of congestion control bring a substantial benefit for interactive multimedia applications in today's Internet?
- Would window-based version of the congestion control ever work for such applications? If so, is it better or not?

In an attempt to address the above questions, we describe what we have achieved in the following section.

1.2 Contributions and Scope

The contributions of this thesis are:

- ***TCP-Friendly Window-based Congestion Control (TFWC):***
  In this thesis, we have proposed a simple but novel window-based congestion control protocol, so-called TCP-Friendly Window-based Control (TFWC), that can provide highly smooth transmission rate while being fair to competing TCP flows, and at the same time being responsive enough to quickly adapt the network changes. We have used the same TCP equation and the averaging loss interval mechanisms, similar to TFRC, but re-introduced a TCP-like Acknowledgment (`Ack`) mechanism to mitigate many problems that a rate-based congestion control protocol can cause.

- ***Extensive Simulation Studies for TFRC, TFWC, and TCP***
  Through the extensive simulation studies across the wide range of network parameters (e.g., bandwidth, queue discipline, queue size, RTT, and loss rate), we have identified the options and limitations of TFWC and TFRC in the comparisons between window-based and rate-based congestion control mechanisms for the use of real-time multimedia applications.

- ***Codec Interactions with Congestion Control Protocols:***
  We have proposed and implemented two control mechanisms combining the congestion control protocols into the transmission system of a video streaming application (e.g., codec and video grabber) to regulate the send rate whilst the image grabber and multimedia codec can alter their output rates (frame rate and bit rate) dynamically based on the feedback information from the congestion control protocols. We have showed that the mechanisms have helped producing better image qualities and minimizing the send buffer length by increasing or decreasing the grabbing rate (fps control) and the codec's quantizer (bps control).

- ***Real-world Evaluation of TFWC and TFRC:***
  To the best of our knowledge, it is the first attempt that evaluated congestion control protocols in a real-world application. We have implemented our TFWC and the IETF standard, TFRC, over Video Interactive Conferencing (Vic) tool [12]. Through real-world experiments we evaluated the two protocols' performance using a Common Intermediate Format (CIF) image sequence, `foreman`, using two metrics: Peak Signal-to-Noise Ratio (PSNR) and Mean Opinion Score (MOS). We are *au fait* in all the matters around these metric – PSNR is a limited metric. As we will summarize below, it is beyond our scope to devise a new quality metric. We will discuss further in Chapter 6.

The scope of the thesis are:

- This thesis focuses on the video transmission. Congestion control for voice packets can be often classified in another domain.

- This thesis did not develop any objective or subjective quality measures that can better reflect the user's final impression.

1.3 Structure of the Thesis

In the remainder of this dissertation, we describe the detailed design, implementation, and evaluation results of the TFWC and TFRC over the simulations as well as real-world experiments.

In Chapter 2, we begin by giving an overview of congestion control mechanisms and its interactions with interactive multimedia applications. We then explain how the existing mechanism, TFRC, has approached to be used for such applications. The unsolved problems and potential drawbacks are discussed in this chapter.

Chapter 3 explains the detailed mechanisms of our proposed algorithm, TFWC, and Chapter 4 presents the extensive simulation results together with those of TFRC. In Chapter 4, we discuss our options and limitations based on the simulations to explore the protocol parameters across a wide range of different settings.

Chapter 5 gives an overview on our choice of application, Vic tool, detailing its underlying transmission architecture. We discuss our design options and some of the important implementation aspects.

Chapter 6 presents the results of real-world experiments over the Heterogeneous Experimental Network (HEN) testbed. We mainly examine the protocol's performance in two categories: network-oriented and end-user oriented. For the network-friendly metrics, we investigate the fairness, smoothness, and responsiveness, whereas for the end-user oriented measures, we evaluate them using PSNR and MOS, and provide discussion.

Finally, we summarize our results and findings, and conclude this thesis in Chapter 7.

Chapter 2

Background and Motivation

This chapter introduces the background to the congestion control mechanisms used for real-time interactive multimedia systems. We start by describing the traditional congestion control schemes that are suited to the generic bulk data transfers in the Internet, and we discuss briefly how and why they are not suitable for the multimedia streaming applications, especially those which require real-time interactivity. We also give an in-depth overview of the proposed congestion control schemes for interactive multimedia applications, detailing how they have solved various problems and what issues are remaining. Finally, we identify some of the important limitations of the proposed standard for such applications, and raise questions as to why the proposed mechanism has not been adopted successfully in real-world applications, in which this thesis endeavors to answer.

2.1 Congestion Control for Data Transfer

The Transmission Control Protocol (TCP) [39] is the most widely used mechanism as a mean for congestion control in the Internet. TCP as a transport layer congestion control protocol serves important functions, which are:

- Congestion Control:
  It prevents the sender from overwhelming the network by detecting congestion signals.
- Flow Control:
  It prevents the sender from sending packets faster than the receiver can process them.
- Reliability Control:
  It provides in-order reliable data delivery to the higher application layer.

These functions have been providing an essential mechanism for the stability of the current Internet by capturing the network congestion condition, and altering the transmission rate for the normal data transfer. The control algorithm is based on three mechanisms: TCP `Ack` mechanism, TCP congestion window computation, and congestion signal detection.

- Acknowledgment (`Ack`):
  TCP uses acknowledgments to carry feedback information – each time a receiver gets a data packet, it informs the sender of the sequence number that it received.

- Congestion Window (*cwnd*):

  TCP limits its sending rate by calculating *cwnd* size, which is the number of packets that may be transmitted for a flow. So, a TCP sender can send up to the *cwnd* number of data packets during a RTT, where RTT is the time between delivering a packet and receiving an `Ack`.

- Congestion Signal (loss detection):

  TCP assumes a packet loss is an indication of congestion, and it re-calculates the *cwnd* size in order to reduce the rate of sending packets. In general, TCP detects losses in two different ways:

  - Retransmission Timeout (t_{RTO}, or simply RTO):

    If the sender does not receive an `Ack` before a specified timeout value, then the timer expires and the data is considered lost. A detailed study of the effect of various timeout settings can be found in [20].

  - Duplicate `Ack` (DupAck):

    The TCP receiver accumulatively acknowledges the sequence number of the received packets. A packet loss causes the receiver to re-acknowledge the sequence number of the lost packet when the next packet arrives. On receiving the duplicate acknowledgments of the same sequence number three or more times, the sender considers it as lost which then triggers a *fast retransmission*, followed by the *fast recovery algorithm* until the receiver sends `Acks` for all the packets in the last window.

Since the first TCP implementation, TCP has evolved in several ways, resulting in different versions of TCP being in use today, the most common being TCP *NewReno* and TCP *SACK* [21, 47, 54]. In principle, TCP uses the *cwnd* as a main control mechanism to regulate the transmission rate. The basic rate control mechanisms of these variants are an exponential initialization stage, so-called *slow start* phase, and Additive-Increase Multiplicative-Decrease (AIMD) stage, so-called *congestion avoidance* phase. Figure 2.1 illustrates these phases and how they form TCP's well-known "*sawtooth*" characteristic.

In the slow start phase, the sender doubles *cwnd* upon each `Ack` reception, resulting in an exponential increase in the sending rate. Upon a loss detection, it stops the slow start phase by halving *cwnd* and changes to the congestion avoidance stage, where it can increase *cwnd* additively on every successful `Ack` reception. These increase/decrease functions can be generalized in a formula using two coefficients

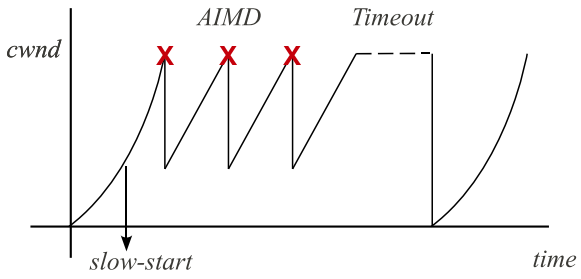


Figure 2.1: TCP slow-start behavior and AIMD characteristic, where X in the graph indicates a packet loss.

(α and β) as below [22]:

$$\begin{aligned} \mathcal{I}: \quad & w_{(t+RTT)} \leftarrow w_{(t)} + \frac{a}{{w_{(t)}}^{\alpha}} && , (a > 0) \\ \mathcal{D}: \quad & w_{(t+\delta_t)} \leftarrow w_{(t)} - b * {w_{(t)}}^{\beta} && , (0 < b < 1) \end{aligned} \tag{2.1}$$

where:

- $\mathcal{I}$: TCP increase function
- $\mathcal{D}$: TCP decrease function
- $w_{(t)}$: *cwnd* at time t
- α, β, a, and b are all constant.

Therefore, for example, if we set $\alpha = 0$ and $\beta = 1$, we then obtain TCP's AIMD formula:

– Increment *cwnd* by *"*$w + a$*"* per RTT

– Decrement *cwnd* by *"*$(1 - b) * w$*"* upon a loss detection.

This section revisited the basic functions and characteristics of standard TCP. Having this in mind, we discuss further its possibilities and limitations for use with real-time interactive multimedia applications in the next section.

2.2 Anti-TCP Dogma

The two common grounds that inhibit the use of TCP for real-time *interactive* streaming services are *Additive-Increase Multiplicative-Decrease (AIMD)* and *re-transmission*.

2.2.1 AIMD

As we have explained in Section 2.1, TCP controls the transmission rate using the *cwnd* mechanism to probe available bandwidth, bearing the familiar "saw-tooth" shape in the send rate where it cycles between *Additive-Increase Multiplicative-Decrease* procedure. In steady-state, TCP converges on an average transmission rate close to a fair-share of available bandwidth[1]. When viewed over shorter time-scales, the instantaneous TCP throughput exhibits high variability because of the abrupt reduction in the send rate upon a loss detection. When applied to the multimedia applications, they must quickly adapt the data rates accordingly to match the suggested rate by TCP. However, due to the sudden changes in the throughput, a multimedia application that strictly follows this established rate can introduce a significant change in media quality, which then affects the user's final impression.

The sender and receiver can implement a buffering technique (send buffer and play-out buffer, respectively) to mitigate the rapid changes in a shorter-term, but it adds delay (and jitter) that brings considerable instability in the received rates. Therefore, a multimedia system that has a large buffer size using TCP congestion control can increase the overall end-to-end delay, hence lessen the interactivity

[1]The fairness definition under distributed control is somewhat subjective. TCP's control algorithm results in bias toward flows with shorter path RTTs.

of the application. Therefore, in general, TCP is not suitable to carry multimedia traffics that require a tighter delay bound of packet delivery.

2.2.2 Re-transmission

Another issue with TCP in multimedia application is that it offers a reliable packet transmission: re-delivery of the lost packets. Again, this feature is not desirable when applications impose a strong timing limit; for example, given the real-time nature of video, the re-transmitted data would arrive at the receiver too late for display in time. This latency constraint can be addressed through client-side buffer management, but only if it does not conflict with application-level latency requirements. With TCP, the point of conflict will be exacerbated when the latency requirements of the application are close to the RTT. A TCP sender's earliest detection of lost packets occurs in response to DupAck from the receiver, therefore the earliest time the re-transmission will arrive at the receiver is one full round-trip time after the original data was lost. For interactive applications such as video conferencing or distributed gaming, users are highly sensitive to end-to-end delays of sub-second timescales, typically in the range of 150 to 200 milliseconds. This end-to-end delay requirement persists for the duration of these applications. Unlike purely-interactive applications, video on demand (VoD) services have interactive requirements only for control events such as start, pause, fast-forward, channel swap, etc., which are relatively infrequent compared to the normal streaming state. The VoD types of applications are rather resilient over changes in the end-to-end latency as the interaction is uni-directional. So, a VoD application may gradually increase the buffer size, hence end-to-end delay, by dividing its use of available bandwidth between servicing video play-out and buffer accumulation. After a time, the end-to-end delay will actually be quite large, but the user perceives it only indirectly, in the sense that the quality during the buffer accumulation period might have been slightly decreased. In this regard, the VoD service does not bear the strict latency requirements of purely-interactive applications, thus TCP's packet retransmissions may not introduce a significant problem. On the other hand, as mentioned above, the strict end-to-end latency requirement limits the use of TCP for the real-time *interactive* multimedia streaming services.

In summary, the interactive multimedia applications have in common two key demands of a transport protocol, in that:

- They prefer timely packet delivery over perfect reliability
- They desire a smooth predictable transmission rate

There are a number of proposed mechanisms that address these goals, of which TFRC is currently one of the accepted standards at the IETF. So far, we have mentioned different types of multimedia applications. In the next section, we clarify the classification of these applications.

2.3 A Classification of Multimedia Applications

We define a classification of multimedia applications that describes the nature of the traffic characteristics into two categories.

2.3.1 Interactive vs. Non Interactive

The *interactive* applications, such as video conferencing, or real-time gaming service, etc., involve a two-way (or multi-way) transmission of the data streams. Such an application highly desires a strict timing requirement in the packet delivery system. As briefly mentioned in Section 2.2, in the case of interactive video streaming, they normally allow the end-to-end latency to be 150∼200 ms, or discard those packets without sending them[2]. Moreover, these applications often require less jitter (usually 20ms). For non-interactive applications, the majority of packet transmission is carried out in one direction only (e.g, VoD services) and the timing requirements are much more relaxed. Unlike the interactive streaming applications, they can allow more buffering size in the receiver to adapt a transient quality degradation. The target applications of this thesis are the *interactive* streaming applications.

2.3.2 Real-time vs. Pre-recorded

The media content may be captured and encoded in real-time communication, or may be pre-encoded and stored for later viewing. For example, real-time streaming applications require that the overall system performance must meet the timing constraint required by applications: capturing encoding/decoding, and packet transmission. In the case of pre-recorded material, the capturing time can be significantly lower than the real-time videos, as they are already stored in a file, hence, only needing a few hundreds of microseconds to load them into a memory. The rest of the time it takes to deliver the media contents is similar to the real-time videos; the overall system necessitate packet delivery in a specific time. We mainly use pre-recorded video sequences for real-world experiments in order to reproduce the results in a consistent manner, whereby the motion complexity of real-time video contents varies significantly according to the object movement in front of capturing devices.

To summarize, this thesis addresses the *real-time* and *interactive* multimedia (video) streaming applications (using pre-recorded image sequences), shortly *interactive* multimedia applications[3].

2.4 TFRC

There are numerous algorithms and mechanisms that can be used with the interactive applications. Among them, the TCP-Friendly Rate Control (TFRC) [30, 36] is the only proposed standard at IETF at present, and it is integrated to Datagram Congestion Control Protocol (DCCP) *CCID-3* [44]. TFRC is an equation-based unicast congestion control protocol, which can be easily extended for the multicast applications as well. TFRC starts flows with the TCP-like *slow-start* phase to quickly increase the rate to a fair share of the bandwidth. On detecting the first packet loss, TFRC immediately terminates the slow-start phase. From then on, TFRC receiver updates the parameters (e.g., loss event rate) and feeds the report back to the sender. The sender then computes the new rate from these parameters and regu-

[2] Assuming the current send buffer length is δ (bytes), and the compressed frame size L (bytes), then we can approximate the time it takes for the delivery of an entire frame as:

$$\tau = \frac{(\delta + L)}{T} + \frac{RTT}{2} + \epsilon \ ,$$

where ϵ includes the time it takes at the receiver (e.g., at play-out buffer and display device), and the network propagation delay. Then, τ should be at most 150∼200 ms, or discard the packets.

[3] By definition, *interactive* holds *real-time* properties.

lates the sending rate accordingly. In addition, to prevent a short-term fluctuation in the calculated rate, TFRC borrows a delay-based congestion avoidance by adjusting the inter-packet gap using a non-linear equation: $\text{New Rate} = \frac{a\sqrt{R_0}}{\mathbf{E}(\sqrt{RTT})} * \text{Calculated Rate}$, where R_0 is the most recent RTT sample, and a is a constant.

The merit of TFRC protocol over TCP is that it can retain smooth sending rate while still being responsive to congestion signals. TFRC estimates the equivalent TCP send rate (T) using the TCP throughput equation [52]:

$$T = \mathbf{min}\left\{\frac{s \cdot w_{max}}{t_{RTT}}, \frac{s}{t_{RTT}\sqrt{\frac{2bp}{3}} + t_{RTO}3\sqrt{\frac{3bp}{8}}p\left(1+32p^2\right)}\right\} \quad (bytes/sec)\,, \qquad (2.2)$$

where:

- T: throughput
- t_{RTT}: round-trip time
- t_{RTO}: retransmission timeout
- s: segment size[4]
- p: loss event rate
- b: number of packets acknowledged by each `Ack`
- w_{max}: maximum congestion window

This equation gives an upper bound on the sending rate T in bytes/sec as a function of the packet size (s), measured RTT, loss event rate (p), and the timeout value (t_{RTO}). TFRC approximates the RTO using a linear function of the measured RTT [36]. TFRC receiver measures the packet loss rate in terms of loss intervals, spanning the number of packets between consecutive loss events. The loss event rate is then calculated by the inverse of the Average Loss Interval (ALI): the average interval between losses for the last k history with different weighting factors. The receiver periodically provides this calculated rate to the sender. The smoothness of throughput is achieved by averaging p, therefore T, over the last k RTTs, where k is the history size of the loss intervals. It is known, though, that the TCP equation works well in environments with a high-level of statistical multiplexing, but care must be taken when only few flows share a bottleneck link. In such an environment, changes to the sending rate reconstruct the conditions in the bottleneck, which in turn determine the sending rate through the equation. Such a feedback loop can render the results of the TCP equation being less infallible. Further discussions are followed by the next section.

As discussed earlier in this chapter, TCP does not provide a satisfactory sending rate for interactive multimedia applications, and this fact has led to the development of slowly responsive congestion control protocols for such an application, TFRC, which has been proven to retain the desired level of smoothness extremely well in the send rate. This characteristic has brought about an important implication: reduced

[4]For the sake of simplicity, we refer "*s*" to "*packet size*" for the rest of this thesis. We, however, use the precise term, "*segment size*", where necessary.

amount of send/receive buffer size. With TCP, the system would need to implement substantial buffers to alleviate drastic changes in the received rate by spacing out the packets.

2.5 The Problems of TFRC

Despite the perfect smoothness in the send rate, TFRC has not yet been so successful enough to be used for a real-world application at present. There are a number of recent works that have identified the problems of using the rate-based congestion control mechanisms in practice [23, 25, 41, 55, 56, 57, 62].

The authors in [56] and [62] showed that the convexity property of the TCP throughput equation and the different RTT measuring methods can result in a long-term throughput imbalance, where sources receive less throughput than originally intended. They also showed that a slowly-responsive flow can suffer a higher loss event rate than the standard TCP flow when competing at the same bottleneck. This can result in a negative impact in the throughput difference between TFRC and TCP flows, and can cause, in some extreme cases, TFRC to consume up to twenty times more, or on the contrary, ten times less bandwidth than a TCP flow in the same condition.

The author in [57] identified that the OS's interrupt timer granularity can cut the rate inappropriately, especially when the network RTT is extremely short with higher sending rates (e.g., inter-packet interval being less than the OS's clock granularity). The root causes of the problem come from the latency in the OS's `sleep()` function[5] such that the sender ends up sending no packets in some RTTs because the OS scheduler was not able to select the process in time, resulting in the receiver reporting $X_{recv} = 0$ to the sender for those instances. On the next rate computation, the sender is then bounded by the policy, $X_{new} < 2 * X_{recv}$, that the new rate could become zero in the worst case. Similarly, the receiver might not be able to send a feedback packet once every RTT because of the timing mismatch between a short RTT and the OS's latency. Therefore, TFRC algorithm itself poses limitations on the endpoint's system performances in which those capabilities play an essential role in the overall transmission system. In order to mitigate the system overload at the receiver side and to solve the *late* feedback report arrival, there are recent ongoing works that have proposed so-called *sender-based* TFRC mechanisms [41, 55].

The authors in [55] identified some of the important implementation aspects of TFRC that have revealed problems in the real-world, for example, measuring RTT, inter-packet interval, and loss-rate estimation. The authors suggested measuring RTT at the sender side to overcome such shortcomings in real situations. The authors in [41, 55] also suggested it being a sender-mode, because most end-users nowadays are short of processing power (e.g., the streaming receiver could be a low-powered device, such as smart phones). If the receiver has to compute the loss event rate and provide feedback once every RTT, it could easily overload the device, especially with those higher transmission rates.

Moreover, the equation-based mechanism can, by itself, cause oscillatory behavior due to the overshooting characteristic; for this reason TFRC builds in a short-term mechanism to reduce the sending rate as the RTT increases. The authors in [23] also observed related problems where rate-based congestion control receives less bandwidth in a highly dynamic network condition (e.g., highly dynamic loss rate condition). The authors introduced a conservative mode to TFRC, whereby the sender is limited by

[5]The author provided that this timing error can be 5 ms on average in modern UNIX-like systems.

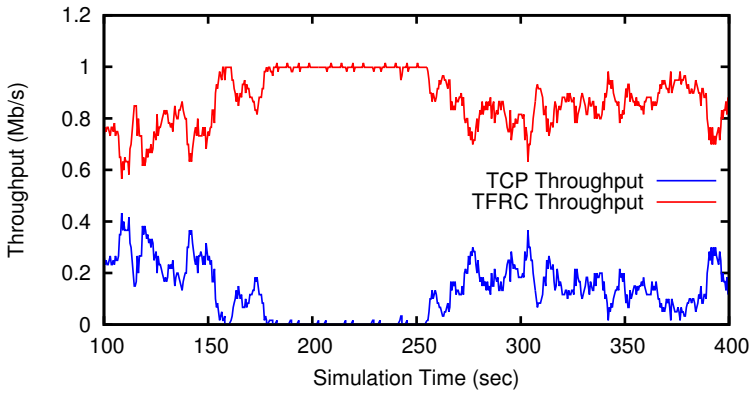


Figure 2.2: TCP vs. TFRC in a typical DSL-like link with a low level of statistical multiplexing.

the measured arrival rate at the receiver. In their simulation results, this worked well to mitigate many problems, but accurately measuring the receive rate (and indeed finely controlling the sending rate) in *real* systems can be hard.

In *ns-2* simulations we have also observed that if a flow traverses a low statistically multiplexed network link such as a Digital Subscriber Line (DSL) line using drop-tail queue, TFRC source can starve TCP sources. This is because TFRC lacks the *fine-grain* congestion *avoidance* mechanism that TCP's `Ack`-clocking provides, meaning that TFRC can briefly overshoot the available link capacity. This fills the buffer at the bottleneck link and can cause TCP's `Ack`-clock to decrease the transmit rate, whereas TFRC does not back off so much.

For example, in *ns-2* simulation, TFRC sources can starve TCP sources as in Figure 2.2; this example is from a dumbbell topology with an 1 Mb/s bottleneck carrying two TCP and two TFRC flows with a range of short RTTs.

All of the problems above essentially stem from the same basic cause: it is difficult to build a rate-based protocol with a rate that is inversely proportional to the round-trip time while achieving stability in all conditions. All these issues beg the question why TFRC is rate-based at all. There were two original motivations:

1. TFRC was intended to be usable for multicast, where window-based protocols are not practical; this is not relevant for many applications.

2. The intent was that a rate-based protocol would be smoother than a window-based protocol on time-scales of a few RTTs.

On shorter time-scales even rate-based protocols cannot be smooth due to clocking issues on the host, and on longer time-scales the behavior is dominated by the varying loss rate. But to be stable, as mentioned earlier, TFRC had to include a short-term mechanism to back off the rate as the RTT increased, so even TFRC exhibits some short-term variability in rates. Thus, it behooves us to ask the question of whether being rate-based is worth all the trouble.

2.6 Summary

In this thesis, we examine a window-based version of TFRC, which uses a TCP-like `Ack`-clocking mechanism while applying the same TCP equation, à la TFRC, to directly adjust the window size. Our objective is to remedy the issues discussed above which relate to the combination of rate-based control with the rate being inversely proportional to the RTT. Using a window makes the RTT implicit in the `Ack` clock, and removing the need to be rate-based makes life much simpler for application writers, as they no longer need to work around the limitations of the OS's short duration timers. This thesis endeavors to answer the questions around these problems, hoping to migrate the congestion control mechanisms prevailing to the interactive multimedia applications.

Chapter 3

TCP-Friendly Window-based Congestion Control

This chapter presents our TFWC [25] protocol. After an overview of the main protocol features, the following sections will describe the basic mechanisms and the important characteristics of TFWC, detailing some aspects of the behavior of this congestion control algorithm.

3.1 Overview

The TCP-Friendly Window-based Control (TFWC) protocol is a TCP-like congestion control protocol for real-time *interactive* multimedia streaming applications. The key objectives of TFWC are to provide *smooth* and *predictable* throughput for multimedia streaming applications while maintaining a reasonable degree of *fairness* to competing TCP flows and to keep *responsiveness* fast enough to adapt to sudden network changes. Specially, for the *interactive* streaming applications, it adds a strict timing requirement. These objectives are exactly the same as what TFRC protocol has aimed at, and a number of media congestion control mechanisms have had a similar goal. So, those congestion control mechanisms increase and decrease its sending rate based on a packet loss rate or network delay (e.g., RTT). Among them, for example, a TCP-like congestion control mechanism increases the *cwnd* each time an `Ack` indicates successful transmission or decreases when packets are lost while keeping the above mentioned goals. TFWC, our proposed mechanism, calculates the `Ack`-clocked *congestion window* just like the standard TCP, but uses the TCP throughput equation [52] when determining its size. This is one of the key differences of TFWC to TFRC which uses the calculated sending *rate* in a rate-based controller. Also, as most real-time streaming media applications prefer timely packet delivery over perfect reliability, TFWC does not have the retransmission feature for the lost packets.

TFWC is operated in two phases, slow start and congestion avoidance phase, where the congestion avoidance phase can be subdivided into two modes as below:

- Window-and-rate Hybrid Mode
- *cwnd* Jitter Mode

In the slow start phase, TFWC doubles the *cwnd* each round-trip time (RTT) just like normal TCP, whereas in the congestion avoidance phase it calculates the *cwnd* size from the TCP equation using the

packet loss event rate. If the network experiences a high packet loss rate where *cwnd* persistently remains very small (like one or two packets), TFWC runs a rate-based timer-out mode when sending packets, so-called the rate-driven mode. In the rate-driven mode, TFWC calculates send *rate*, for example, inter-packet interval using the TCP equation, instead of calculating *cwnd*. Another interesting characteristic worth mentioning is that TFWC, in the simulations, often does not exhibit smooth or fair enough for the use of streaming applications, especially when the bottleneck uses drop-tail queue. However, we have observed that this behavior significantly disappears when used with an active queue (e.g., RED queue) in the bottleneck. From this observation, we, therefore, introduced the *cwnd* jitter mode which artificially inflates *cwnd* a little in the same RTT hoping to get *early* packet drop notification emulating the Random Early Detection (RED) queue behavior at the data *sender*.

These modes can be run simultaneously during the congestion avoidance phase. For example, TFWC uses the jitter mode in the congestion avoidance phase where it switches into the rate-based mode when *cwnd* becomes very small. In the remainder, we describe the TFWC protocol operations in more detail, explaining the sender and receiver's functionality of the protocol.

3.2 The TFWC Protocol

TFWC is a sender-driven congestion control protocol with the help of feedback message from a receiver, meaning the data sender computes *cwnd* size upon every `Ack` reception. The receiver, on the other hand, only needs to maintain a data structure, what we call `Ack`-Vector (AckVec), which reports the received packet sequence numbers to the sender. Simply speaking, the sender computes the packet loss rate (p) and calculates *cwnd* using the TFWC equation (which we will introduce in the next subsection), whereas the data receiver maintains a feedback data structure (i.e., AckVec). Start by giving the TFWC equation, we describe what parameters are necessary and how to calculate them when determining *cwnd*. We, then, explain the detailed mechanisms that are required in the sender and receiver sides, respectively.

3.2.1 TFWC Equation

Before explaining the detailed TFWC mechanisms, we introduce a simplified version of the TCP equation. For most applications it suffices to set t_{RTO} to a multiple of RTTs and to have a fixed size packet. While the standard TCP's t_{RTO} can be calculated as defined in [53], the TFRC protocol specification [31] used an approximated t_{RTO} value to reduce the computation load, that is $4 * \mathsf{RTT}$. Given that TFRC does not compute RTTVAR directly in the sender, it is a good simplification for TFRC to use a *conservative* estimate of t_{RTO}, so that it does not overshoot too much in heavily congested environments. In other words, in times of low congestion with a low loss event rate, the t_{RTO} term does not have much effect on the original TCP equation. Therefore, with $t_{RTO} = 4 * \mathsf{RTT}$ rule, the original TCP equation (Equation (2.2)) can be rewritten as below:

$$T = \frac{s}{t_{RTT}\left(\sqrt{\frac{2p}{3}} + \left(12\sqrt{\frac{3p}{8}}\right) p\left(1 + 32p^2\right)\right)} \qquad (bytes/sec)\ . \tag{3.1}$$

TFRC uses this equation when computing the sending rate, whereas in our case we want to compute the send window in *packets*, instead of send rate ($bytes/sec$), by multiplying t_{RTT}/s to the both sides of Equation (3.1). Here we have assumed the packet size is fixed.

First, let,

$$f(p) = \sqrt{\frac{2p}{3}} + \left(12\sqrt{\frac{3p}{8}}\right) p\left(1 + 32p^2\right) \ , \tag{3.2}$$

then Equation (3.1) can be expressed as

$$T = \frac{s}{t_{RTT}\ f(p)} \tag{3.3}$$

By multiplying $\frac{t_{RTT}}{s}$ to the both sides, we get

$$W = \frac{1}{f(p)} \ , \tag{3.4}$$

where W is the number of packets and p is the packet loss rate. With this equation we can compute the sending window, W (in packets), by simply using the packet loss rate. We call Equation (3.4) as the *TFWC equation* for the rest of the thesis.

3.2.2 Calculating Parameters

To calculate the window (*cwnd*), TFWC only needs to compute the loss event rate (p), as the RTT is implicit in an `Ack`-clock. The TFWC equation also shows the RTT does not directly affect the computed *cwnd* value. TFWC also needs to calculate an RTO timer (although this will actually send a new data packet rather than a retransmission), for when the `Ack`-clock stalls. To calculate p we use the same ALI mechanism that TFRC has used, where the last eight intervals between loss events are held, and a weighted average is calculated.

To compute ALI we use an n-ary loss history information, where each bucket in the array has the number of packets between loss events; for this reason we call the numbers in each array as the loss interval. After all, the loss history is an array that stores the last n loss intervals. As mentioned, we have chosen $n = 8$ just like TFRC, meaning that there are 8 elements in the history array in addition to the current loss interval. This way we have a total of 9 buckets in the array as shown in Figure 3.1, where $\mathsf{I}_{\mathsf{new}}$ indicates the 8 history information including the most recent loss interval and I indicates the 8 loss history excluding the current interval. We then compare $\mathsf{I}_{\mathsf{new}}$ to I and take the larger value, ignoring the current loss interval if it is not enough to change ALI.

Then, we have:

$$\mathsf{Loss\ Event\ Rate\ (p)} \ = \ \frac{1}{\mathsf{Average\ Loss\ Interval}}. \tag{3.5}$$

3.2.3 `Ack` Vector

As TFWC is intended to be used for unreliable data streams, the calculation of loss needs a little care, and we use a mechanism similar to that used by DCCP [45].

Each TFWC `Ack` message carries feedback information, but unlike TCP's `Ack`, it carries no cumulative `Ack` numbers as these would be meaningless for unreliable flows. Instead we maintain an `Ack`-Vector (AckVec) data structure that contains a packet list indicating whether or not the packet was delivered successfully. The AckVec grows in size as the receiver gets packets from the sender, so the `Ack` message itself needs to be acknowledged explicitly to prune the packet lists for the successfully delivered packets. Specifically, the sender must periodically acknowledge the receipt of an acknowledgment packet (i.e., `Ack` of `Ack`); at this point, the receiver can prune the corresponding packet list in the AckVec.

Upon receipt of an AckVec, the sender generates a margin vector by looking at the head value[1] of the AckVec and includes up to three latest `Ack` numbers. For example, if the head value of the received AckVec is 10, then the margin vector will be (9 8 7). This margin vector is to be tolerant of packet reordering by applying the TCP's three DupAck rule, meaning that the sender will *not* consider these sequence numbers when it determines a lost packet in the received AckVec[2]. Now, the sender generates a matching array from the next sequence number up to the `Ack` of `Ack` (AoA), which we call genvec[3]. In the above example, the sender had generated the margin vector up to the packet sequence number 7, so the matching `Ack` array, so-called genvec, will be (6 5 4), assuming the current AoA is 3.

Suppose a sender has just received an AckVec (20 19 18 16 15 14 13 11 10) from a receiver and assume the current AoA is 10, as shown in Figure 3.2. From the latest `Ack` number, 20, the sender calculates the margin vector, (19 18 17), and the genvec (16 15 14 13 12 11). Note that although the packet number 17 is missing, the sender does not think of it as a lost packet *yet* because it is still in the margin. The sender will compare the genvec with the received AckVec and find out which packet sequence number is missing in the list. In other words, the sender will compare the received AckVec with the calculated genvec in order to find out any missing packet sequence number. In this example, we can say the packet number 12 is lost.

In short, from the sender's point of view, what is important always is the latest `Ack` number because, using this value, it can generate margin vector and genvec, and can compare with the AckVec to see if there are any losses.

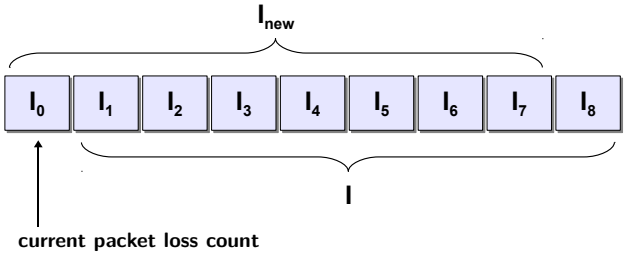


Figure 3.1: Loss History Array

[1] The head value of an AckVec is the latest `Ack` number.

[2] We assume there is no packet reordering in the simulations. Instead, we consider the packet reordering in a real-world implementation later in this thesis.

[3] It is a *generated* vector at the sender by looking at the received AckVec — i.e., it is *not* the actual packet sequence numbers that the receiver is reporting.

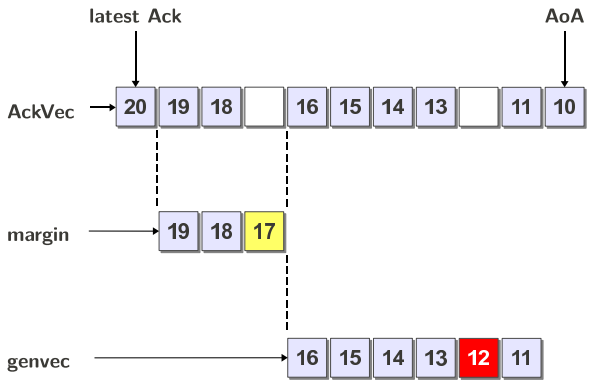


Figure 3.2: TFWC AckVec, margin, and genvec

3.2.4 Principles of Send and Receive Function

We now have necessary parameters to compute the send window. This section explains the principles of the sender and receiver mechanisms: we give a full depth implementation details in the next section.

Unlike TFRC, the TFWC sender calculates the new *cwnd* value, whereby TFWC can be seen as a sender-driven controller and TFRC as a receiver-driven controller. Upon receipt of every AckVec, the sender checks it to see if there are any lost packets. Except the slow-start phase where it doubles *cwnd* when no losses occur[4], it computes the ALI and the loss event rate. By feeding the loss event rate to the TFWC equation, Equation (3.4), the sender computes the new *cwnd* value. So, as illustrated in Figure 3.3, if the sequence number of new data waiting to be sent meets Equation (3.6), it can then be sent, resulting in an `Ack`-clock.

$$\text{seqno of new data} \leq \text{cwnd} + \text{unacked seqno} \tag{3.6}$$

Also, every time an AckVec arrives at the sender, it updates the RTT and the RTO value. With the calculated timeout value, the retransmission timer is set whenever a new packet is sent. If the timer expires, the next available packet is sent[5].

While all the important steps are carried out by the sender, the receiver's functionality is relatively quite simple. TFWC receiver provides feedback information (AckVec and time-stamp echo) to allow the sender for *cwnd* and RTT calculation. The main role of the receiver is to build AckVec and process AoA:

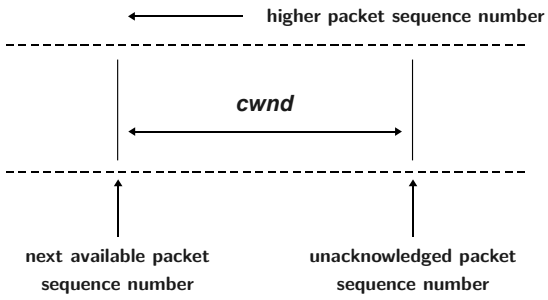


Figure 3.3: TFWC *cwnd* mechanism

[4]This resembles the standard TCP's slow-start phase.
[5]Note, this is, in fact, *not* a packet retransmission caused by that packet loss.

it trims out smaller sequence numbers than AoA when building a new AckVec.

3.2.5 TFWC Timer

As we briefly mentioned above, every time an AckVec is received, the RTT and RTO values are updated. So, every time a sender transmits a new data packet, it sets the retransmission timer using t_{RTO}. On expiration of the retransmission timer, it backs off the timer by doubling t_{RTO}, and sets the timer again with this value. When the retransmission timer goes off, it artificially sends a new data packet if the packets are ready in the transmitter queue. Note, when in the rate-driven mode, the sender does *not* back off the timer as the RTO in the rate-based mode is computed differently. During the rate-driven mode, it computes the packet interval using the TCP equation[6], and sets it as a new RTO, so that the TFWC timer goes off after this calculated interval triggering the next packet transmission; overall resulting in a rate-based timer mode. The TFWC timer mechanism is described in Algorithm 1.

```
1  On the TFWC timer expiration
2      if isRateDriven == true then
3          setRtxTimer(rto);
4      else
5          backoffTimer();
6          setRtxTimer(rto);
7          /* inflate the unacked seqno by one                    */
8          unACKed ++;
9          /* call send method                                    */
10         send(seqno);
11 end
```

Algorithm 1: TFWC Timer

To summarize, on every AckVec arrival, the TFWC sender calculates the new window using the loss event rate, and updates RTT and retransmission timer as illustrated in Figure 3.4.

[6]Assuming the packet size is fixed, the inter packet interval (*ipi*) can be expressed as:

$$\mathsf{ipi} = t_{RTT}\sqrt{\frac{2}{3}p} + t_{RTO}\left(3\sqrt{\frac{3}{8}p}\right)p\left(1+32p^2\right)$$

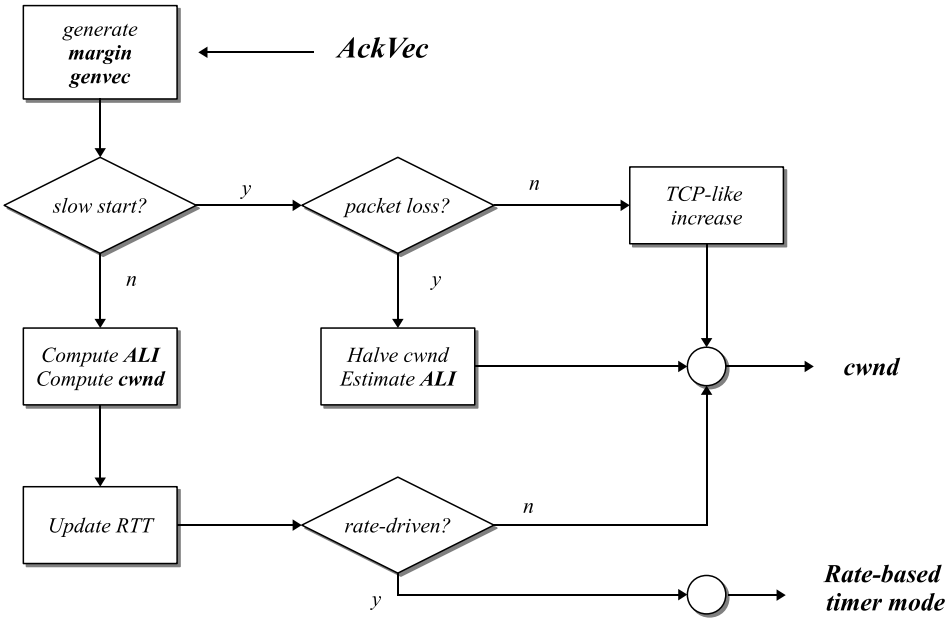


Figure 3.4: Diagram of the TFWC Functions in the Sender

3.3 TFWC Implementation

We have implemented TFWC protocol over *ns-2* network simulator [51], and presented some of the important implementation aspects in this section.

3.3.1 Slow Start

The initial slow start is identical to the TCP's slow start mechanism in which the sender doubles the window per RTT. Once the TFWC sender detects the first packet loss, it halves the current window and seeds the TFRC-like loss history mechanism by calculating the average loss interval that would have produced this window size. To obtain this ALI value, what we do is reverse engineer the TCP equation as described in Algorithm 2. Note it is to approximate the ALI and the loss event rate when the first packet loss occurs (i.e., we do not have any ALI information yet), so TFWC can use it from the next ALI calculation.

```
1  On detecting the very first packet loss
2      /* halve cwnd                                                    */
3      cwnd = cwnd/2;
4      /* increment loss rate from a very small value up to 1, and match */
5      /* the closest window value (increment step size = .00001)       */
6      for pseudo = .00001 to 1 do
7          out = f(pseudo);
8          win = 1/out;
9          if win < cwnd then
10             break;
11     p = pseudo;
12 end
```

Algorithm 2: Approximate the loss event rate on detecting the first packet loss.

3.3.2 Hybrid Window and Rate Mode

When the loss rate is high, TFWC's window will reduce. With very small windows, a loss will cause a timeout because the `Ack`-clock fails. With even higher loss, the equation gives a window size of less than one, which is not useful for any kind of window-based control algorithms. We could use TCP's retransmission timeout mechanism to clock packets out at these very low rates, but constant switching between window and timeout, with potential exponential back-off of the timeout, will result in a very uneven rate. What is required is, as briefly mentioned in Section 3.2.5, to be rate-based when the window is so small the `Ack`-clock cannot function. Thus TFWC falls back to being TFRC-like (but still sender-based) when the window reduces to less than two packets as described in Algorithm 3.

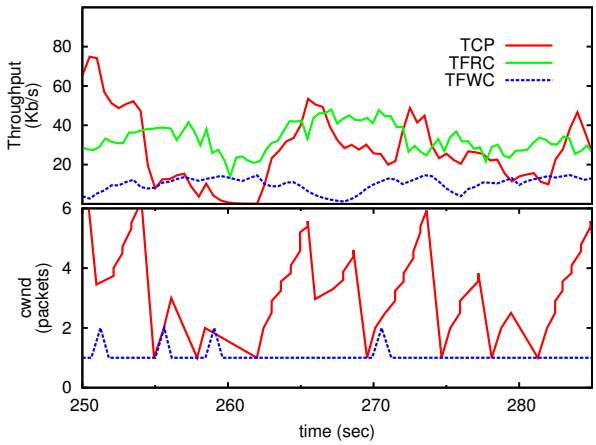
Consider an example where two TCP, TFRC, and TFWC sources all compete for a tiny bottleneck buffer (5 packets) for the bandwidth 500 Kb/s with link delay of 50 ms[7]. Then the steady-state average window size per flow is roughly 1∼2 packets (i.e., very small window size). We took one flow per TCP, TFRC, and TFWC sources for the comparison. Figure 3.5 shows that if TFWC is only operated by the window-driven mode, there are then timeout periods where it ceases sending (simulation time between

[7]The access link speed is 15 Mb/s with random access link delay between 0.1 ms and 2 ms per flows. Also, the BDP is roughly 6∼7 packets assuming the packet size is 1000 bytes.

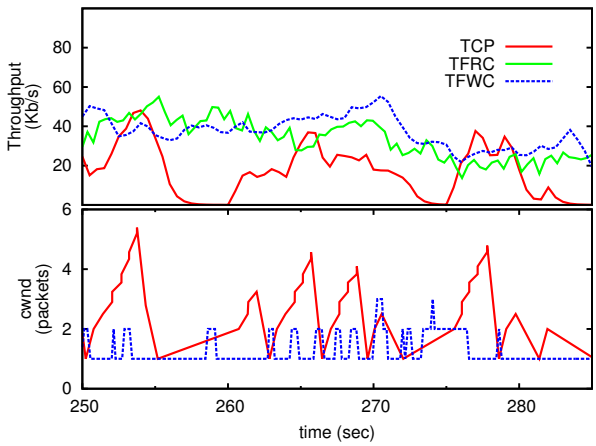
input: Average Loss Interval (avgInterval)

Upon an ACK vector arrival
    p = $\frac{1}{\text{avgInterval}}$;
    out = f(p);
    floatWin = $\frac{1}{\text{out}}$;
    /* convert floating point window to a closet integer */
    cwnd = (int)(floatWin + .5);
    /* if window is less than 2.0, */
    /* TFWC is driven by rate-based timer mode */
    if floatWin < 2.0 **then**
        isRateDriven = true;
    else
        isRateDriven = false;
end

Algorithm 3: cwnd computation: Hybrid Window and Rate Mode



(a) operated by win-based only



(b) operated by win/rate hybrid mode

Figure 3.5: Window-base Only vs. Hybrid Window/Rate Mode depending upon *cwnd* value

265∼270 and 275 sec), while a TFWC that can become rate-based continues sending even when *cwnd* reaches a very small value. TCP in Figure 3.5(b) still frequently goes into timeouts when *cwnd* are small (1 or 2 packets), similar timeouts that TFWC would have been doing with window-mode only, while hybrid TFWC, with such small *cwnd*, can still send packets: the rate it can reach is almost close to the TFRC rate.

3.3.3 TFWC *cwnd* Jitter

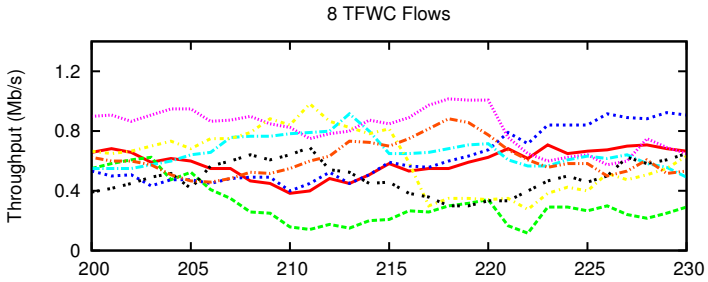
We have observed in simulation that the throughput of TFWC is often less smooth of TFRC, especially with drop-tail queueing at the bottleneck. In such conditions, the individual TFWC throughput was unlikely to be smooth enough for interactive streaming applications, nor was TFWC fair when only competing with itself. This appears to be due to strong simulation phase effects, and flows through the same bottleneck experience different loss rates. TFRC is also somewhat susceptible to phase effects in simulation, and jitters its rate slightly to reduce this. These phase effect can be mitigated significantly with RED queueing at the bottleneck in the simulations. Being motivated by this, we randomly added so-called *cwnd* "**jitter**" at the sender. With a window-based protocol, this can be done by inflating *cwnd* at least once every RTT so that the sender gets an *earlier* notification of the congestion signal. Note the sender actually did not send one additional packet but borrowed a "future" packet that would have been sent later in the RTT. So, we artificially inflate the computed *cwnd* at least once in an RTT with a small random probability.

To this end, we have chosen to inflate *cwnd* in an RTT with 10% of random probability on every AckVec reception. In other words, in the same RTT, each *cwnd* computation routine (initiated by AckVec reception) has 10% of random probability to inflate the computed *cwnd* at the end. However, there would be cases where *cwnd* does not get inflated due to the relatively low random probability that we have chosen. Therefore, we implement so-called "`force_inflater`" on ending the RTT, which ensures inflating *cwnd* at least once in an every RTT.

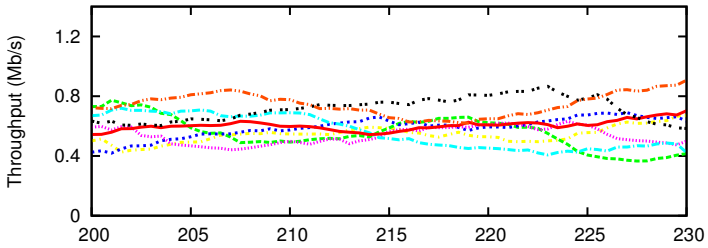
Consider an example that 8 TFWC flows share a 5Mb/s bottleneck link with a delay of 50 ms. Here the access link speed is 50Mb/s with varying link delay between 0.1 to 5 ms. Therefore, the approximated end-to-end delay is 0.1 sec and the Bandwidth-Delay Product (BDP) is about 63 packets assuming the packet size is 1000 bytes. We have set the bottleneck queue length almost equal to the BDP with drop-tail queueing. Figure 3.6 shows that the jittered result is smoother and fairer than the original version, and the level of smoothness/fairness is quite similar to what TFRC can provide (See Figure 3.6(b) and Figure 3.6(c)).

Note that we do not apply the *cwnd* jitter scheme when the loss rate is high (greater than 25%) or its variation is significant (when the difference between the first and last ALI is greater than 10.[8]). This is because the jitter scheme itself may introduce unwanted oscillation when the loss rate is high or its variation highly dynamic. Remember the *cwnd* jitter is only meant to be used for emulating an earlier congestion signal at the sender without harming the original function as a multimedia congestion control protocol.

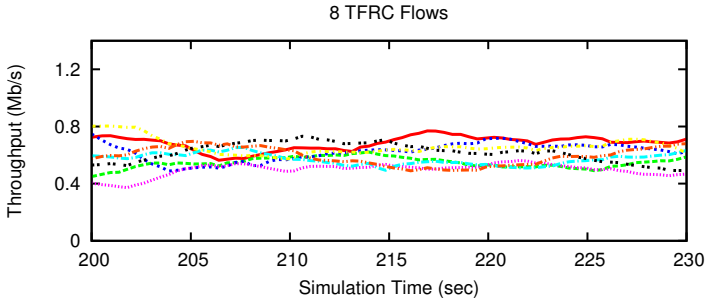
[8]This difference is corresponding to 10% loss rate variation within the same RTT



(a) Without Jitter



(b) With Jitter



(c) TFRC flows

Figure 3.6: TFWC *cwnd* Jitter

3.4 Summary

In this chapter we presented a new TCP-Friendly Window-based congestion control protocol, so-called TFWC, for a real-time interactive multimedia streaming application. To retain such smoothness throughput property, TFWC uses the same equation that TFRC has used, but calculates the send window, *cwnd* instead. We added two features to TFWC to overcome the problems that a window-based control protocol may suffer:

- Prevent frequent timeouts when *cwnd* is very small
  TFWC goes into a rate-based timer mode to clock out packets instead of doing time-out.

- Improve smoothness/fairness by introducing *cwnd* jitter
  TFRC sends packets with an extremely even packet spacing, whereas, with a *cwnd* scheme, TFWC is likely to open a send window in more bursty manner. This property that a window-based control protocol inherently bears may cause the problem. So, to mitigate it, we jitter *cwnd* randomly to get a little earlier congestion notification, solving this uneven/unfair problem after all.

If we can achieve the same goal, with an `Ack`-clocked, sender-driven window-based protocol (e.g., TFWC), that many similar congestion control protocols aimed for, it would be a much more preferable option because we can keep the Jacob's packet conservative rule (with the `Ack`-clock) while retaining the smooth and fair throughput property (with the TCP equation) without having to worry about all the timer issues that a rate-based control protocol faces. We would like to examine the full depth performance evaluation in the next chapter.

Chapter 4

TFWC Performance Evaluation

This chapter presents the results of the TFWC protocol performance analysis compared with TCP and TFRC under various network environments using *ns-2* network simulator [51]. Through the simulations, we explore the protocol's functions and performances using the wide range of network parameters (e.g., bandwidth, bottleneck queue type, queue size, RTT, loss rate, etc). In an attempt to evaluate the performance, we discuss what the options and limits of our proposed protocol are in the comparisons between TFWC and TFRC. Whilst this chapter focuses on the simulation results, we conduct a series of real-world experiments in Chapter 6 to show how such protocols perform in a real application.

4.1 Evaluation Methodology

The correct evaluation of a congestion control protocol can be a complex task. It requires the development of a well-designed set of test environments that can validate the protocol's functions and goals, whether or not in simulations or real-world experiments. In particular, care is needed when evaluating the functions over a controlled environment, for example, with *ns-2*. This means we need to set up correct/meaningful parameters and test scenarios to explain the functions and limitations properly.

First, when choosing various network parameters (for example, network bandwidth, delay, bottleneck link speed, queue discipline and its size, etc.) we try to be as realistic as possible to better reflect the current Internet settings. While keeping the parameters reasonably realistic, we also cross check the protocol functions over a wider range of parameters to explore the boundaries in terms of its functional capability. Second, just like other congestion control protocols, we create a group of scenarios that can evaluate the protocol's performance. As mentioned earlier, one of the consensus in developing media congestion control protocols include the throughput smoothness property and a relatively quick responsiveness under a sudden network change while maintaining a reasonable degree of fairness[1] with the standard TCP flows, and we construct a set of simulations to validate these properties.

Note the detailed validation steps on the protocol's functional components are given in Appendix A.2. As explained in the previous chapter, there are several fundamental but important functional blocks to be validated before conducting various performance evaluations.

These are:

[1] Further discussions on the fairness issues will be made in Section 4.2

- AckVec mechanism
- ALI computation
- *cwnd* computation
- TFWC Timer (to update RTT and RTO)

It is to be certain that each component generates correct input/output sequences as specified in the protocol description for us to rely on the complex simulation results. Therefore, in this chapter, we focus more on the performance aspects of the TFWC protocol, leaving the verification of each unit in Appendix A.

To summarize, it is critical, when conducting performance analysis, that we choose right parameters and environments with a correct implementation. By all means, we also need to have right metrics when comparing the performance with others. Satisfying all points discussed has been a challenge.

4.1.1 Simulation Environments

Generally speaking, congestion control algorithm uses a packet loss as an indication of network congestion assuming routers do not mark packets explicitly. Thus, different loss models in the simulations can lead to different results. It could also mean a *requirement*, when evaluating them, to have either a manually constructed congestion or a contrived congestion environment. When the bottleneck node operates an active queue management scheme (e.g., RED, REM, and etc.), the losses can be induced with a certain random probability. This thesis will consider all cases as below:

- Constructed Losses

  In this model, we introduce a group of periodic and deterministic losses per specific test scenarios. This loss model is mainly used for validating the TFWC protocol functional blocks. To control the loss rate explicitly and manually, we intentionally disturb packet transmission at the sender. For example, we drop sending one packet out of every hundred transmission to emulate 1% (deterministic) loss event rate. We then verify if the protocol's functions work correctly as we have designed (e.g., in this case, ALI should indicate 100 at all times as shown in Figure A.2(a)).

- Contrived Losses

  In this model, the packet losses are contrived by a higher degree of statistical multiplexing in the simulations, creating a real-world-like congestion environment. To this end, we place a group of background traffic sources in the simulations, for instance, parallel TCP sources, or UDP On/Off CBR sources. This model is primarily used for evaluating the protocol's performance. These types of losses occur because the bottleneck queue will overflow during the simulation.

- Induced Losses

  In this model, the losses are arbitrarily induced by the bottleneck routers, so-called the active routers (e.g., RED routers). There are many different algorithms to mark packets, but we consider RED queue mainly in this thesis. It is partly to break the simulation phase effect, but also to reflect a better and realistic loss condition. In simulations, the losses can be too deterministic sometimes

when using the drop-tail queue, because the simulator itself is run by a discrete event scheduler, which is less likely to be a real-world condition.

As mentioned earlier, we focus on the latter two loss models in this chapter; the manually constructed loss cases are presented in Appendix A.

After all, a packet loss occurs once the bottleneck queue has filled up, or the bottleneck queue, in the case of an active queue, artificially drops it by a certain marking algorithm. Therefore, setting a queue size is also an important matter in the simulations in reaching a right conclusion. That is, the packet loss is a major indication of a network congestion, and the loss itself is heavily dependent on the types and sizes of queue. In this thesis, we use the BDP as a reference number when setting the queue size. When setting the queue length in the bottleneck, we set it to be equal to the BDP in most cases. However, it is known that the queue size of cheap DSL or cable modems are ill-configured in many cases nowadays[2], we also conduct simulations for a larger queue size as well (e.g., more than just a single BDP).

4.1.2 Performance Metrics

The performance evaluation of TFWC protocol depends upon the metrics that we choose, and there are numerous performance metrics available, of course, for different purposes. In our case, we choose the metrics that can describe the protocol's functions and goals. There has already been a handful of literatures that have defined useful performance metrics, and we have selected among them. In fact, we choose the performance metrics that the TFRC has used. These are:

- fairness
- smoothness
- stability
- responsiveness

The above metrics are obtained based on a fundamental and commonly used metric, *throughput*. The throughput can be measured and averaged on a certain time-scale (δ), making it to be a time averaged throughput ($\mathcal{T}$). We can then define the time averaged throughput as:

$$\mathcal{T}_{t+\delta} = \frac{\sum_{t}^{t+\delta} b_i}{\delta} \qquad (bits/sec)\ , \tag{4.1}$$

where b_i is the total number of bits received at the receiver in a time interval between t and $t + \delta$. Note the time averaged throughput can have different meanings when used with different time scales. Generally we average the throughput using the RTT value on the network, so it can represent a received bit rate per RTT. However, when the RTT is extremely short (e.g., in an LAN environment having less than a millisecond of RTT), then we use a value little higher than this, because in this case the time averaged throughput would not be much different to an instantaneous throughput – e.g., one would see a bunch of spikes when averaging the throughput with such a small RTT value, making it hard to analyze

[2]Surprisingly, the queue size in these boxes are set extremely large (e.g., much larger than the BDP size).

after all. For the rest of thesis, we mean it to be a time averaged throughput when mentioning simply a throughput.

So, using the fundamental metric, time averaged throughput, we evaluate and discuss the four performance metrics in the next sections.

4.2 Fairness

There has been a consensus in the research community (initiated from the Kelly's seminal work [28]) that the newly designed congestion control protocol should be better being TCP-friendly (or TCP-compatible), where this *friendliness* and *compatibility* are often measured by how much it can achieve throughput fairness to the standard TCP. This means the average throughput per flow should be statistically the same as or, at least, less than the TCP's per flow average throughput. Having satisfied this condition, we say the protocol is TCP-friendly. In short, it requires a *proportional* fairness in terms of allocating shared resources. On the other hand, there has been another discussion at the IRTF recently [48] in that the *friendliness* should not be the goal when designing congestion control protocols and its definition is not useful in the current Internet usage patterns, suggesting a new type of fairness concept, so-called *relative* flow fairness [24]. The arguments are sometimes applications nowadays open multiple connections (e.g., peer-to-peer (P2P) applications) to receive more throughput, or it is simply inappropriate to maintain the fairness level with the standard TCP in high BDP network environments. Thus, the fairness concept for such an application would be better a per-application measure rather than per-flow, or it should be designed more aggressively than the standard TCP in a high BDP network. However, we examine the flow-level fairness in a broadband Internet environment, assuming an application has a single flow. Although some non-interactive P2P applications often open multiple connections, the target application for TFWC protocol is the real-time interactive streaming applications which normally open single connection between a sender and receiver, hence the fairness study still being necessary. Nonetheless, our ultimate goal is not to reach the perfect fairness with the TCP sources, but to understand the dynamics of the fairness when competing with the TCP flows in various network environments. At the same time, if we can achieve and prove our proposed mechanism can maintain a better fairness than the existing mechanisms (e.g., TFRC) across different network settings, it would be possible to say that we are in a better position to realize the *relative* fairness, for example, by assigning a weighting factor per flow accordingly when necessary. What we cannot forgive is a situation where all TCP flows are starved, or similarly all TFRC/TFWC flows are starved. So, for now, our objective is to achieve a better throughput fairness measured by the *fairness index* (θ) as defined below:

$$\text{Fairness Index } (\theta) = \frac{\sum_{i=1}^{n} \mathcal{T}_{tcp_i}}{\sum_{i=1}^{n} \mathcal{T}_{tcp_i} + \sum_{j=1}^{n} \mathcal{T}_j} \ , \tag{4.2}$$

where $\mathcal{T}_{tcp_i}$ is the throughput of the i^{th} TCP source, and $\mathcal{T}_j$ is the throughput of the j^{th} TFRC (or TFWC) source. It is the aggregated throughput of n TCP flows divided by the total aggregated throughput of all TCP and TFRC (or TCP and TFWC) flows. If θ is 0.5, then it indicates TCP perfectly shares the link with TFRC (or TFWC). As θ approaches 1, TFRC (or TFWC) sources are starved by

TCP sources[3].

The results in this section use a dumbbell topology as shown in Figure 4.1. We vary the number of sources from 1 to 100 and use the 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 10 ms link delay, and the access link delay was chosen randomly between 0.1 ms to 2.0 ms. There are reverse path TCP flows included in the simulations; these serve both to reduce simulator phase effects and reveal any issues that might be caused by TFWC `Ack` compression (which TFRC does not suffer from). With this scenario, the approximated average RTT is 20∼40 ms roughly − a fairly typical DSL RTT to a local ISP. For the bottleneck queue type, we used the drop-tail and RED routers respectively for the evaluation. We set the buffer size to be roughly a BDP when with the drop-tail queue: this allows TCP and TFWC's `Ack`-clocking to perform correctly. As mentioned, with the small windows, the `Ack`-clock will not function properly. On the other hand, with the RED routers, we set the size a little larger than the BDP because the RED algorithm will start marking packets before it reaches the full capacity. Nevertheless, for both cases, we settle a fixed, lower bound in the minimum queue size. Consider the following two extreme cases. First, a fast link with an extremely short delay, such as a LAN environment. Let's assume a link that has 20 Mb/s bandwidth and 0.5 ms RTT. It then yields the BDP roughly 1250 bytes; it is only *one* packet roughly. Second, assume an ISDN-like link where the bandwidth is 100 Kb/s with an RTT of 100 ms, which also result in the BDP of one packet. This kind of short queues may end up dropping all packets associated with single window, making the connection to back off suddenly (or, even in the worse case, to stop sending all at once). It would not be a serious problem when the link is highly multiplexed because the packets are well spaced among flows, but the problem will be aggravated especially with a low statistically multiplexed link: it would exhibit unstable rates, resulting in uneven and unfair sending rates. Therefore, we set the minimum queue size to 15 packets (15 Kbytes) for the rest of the simulations in this section. Finally, we evaluate the *fairness index* (Equation (4.2)) in a steady state environment for the drop-tail and RED routers, respectively.

4.2.1 Fairness using DropTail router

Figure 4.3 shows the protocol fairness for (a) TCP vs TFRC and (b) TCP vs TFWC. We used the simulation topology as illustrated in Figure 4.1 with the fixed bottleneck delay of 20 ms, making the end-to-end delay 40 ms roughly. These are fairly similar situations where users experience behind the

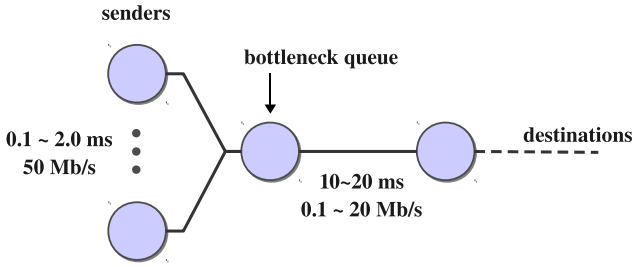


Figure 4.1: Simulation Topology

[3]In *ns-2* simulations, we have used `TCP/Sack1 Agent` throughout the thesis. In *ns-2*, it is the closest implementation to the standard TCP in the modern Linux kernel, which supports the "*Fast Re-transmission*" feature.

DSL or cable Internet. The x-axis shows the bottleneck bandwidth, the y-axis gives the number of flows sharing the bottleneck[4], and the z-axis stands for the *fairness index (θ)*. We run the same set of simulations 5 times with different random seed numbers (to randomize the FTP start-up time and the access link delay per sources). The results show the TFRC protocol fairness is largely dependent upon the bottleneck bandwidth. If the bottleneck bandwidth is less than 2 Mb/s, then we can hardly conclude the TFRC is fair to TCP, no matter what the number of sources is: $\theta \cong 0.1$.

Equally with a small number of TFRC flows, the result shows values where $\theta > 0.85$, indicating TFRC is starved. Although TFWC is not perfect, it does significantly better, with no value of θ less than 0.35 or greater than 0.75. However, both protocols tend to receive more average throughput than the competing TCP flows when the bottleneck bandwidth is small and highly multiplexed – it does after all use the same equation.

We have run the simulations with different queue sizes at the bottleneck (half the BDP, two times of the BDP, and four times of the BDP) and with a different RTT value (e.g., 20 ms), which all exhibited the similar results as in Figure 4.3.

4.2.2 Fairness using RED router

We evaluate the protocol fairness as the same network environments described in section 4.2.1, but place the RED router in the bottleneck. The RED parameters are set as recommended in [29], which are:

- max_{th}: $\frac{\text{queue size}}{2}$
- min_{th}: $\frac{max_{th}}{3}$
- max_p: $2 * \text{estimated loss rate on that link}$
- w_q: ensure the lower bound (0.002) used for EWMA coefficient

The notations used here follow ones in the original RED paper [32]: max_{th} stands for the maximum threshold value, min_{th} the minimum threshold, and max_p the maximum probability that the two consecutive packets will be dropped. As widely known, the target is to control the *average* queue size between max_{th} and min_{th} while accommodating transient congestion. In the simulator, we set to mark/-drop packets randomly (i.e., drop packets with a random probability by setting "drop_rand_" as true and "drop_tail_" to false.). Also, we have enabled the "gentle_" parameter so the packet drop probability varies from max_p to 1 when the *average* queue size exceeds max_{th} (but only up to $2 * max_{th}$). The marking probability with gentle option enabled is illustrated in Figure 4.2.

There is another parameter that can affect the average queue size calculation: queue weight factor (w_q). This weight factor servers as a coefficient when calculating the average queue size in Exponentially-Weighted Moving Average (EWMA) fashion. That is when w_q is set too low, the calculated average queue does not reflect the current average queue size effectively, whereas if w_q is too large, then the EWMA function will not filter out the transient congestion in the queue. The RED papers ([29], [32]) suggest it to be greater than 0.002 at all times. In fact, in the simulator, we compute w_q dynamically depending upon the link status; after all, w_q can be represented as a function of the bandwidth

[4] It indicates the number of sources for a single type of flow, so the number 50 would mean there are a total of 50 TCP sources *and* the same number of competing TFRC or TFWC sources in the simulation.

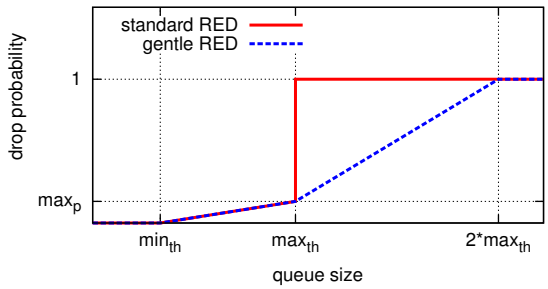


Figure 4.2: Packet marking probability for the standard RED and gentle RED

and the link delay [32]. The RED implementation in *ns-2* simulator, for example, computes the weight factor as:

$$w_q = 1.0 - exp\left(\frac{-1.0}{10 * t_{RTT} * \mathsf{ptc}}\right) \quad \text{, where } \mathsf{ptc}^5 = \frac{\mathsf{link\ bandwidth}}{\mathsf{packet\ size}}. \tag{4.3}$$

Figure 4.4 shows the protocol fairness (a) TCP vs TFRC and similarly (b) TCP vs TFWC using RED router in the bottleneck. Although TFRC was not completely starved as it did with the drop-tail queue, the fairness of TFRC protocol is still as bad as the drop-tail case, whereas TFWC protocol maintained the index (θ) perfectly fair in all evaluation cases. From Figure 4.3 and Figure 4.4, it strongly appears that TFRC can starve the standard TCP sources significantly when the bottleneck bandwidth is less than 2 Mb/s. This is because TFRC does not cut the sending rate in half immediately upon a congestion signal (in which TCP halves the send window in a similar situation), making it hard for TCP packets to get in the queue. Also, although the "$\mathsf{RTO} = 4 * \mathsf{RTT}$" rule was derived after extensive simulations and Internet experiments which actually proved it a little more conservative measure than the TCP scheme, it is still a question whether this simplification is still valid in the current Internet settings: different RTO timers will result in different behavior in a heavily congested network.

As we did in Section 4.2.1, we have conducted the simulations with different queue sizes settings at the bottleneck (half the BDP, two times of the BDP, and four times of the BDP), which all showed the similar results as in Figure 4.4. We also have run the all simulations with a little shorter RTT value (e.g., 20 ms), and those results have turned out to be quite similar, too.

Overall, the nice and stable fairness feature can bring a great advantage to the application writers providing a precise point of how much the application flow can achieve fair share of the link proportionally, moreover, how correctly it can.

4.3 Stability

In general, there are pros and cons for the choice between the rate-based congestion control and the TCP-like window-based congestion control schemes. One of the obvious advantages being rate-based is it can generate much smoother throughput than the TCP-like window-based congestion control by avoiding

[5]The ptc in *ns-2*, so-called the packet-time constant, represents the max number of average packets per second which can be placed on that link (i.e., the max number of outstanding packets on the link). This equation, therefore, promises the computed w_q value is reasonably large in order to reflect the current average queue size effectively.

the saw-tooth like behavior in the send rate. So, are we going to loose the nice smoothness property at all by re-introducing the TCP-like `Ack`-clocking in TFWC? To answer this we use Coefficient of Variation (CoV) as the performance metric for smoothness of the steady-state sending rates. The CoV is the throughput variation experienced by a flow over time. Let T_i be the sending rate of a flow during the i^{th} time interval, then CoV of that flow can be defined as:

$$\mathsf{CoV}(\Delta) \;=\; \frac{\sqrt{\frac{1}{k}\sum_{i=1}^{k}(T_i - \overline{T})^2}}{\overline{T}}\,, \tag{4.4}$$

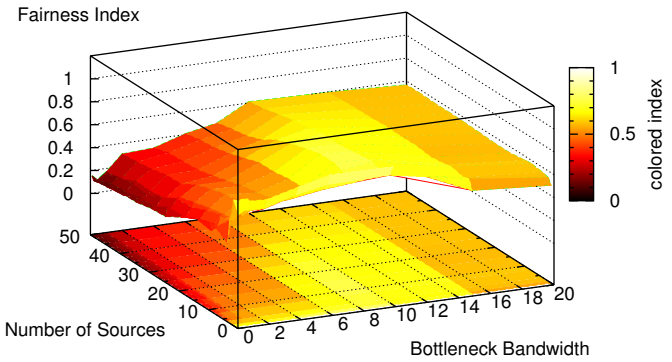
where $\overline{T}$ is the average throughput over time and k is the number of time intervals. The time interval of 0.5 second is used for the measurement, a reasonable timescale for changing codec parameters for real-time traffic.

Figure 4.5 depicts the mean CoV (Δ) of both the TFRC and TFWC flows when used with the drop-tail queue at the bottleneck. The CoV results here use the same simulations conducted in Section 4.2.1. So, the coordinate description is same as in Figure 4.3. It showed the CoV of both TFRC and TFWC is quite small ($\Delta < 0.3$) in all cases with the mention of the following exceptions:

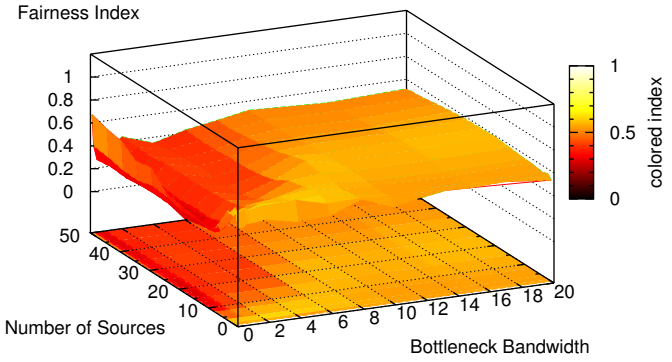
- TFRC's CoV is extremely high ($\Delta > 0.9$) when the bottleneck bandwidth is small.
- TFWC's CoV is relatively high ($0.4 < \Delta < 0.8$) when the bottleneck bandwidth is small.

For the TFRC's CoV, because the network is highly congested with many flows competing for a tiny bandwidth, the fluctuation on the individual TFRC's throughput made it high. Recall that, as shown in Figure 4.3(b), the *aggregated* and average TFRC throughput almost completely starved the TCP throughput in these environments. Although the TFWC's CoV is somewhat high in the same regime, the average TFWC throughput shared fairly with the TCP flows with a little variation, which can be acceptable.

We also evaluated CoV (Δ) using RED queue at the bottleneck as shown in Figure 4.6. The results here are similar to the drop-tail environments with a little improvement on the stability for the TFRC mechanism (no value of Δ greater than 0.8.). Figure 4.5 and Figure 4.6 showed that the stability of both protocols (TFRC and TFWC) seemed a little high when many flows are competing for a tiny bandwidth. Each flow, therefore, was likely to be operated by the timeout behavior for most of its life time in this regime. We believe that the little instability in these cases is due to the artifact of the TCP throughput equation, in that the TCP's timeout behavior was not modeled accurately in the original TCP equation (i.e., it does not look like a protocol artifact).

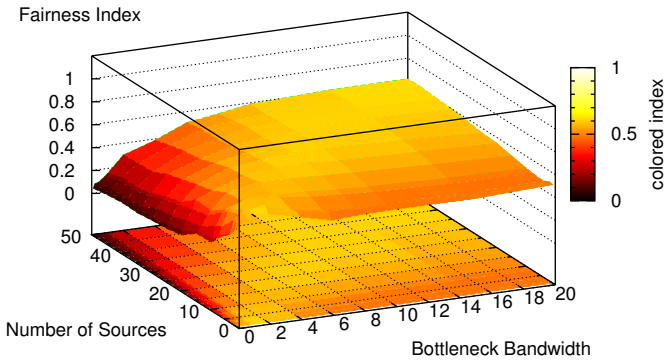


(a) Fairness (TCP and TFRC)

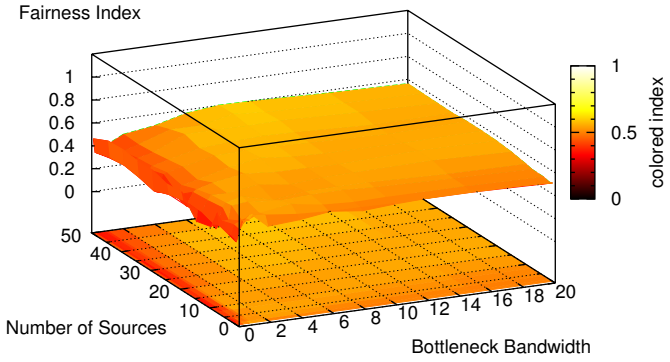


(b) Fairness (TCP and TFWC)

Figure 4.3: Fairness Comparison using DropTail queue, $t_{RTT} \cong 40$ ms

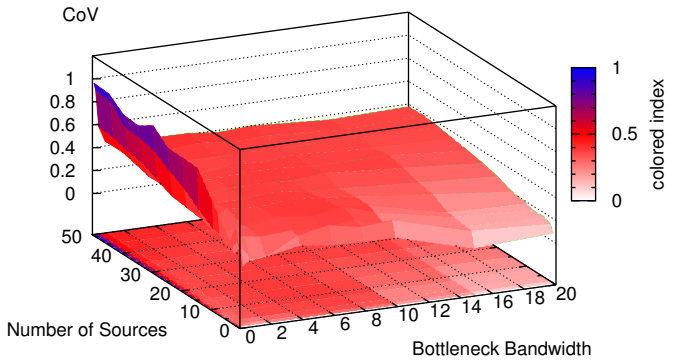


(a) Fairness (TCP and TFRC)

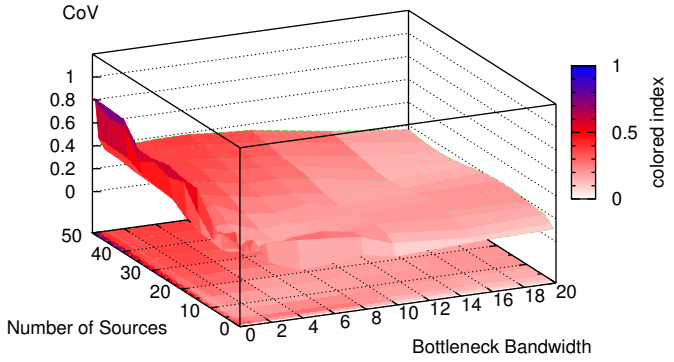


(b) Fairness (TCP and TFWC)

Figure 4.4: Fairness Comparison using RED queue, $t_{RTT} \cong 40$ ms

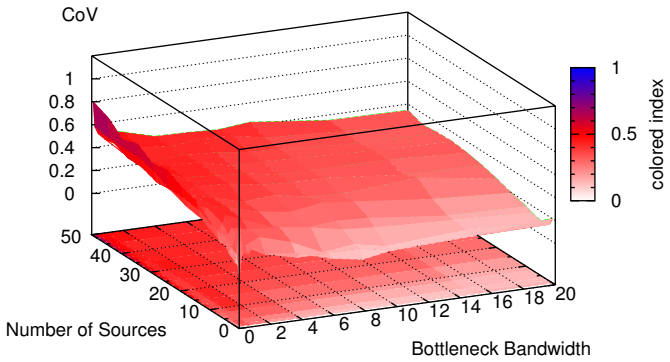


(a) TFRC's CoV

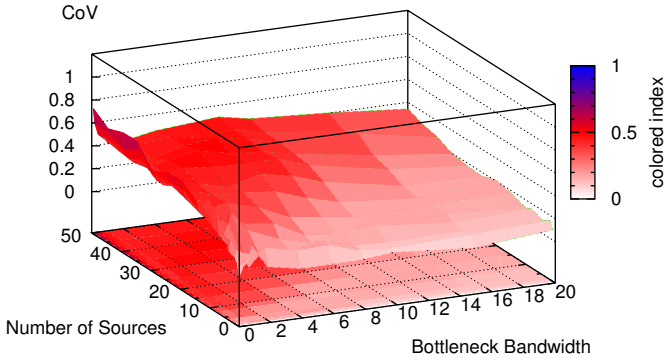


(b) TFWC's CoV

Figure 4.5: Coefficient of Variation for TFRC and TFWC using DropTail queue



(a) TFRC's CoV



(b) TFWC's CoV

Figure 4.6: Coefficient of Variation for TFRC and TFWC using RED queue

4.4 Smoothness

As studied in [62], the long-term smoothness of a traffic is mainly determined by packet loss patterns and it tends to follow the same distribution at large time-scale (more than 100 RTT), regardless of which congestion control scheme is applied. In this section, we look in more detail at a single simulation, similar to those in Figure 4.3 and Figure 4.4. In this example, we have chosen 8 TFRC (or TFWC) competing with the same number of TCP sources for a bottleneck link with capacity 5 Mb/s. There is reverse TCP traffic, partly to reduce the phase effects, but more importantly to perturb the smoothness of the TFWC `Ack` flow. Unlike TFRC, TFWC's smoothness is reduced if the `Ack` flow is disrupted. The bottleneck queue is set to 300 packets, as illustrated in Figure 4.7.

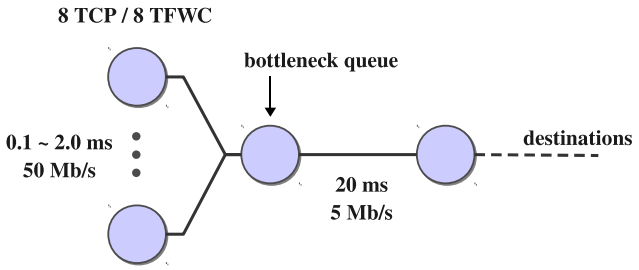


Figure 4.7: Simulation Topology

In such environment, the average *cwnd* size per flow can be approximated as:

$$\bar{\mu}_{\mathsf{cwnd}} = \frac{\max\{\mathsf{pkt}_{\mathsf{in\text{-}flight}}\}}{n_{\mathsf{total}}} = \frac{\mathsf{BDP} + Q_{\mathsf{size}}}{n_{\mathsf{tcp}} + n_{\mathsf{tfwc}}}, \tag{4.5}$$

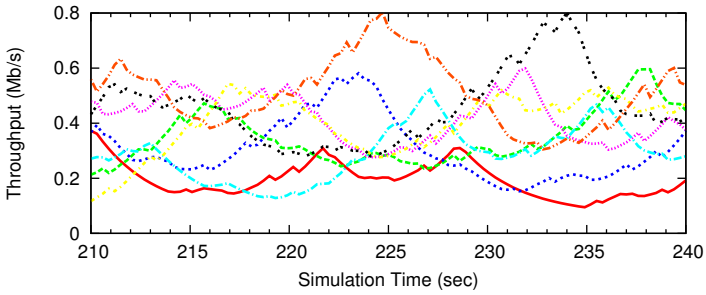
where:

- $\max\{\mathsf{pkt}_{\mathsf{in\text{-}flight}}\}$: the maximum value of the outstanding packets on that link
- Q_{size}: the bottleneck queue size
- n_{total}: the total number of connections
- n_{tcp}: the total number of TCP connections
- n_{tfwc}: the total number of TFWC connections

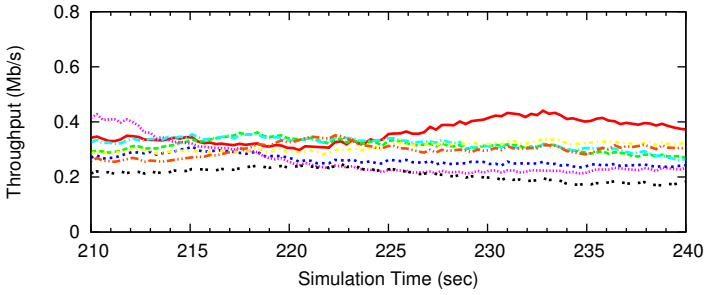
Using Equation (4.5), we can obtain the average *cwnd* size per flow ($\bar{\mu}_{\mathsf{cwnd}}$) in this test environment: roughly 15 packets for this case. We can say $\bar{\mu}_{\mathsf{cwnd}}$ is reasonably large enough to make `Ack`-clocking the dominant TFWC mechanism in this environment and ensure it does not switch into the rate-based timer mode so frequently. The smoothness results are shown in Figure 4.8. The degree of TFWC protocol's smoothness is quite similar to what TFRC can perform as seen in Figure 4.8(b) and Figure 4.8(c).

4.5 Responsiveness

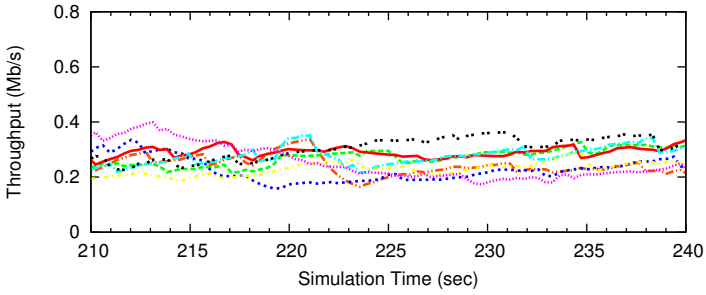
It is of course possible to make equation-based congestion control arbitrarily smooth by using long time constants in the average loss interval filter (i.e., by increasing the number of history elements as shown in Figure 3.1), but this would come at the expense of responsiveness to real changes in network conditions. To better illustrate TFWC's responsiveness, we now study how the protocols respond to the



(a) TCP's Abrupt Sending Rate Change

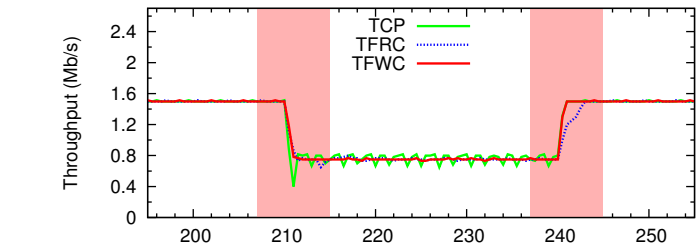


(b) TFRC Smoothness

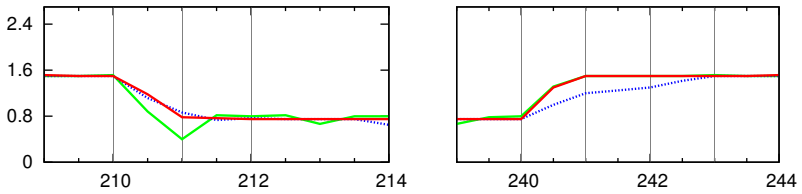


(c) TFWC Smoothness

Figure 4.8: TCP/TFRC/TFWC Protocol Smoothness



(a) Impulse Response



(b) Zoom-In (high-lighted area)

Figure 4.9: TCP/TFRC/TFWC Responsiveness

impulse response as a form of On-and-Off Constant Bit Rate (CBR) source. The simulation model is similar to those used in [22]. In the simulation, we turn a CBR source on and off with sending rate half of the bottleneck bandwidth. The results shown in Figure 4.9 are from the bottleneck bandwidth of 1.5 Mb/s, average RTT of 60 ms. The drop-tail bottleneck queue size is arbitrary large so that each TCP, TFRC, or TFWC flow can fill the bandwidth fully. One TCP source is competing with one CBR source and similarly one TFRC/TFWC with a CBR source. CBR is turned on at 160 sec and runs 30 sec duration, and similarly turned on 210 and 270 sec with 30 sec running time, respectively. As we can see in Figure 4.9, TFWC's responsiveness is similar to TFRC, and TCP has little ripple in throughput when sudden network condition changes. If we zoom in a little more as in Figure 4.9(b), TFWC is faster than TFRC when filling up the available bandwidth, for reasons that TFWC can cope with the speed of `Ack`-clock whereas TFRC is bounded by that of an RTT at least.

4.6 Conclusion

In this chapter, we discussed the performance of TFWC protocol, a simple but novel congestion control mechanism for streaming multimedia applications. We showed that TFWC can outperform TFRC in various network settings through extensive *ns-2* simulations. A rate-based congestion control faces practical implementation issues with timer granularity when RTT is small, like a local LAN environment, whereas TFWC works just like TCP with its `Ack`-clocking mechanisms and does not suffer from this limitation, giving better fairness. Similar limitations are well pointed out in [56] and [62]. Also, TFRC can starve almost all TCP traffic under a DSL-like environment with low level of statistical multiplexed link, whereas TFWC can co-exist with TCP nicely in the same conditions. As a result, rate-based congestion control can starve or be starved with other types of congestion control mechanisms, especially TCP, which is not desirable at all in the Internet.

In this context, TFWC is more realistic and practical when it is implemented in a real-world ap-

plication without bearing all the troubles as introduced in Chapter 2, and it can still retain the nice properties that the rate-based congestion control mechanisms have: favorable protocol smoothness, fairness, and responsiveness for such an application. Therefore, if we have an option for multimedia streaming applications, then this certainly seems to be a better choice than any other rate-based protocols.

To summarize the lessons learnt in terms of the performance metrics introduce in Section 4.1.2:

1. Fairness:
   - Figure 4.3 shows that TFRC significantly starved TCP sources when the bottleneck bandwidth is less than 2 Mb/s ($\theta \approx 0.1$), whereas TFWC was overtaken slightly in the same conditions ($\theta \approx 0.6 \sim 0.7$).
   - Figure 4.4 shows that TFWC was really fair in all simulation cases ($\theta \approx 0.5$) using the RED queue at the bottleneck, whereas TFRC still starved TCP sources when the bottleneck bandwidth is small ($\theta \approx 0.1$).

   $\Rightarrow$ These results explain that TFWC is less susceptible in reaching fair share of the link despite the bottleneck bandwidth being small. In the high bandwidth regime, TFWC was even fairer than TFRC using DropTail queue, though they both exhibited quite fair competing with TCP flows with the RED queue at the bottleneck.

2. Stability
   - Both TFWC and TFRC, the CoV developed markedly high values in the low bandwidth regime (less than 1 Mb/s) as the number of sources increased. In this zone (low bandwidth with extremely large number of sources), because the loss rate will be high, the second term of the denominator in the complex TCP equation (Equation (2.2)) will be dominant (i.e., $t_{RTO}\, 3\sqrt{\frac{3p}{8}}p\left(1+32p^2\right)$), where the timeout behavior comes into play throughout the simulation. However, the TCP equation did not model the timeout behaviors accurately. Therefore, it appears to be a limitation of the equation, instead of the protocols themselves.

   $\Rightarrow$ Figure 4.5 and Figure 4.6 show that there to be no case where TFWC performs worse than TFRC (i.e., they are not substantially different).

3. Smoothness
   - With *cwnd* jitter enabled, the smoothness result presented in Figure 4.8 shows that TFWC and TFRC flows are equally smooth (and much more smoother than those of TCP).

4. Responsiveness
   - Figure 4.9 shows that TFWC decreased its sending rate as fast as TFRC when a CBR source entered the network, whereas TFWC increased a little faster than TFRC when filling up the pipe in the *ns-2* simulation.

From these results and analysis, we conclude that TFWC actually demonstrated a potential to be used for a congestion control mechanism in the real-world (i.e., no case found where TFWC significantly performed badly than TFRC). In the next chapters, we discuss and evaluate the TFWC protocol implementations and performance when applied to a real application, the Vic tool [12].

Chapter 5

Congestion Control for Interactive Video Streams

This chapter describes the implementation aspects of TFWC and TFRC in the Vic tool. We give a brief overview on the Vic transmission system, and show how we have integrated those congestion control mechanisms into the system.

5.1 Introduction

The first real-world applications to provide interactive video streams over the Internet (by that time, MBone [9]) were the INRIA Video Conferencing System (*ivs*) [6] and the Network Video (*nv*) [33] tool. These tools had initially aimed at providing video streaming services at a relatively low bit rate based on H.261 codec family, where *nv* tool only provided the video streaming. A little later, Vic [12] emerged, reflecting the pros and cons in its design from these two applications, to support many features, for example [49]:

- multiple network abstractions
- hardware-based codec support
- flexible and extendible object-oriented architecture
- various video compression schemes

However, these early-day video conferencing tools lack the support of congestion control mechanisms as those control algorithms were not cooked well in the days. As we now have the decent options for choosing the congestion control algorithms (i.e., our proposed mechanism and the IETF standard), we have taken in one of the video conferencing tools for our research tool, Vic [12], and have rigorously evaluated the congestion control mechanisms using the chosen system. In this section, we give an overview of the Vic and its architecture, then describe our implementation aspects of two congestion control mechanisms: TFWC and TFRC.

5.2 Vic Overview

The development and deployment of audio/video conferencing tools was initially fostered by the introduction of IP multicast technology back in the 1990s. Since then, among other tools [6, 11, 33, 58], the

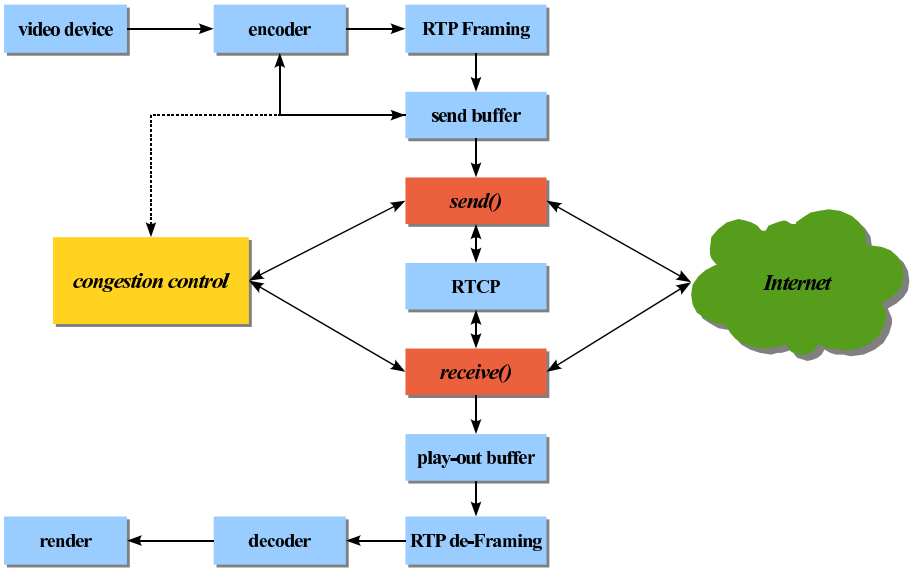


Figure 5.1: Vic Architecture: high-level overview

Vic tool has continually developed until recently [1, 3], which is one of the reasons for choosing it as our development platform. The Vic tool is a real-time, multimedia application for video conferencing over the Internet, and supports various multimedia codecs, such as H.26x, DV, and MPEG. It is also designed with a flexible and extensible architecture to support heterogeneous environments and configurations. For instance, in a rich bandwidth environment, multi-megabit full-motion JPEG streams can be sourced using hardware acceleration, while in low bandwidth environments like an ISDN environment, aggressive low bit-rate coding can be carried out using software alone. Due to its high flexibility to extend features, Vic can be an ideal system when testing new transmission control algorithms, like TFWC or TFRC. The object-oriented modular design allows us to seamlessly integrate two congestion control mechanisms, creating an excellent system environment to conduct various experiments for the new mechanisms in a real network. Our application-level implementations stand out with other real-world implementations (e.g., kernel implementation), in that the higher layer implementation relieves the requirement for re-building a kernel or installing a new kernel module, but instead helps easier deployment of the applications as provided. In the next section, we briefly overview the Vic architecture.

5.2.1 Vic Architecture

The high-level data flow of the Vic is illustrated in Figure 5.1. As the video device grabs frames, the encoder performs block-based image compression, Real-Time Transport Protocol (RTP) framing, and transmission of the packets. On the receiver side, as the packet arrives, it stores them into the play-out buffer, de-framing RTP packets, and then finally decodes/renders the image frames. The congestion control mechanisms can be sit in between the sender and receiver in which it provides the calculated sending rates (*cwnd* for TFWC and inter-packet interval for TFRC), for example, to the `send()` method at the sender. We uses the RTCP Extended Reports (RTCP XR) [34] to carry the information needed by the congestion control mechanisms, while RTP delivers the data packets. For example, the data receiver reports the feedback messages, carrying AckVec messages, to the sender using RTCP XR packets.

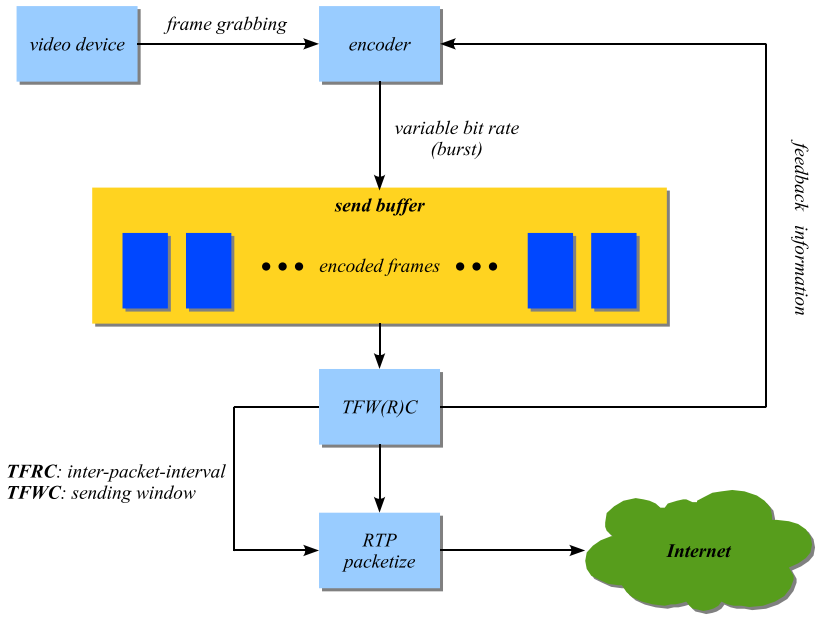


Figure 5.2: Overview of Vic Sender

Likewise, the data sender provides the AoA messages once in a while using the RTCP XR packet. Traditionally, RTP data packets and RTP Control Protocol (RTCP) control packets have been run on separate User Datagram Protocol (UDP) ports. Vic also implements two separate channels, namely DataHandler and ControlHandler, to convey data packets and control packets, respectively, on separate ports. We will describe the full packet structure in Section 5.3.1.

5.2.2 Vic Sender

A general overview of the sender functionality is shown in Figure 5.2. At the sender, there are three components that determine the overall system performance: grabbing device, encoder, and transmission module. As Vic is a single threaded application, the interactions among these parts, therefore, greatly influence the overall behavior of the packet transmission system. The ideal system in the steady state at the sender is, then, to control the send buffer to:

- avoid buffer under-run
  – buffer containing less packets than it would have been able to send.

- avoid buffer over-run
  – buffer containing more packets than it can deliver in time.

To re-produce and compare the same experiments using different network parameters and congestion control mechanisms, we have implemented a `FileGrabber`, which loads pre-recorded image sequences (e.g., sources introduced in Appendix B) into so that the encoder can parse the frames from memory for its necessary operation. Once the encoder performs its task (compression/encoding), the encoded frames are inserted into the send buffer, from where the transmission system can perform RTP packetization and transmission of the packets in a periodic manner.

5.2.3 Vic Receiver

The Vic receiver's primary task is to receive the transmitted packets, perform RTP de-framing, decode them, and finally display on screen. As the packets arrive at the receiver, the network object dispatches them to the de-multiplexer which implements the bulk of RTP processes. Then the packets are de-multiplexed to the decoder, which decodes the frame and displays on to an output device.

We have presented the important fundamentals of the Vic system in this section. The rest of the sections describe in more detail how the congestion control mechanisms have fitted to the Vic application.

5.3 Congestion Control in Vic tool

We have implemented TFWC and TFRC in Vic so that they can regulate the overall transmission rate appropriately. We explain how we have realized TFWC integration into the Vic transmission system, then the TFRC description is given later. In fact, we have used exactly the same underlying transmission architecture for both protocols by converting TFRC to a sender-based protocol.

5.3.1 Packet Format

TFWC operates based on `Ack` flow. Although it seems obvious using RTP packet structure to convey data packets, we cannot simply re-use RTCP packets (e.g., RTCP Sender Report, RTCP Receiver Report, etc) as a means for carrying `Ack` messages. In general, those RTCP packets are very coarse-grained in terms of timing, which does not satisfy our frequency requirement (i.e., one `Ack` message generation upon reception of every data packet). Therefore, we choose an alternative packet format that satisfies the requirements of `Ack` messages[1].

To implement the `Ack` messages in Vic, we use the IETF standard, RTCP XR [34], meaning AckVec and AoA messages are carried by the RTCP XR packets (in short *XR packet* from now on). The XR format is shown in Table 5.1, where BT stands for the *Block Type*.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      BT       | type-specific |         block length          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:             type-specific block contents                      :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Table 5.1: RTCP XR Format [34]

There are a number of *block types* defined in the XR packet specification. Among them, we have selected the Block Type 1, *Loss RLE Report Block*, to carry the AckVec and AoA messages. This block type permits detailed reporting of individual packet receipt and loss events. The block format is shown in Table 5.2, whereby `begin_seq` indicates the starting sequence number that this `Ack` is reporting, `end_seq` the ending sequence number that this `Ack` is reporting, and then the rest of *chunks* build AckVec as a bit vector format (we give a full depth detail on AckVec implementation in the next section.).

[1]RTCP feedback mechanisms itself is, therefore, insufficient for congestion control purposes as it is likely to operate at much slower timescales than other transport layer feedback mechanisms (that usually operate in the order of RTT), which our proposed TFWC desperately desires.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     BT=1      | rsvd. |   T   |         block length          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         SSRC of source                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          begin_seq            |             end_seq           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          chunk 1              |             chunk 2           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                               ...                             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          chunk n-1            |             chunk n           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Table 5.2: RTCP XR Loss RLE Report Block Format [34]

It is worthwhile pointing out that there is a rate limit for RTCP packets transmission [59]. The control traffic should be limited to a small and known fraction of the session bandwidth, so that the primary function of the transport protocol to carry data is not impaired significantly. In order to meet this bandwidth requirement while being able to provide immediate feedback, a recent standard has been proposed at IETF [40], which defined an extension to the Audio-Visual Profile (AVP). This early feedback profile (AVPF) maintains the AVP bandwidth constraints for RTCP and preserves scalability to large groups. This *reduced-size* XR reports can certainly help in preventing the network from being overwhelmed by control packets. However, these reduced control packets are likely filtered out at most current Network Address Translation (NAT) devices or middle boxes, preventing RTP sessions establishing correctly. In this thesis, we leave these issues aside, and adopt RTCP XR Block Type 1 to convey our `Ack` messages as it serves perfectly well within our design scope[2]. It also makes sense as our congestion control mechanism currently supports unicast multimedia applications, which does not require a tremendous number of control packets (i.e., the rates for the control packets are well bounded by the 5% rule in our scenarios.). But, again, if these congestion mechanisms are considered for use in a multicast application or an IPTV-like application, the XR report rates should be deemed carefully.

5.3.2 Architecture

In this section, we give an overview of the work flow of the congestion control mechanisms within the Vic tool. For the `Ack`-clock based TFWC protocol, there are two triggering mechanisms to invoke packet transmission at the sender:

- as the sender receives an `Ack` message, it prompts packets to be sent
  – we can say the `Ack`-clock invoked packet transmission
- as an application generates packets, they become due for transmission
  – we can say the application drove packet transmission

This is illustrated in Figure 5.3. Initially the application generates a data packet and sends it to the receiver immediately. Then, as the `Ack` arrives, it updates *cwnd* and RTT, and sends the next available

[2]According to [59], the control packet rate should not exceed 5% of those for data packet rates. With 1000 bytes RTP data packet and 40 bytes of RTCP packet, the 3.75% limitation is large enough for a receiver to send an RTCP packet at every other data packet interval.

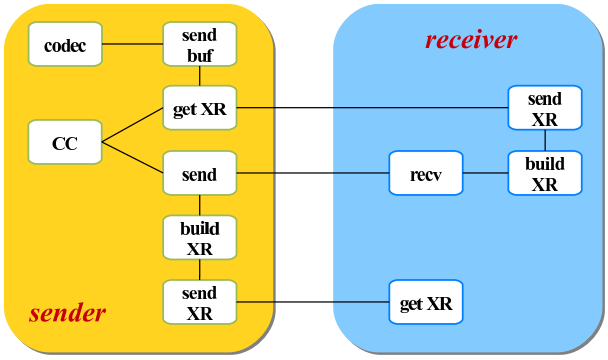


Figure 5.3: High-level Architecture of Congestion Control in the Vic tool

data packets, and so on, where it can be seen as the application-driven packet transmission. At the later stage of transmission, as the sender receives `Ack` message, it updates *cwnd* and RTT and check if the updated *cwnd* allows sending packets in the send buffer, where this can be seen as the `Ack`-clock driven packet transmission.

So, what we actually implement is that, as the sender produces packets, it first checks the control channel to see if any XR packets have arrived at the socket buffer. If they have, the AckVec information is then processed first by the TFWC sender module (in order to quickly update *cwnd* and RTT). If not, the application looks up the send buffer and transmits packets as long as *cwnd* allows. The sender also builds the AoA message using the XR packet once in a while and sends it to the receiver.

So far, we have described the Vic architecture and some of the fundamentals to the congestion control in the Vic tool. In the next sections, we give important implementation aspects of these two congestion control mechanisms.

5.4 Implementation

5.4.1 AckVec – *bit vector*

As described in Section 5.3.1, TFWC AckVec uses the XR packet format shown in Table 5.2. To implement AckVec, unlike the implementation in the simulator, we use a bit vector to indicate the reception of packets using the `chunk` fields in the XR packet. Suppose the receiver builds an AckVec reporting the following packet sequence numbers, (20 19 18 16 15 14 13 11 10), assuming the current AoA is 9. Then the receiver calculates the required number of bits as `num_bits` $=$ `latest_seqno` $-$ `AoA`: in this example, the number of required bits to build the AckVec is $20 - 9 = 11$. Then, it lets left-shift `chunk` indicating 1 if it has received it or 0 if not received, as shown in Figure 5.4. Finally, the receiver sets:

```
begin_seq  =  AoA + 1

  end_seq  =  latest_seqno + 1
```

When the number of required bits exceeds 16 bits (i.e., when the receiver is reporting the feedback more than 16 packets), then it builds additional `chunk`(s). So, the receiver performs bit shifting tasks on receipt of each data packet.

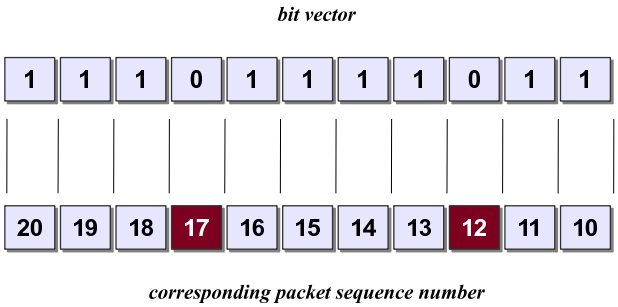


Figure 5.4: AckVec implementation using bit vector

When the sender receives this AckVec message, it extracts the `begin_seq` and `end_seq` from the XR packet, and re-construct AckVec to the real sequence numbers by examining each bit in the `chunk` array.

5.4.2 Packet Re-ordering

In the Internet, packet re-ordering is generally not common. However, re-ordering can potentially harm the performance of *interactive* applications when it happens, although the margin is a little larger for non-interactive streaming applications. There are normally three causes that the sender detects re-ordering, which are:

1. Packets are re-ordered in transit
2. `Ack` messages are re-ordered
3. Both packets and `Acks` are re-ordered

The packet re-ordering can introduce two errors in the processes of congestion control modules, if not properly dealt with. They can be:

- *cwnd* computation error

  When packet re-ordering occurs, the late arriving AckVec (containing the re-ordered packet sequence number) will supersede the AckVec that already arrived at the sender (which does not contain the re-ordered packet sequence number). In this case, both the late and the early AckVec will result in a wrong ALI computation, which eventually can lead to a wrong *cwnd* calculation.

- RTT measurement error

  The standard TCP does not update RTT with the re-ordered packets, simply in order not to reflect them when calculating the new RTT. Furthermore, when the re-ordering is out of the DupAck range, they do not reflect them in a new *cwnd* computation by discarding the excessively disordered packets. Similarly, TFWC and TFRC in our implementation do not update RTTs when packets are disordered.

To address these errors, we have devised a mechanism that can *revert ALI* when the re-ordered packet is received. That is, when the disordered packet triggered a new loss event previously, we revert the ALI and the timestamp for the last lost packet to the earlier state[3].

[3] Assume the disordered packet is received within the three DupAck range.

5.4.3 RTT Measurement – *socket timestamp*

TFRC uses the smoothed RTT directly into the TCP equation when computing the sending rate, whereas TFWC operates on an `Ack`-clock basis in which RTT is implicit. Therefore, in the case of TFRC, the correct and accurate RTT measurement is extremely important to get the right rate. However, we have observed that, due to the OS's clock granularity, the varying overall system load, and various buffering issues, the sampled RTTs did not meet the expected level of precision when the link delay is extremely short (e.g., less than 1 ms in a university LAN environment). In such an environment, the measured RTT demonstrated higher values than it should have been: a considerable drawback for a rate-based protocol. In these short RTT environments, there could be two options to alleviate these side effects by using Unix socket options[4] at the sender and receiver side, respectively, as below:

- Sender Side Option

  `SO_SNDBUF` – when sending packets, they can go into a large kernel buffer and stay there for an arbitrary amount of time *after* timestamped. Therefore, the sampled RTT cannot reflect them accurately when the link delay is so small. To mitigate this side effect, we can shrink the UDP send buffer size adequately, using `SO_SNDBUF` option in the socket.

- Receiver Side Option

  `SO_TIMESTAMP` – likewise, when we compute RTT at the higher layer of application by calling `gettimeofday()`, these values are likely to include the kernel buffering time as well, adding arbitrary delay after all. When RTT is extremely short, these kernel buffering time can go several times greater than the actual RTT. In this case, we can use the kernel timestamp to abate the problem by using `SO_TIMESTAMP` option in the socket.

With the above socket options, the sampled RTT and its smoothed version captured the actual values more accurately, which then resulted in improved stability for TFRC sending rate computation. Moreover, as we mentioned in Section 5.3.2, whenever the Vic sends packets, it looks up the control channel to see if there are any XR packets awaiting in the socket. As the Vic is single threaded, probing to retrieve XR (and to update congestion control state information) would help to obtain more accurate values (*cwnd* and RTT), instead of waiting for the scheduler to be arranged next time.

5.4.4 Packets or Bytes? – *estimating packet size*

Unlike the simulations, the packet size of the real video streams are highly variable. When the packet sizes are small, the control of sending rate by a window that computes available number of *packets* can become ungainly. Instead, if we compute the window in *bytes* then it could give more headrooms when the packet sizes are small. One may argue why don't we develop the TFWC equation (Equation 3.4) that can compute *cwnd* in bytes in the first place, instead of calculating the number of packets? However, recall that the loss event rate (p) in the TFWC equation had to have a measuring unit, which is a fixed size packet, hence the equation yields *cwnd* computation in packets. Therefore, in order to compute *cwnd* in

[4]This Unix socket options can be enabled using `setsockopt()`. For further information, `man 2 getsockopt` on any Unix-like systems.

bytes, we introduce a simple converting formula:

$$\beta \; = \; s * \omega \tag{5.1}$$

where β stands for *cwnd* in bytes, s the average packet size, and ω being *cwnd* in packets.

Then, the question is how to measure and compute the average packet size. In order to compute the average packet size, we have used the EWMA averaging method. As known, the choice of the EWMA coefficient can drastically affect the averaged values. To this end, we have conducted a simple heuristic measurement study which worked well, and come to a conclusion for the use of two separate coefficients: use larger coefficient for the large frame, and small coefficient for the small frames.

We have observed that the packet sizes are often small and so as the frame size. Thus, we applied the smaller coefficient when frames contained two or less packets, whereas the bigger coefficient when frames contained more than two packets, which is shown in Equation 5.2.

$$\begin{aligned} \mathcal{L}: \quad & s_1(n) = \lambda_1 * \bar{a}(n) + (1 - \lambda_1) * s_1(n-1) \\ \mathcal{S}: \quad & s_2(n) = \lambda_2 * \bar{a}(n) + (1 - \lambda_2) * s_2(n-1) \end{aligned} \tag{5.2}$$

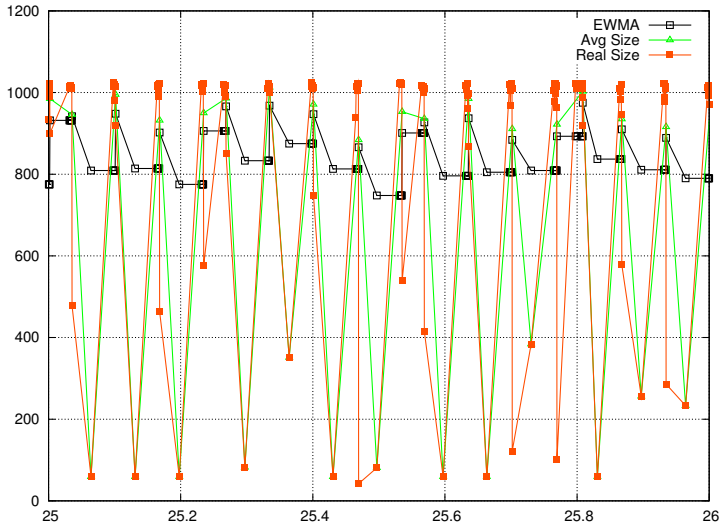
where:

- $\mathcal{L}$ and $\mathcal{S}$: long and short frames, respectively
- $s(n)$: EWMA averaged packet size for the n^{th} frame
- $\bar{a}(n)$: arithmetic mean packet size for the n^{th} frame
- λ_1: 0.75
- λ_2: 0.15

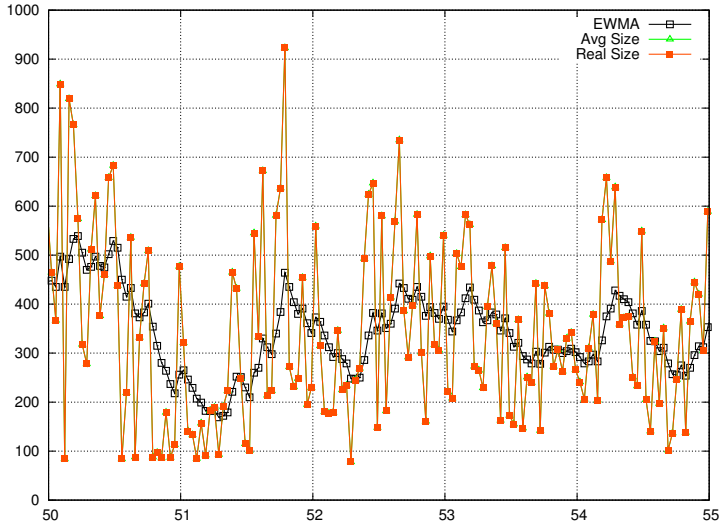
The estimated packet size dynamics are shown in Figure 5.5, where x-axix shows the run time in seconds, and y-axis shows the packet size in bytes. Figure 5.5(a) uses the Macbook built-in iSight® USB webcam that has high motion complexity. The actual packet sizes are nearly 1000 bytes, where the last packet on every frame appears to be a lot smaller. Figure 5.5(b) uses the same USB webcam which shows a fairly low to medium motion complexity. For the high motion complexity, it has generated more than 2 packets per frame, hence, applied the $\mathcal{L}$ equation, whereas the low to medium motion complexity cases have used the $\mathcal{S}$ equation to estimate the average packet size. With this packet size estimation result, we calculate the *cwnd* size in bytes using Equation (5.1).

5.4.5 Sender-based TFRC

For the fair comparison between TFWC and TFRC, we have converted TFRC to a sender-based mode, where the data sender computes the loss event rate. As we already have the underlying transmission system to carry data and control packets, we use exactly the same platform when converting TFRC to a sender-based mode. For example, we carry on using AckVec mechanism that TFWC has used when



(a) Macbook built-in iSight USB webcam with high motion complexity



(b) Macbook built-in iSight USB webcam with low-medium motion complexity

Figure 5.5: EWMA-Averaged Packet Size

reporting the reception of data packet. Then the rest of the implementation becomes relatively straight forward apart from calculating the send rate: in this case, *inter-packet interval*. To calculate the inter-packet interval, we translate the calculated sending rate (from the TFRC algorithm) to those intervals as follows:

$$\gamma = \frac{s}{\chi} \tag{5.3}$$

where γ is the inter-packet interval in seconds, s the EWMA-averaged packet size in bytes, and χ being the computed sending rate by TFRC. Then the Vic transmits the packets by spreading them out in the computed interval by invoking `timeout(`γ`)` method in those intervals, whereby the `timeout(`γ`)` method calls the send routine in the Vic after γ seconds and checks whether it allows the next packet transmission. One thing that should be considered in this operation is that we need to calibrate γ appropriately when the OS scheduler does not select the Vic process timely due to its interrupt timer granularity.

For example, the Vic's send routine is scheduled at time $t > \gamma$, where t is a positive constant[5]. Then, the next `timeout()` should be called using a new interval (γ_{new}) followed by an immediate packet transmission, where $\gamma_{new} = \gamma - (t - \gamma) = 2\,\gamma - t$. In order to compensate this timing error, we therefore introduce a calibration formula as follows.

Let:

- t be the time that OS scheduled the send process, and γ the inter-packet interval.
- $\gamma < t < k\,\gamma\,, \quad$ where $\; k = \left\lfloor \frac{t}{\gamma} \right\rfloor$
  (k is a maximum integer value that satisfies the inequality)

Then, the calibrated inter-packet interval (γ_{new}) is:

$$\gamma_{new} = (1 + k)\,\gamma - t \tag{5.4}$$

where the Vic immediately sends k packets before it sets a new timer using γ_{new}. However, when the calculated γ is much smaller than the host OS's clock granularity, this calibration would not work effectively. In an extreme case where RTT is extremely short with high data rates, it would result in a very bulky packet transmission: one of the limitations being a rate-based controller.

5.5 Summary

In this chapter, we have given the overview of the Vic architecture and its underlying transmission system. We have also described how we have integrated two congestion control mechanisms into the Vic tool. To implement `Ack` mechanisms, we have used the RTCP XR packet format (*Block Type 1*) to carry AckVec and AoA information between a sender and receiver. When packet re-ordering occurs, we investigate if the disordered packet initiated a new loss event. If it did, we then revert the ALI and the timestamp for the last lost packet into the previous state. For TFRC, the accurate RTT measurement is critical as it

[5] Assume $(\gamma < t < 2\,\gamma)$ for this particular example, and $(\gamma \gtrsim C)$, where C is a host OS's clock granularity.

is directly used in the equation. Therefore, we have dealt with the small but influential arbitrary delay using two socket options: `SO_SNDBUF` and `SO_TIMESTAMP`. We then introduced the *cwnd* computation in bytes instead of packets in order for the congestion control to actually administrate the sending rate efficiently. For example, suppose there are 20 packets (small-sized packets) in the send queue awaiting for the transmission with a small *cwnd*, let say 10. It could build a persistent queue of 10 packets bounded by an `Ack`-clock, whereas if we maintain the *cwnd* size in *bytes*, then we could send as many number of bytes as `Ack` acknowledges them. Finally, we turned TFRC into a sender-based mode partly for a fair comparison, and for the easiness of implementation (re-used the underlying transmission system). In the following chapter, we will evaluate the performance of TFWC and TFRC using the developed Vic system.

Chapter 6

Experiments and Evaluations

There have been dozens of algorithms and techniques to propose end-to-end congestion control for real-time media streaming applications, but none of them have been widely used to date in real-world applications. Instead, such applications simply use TCP directly for pre-recorded, stored media, or UDP when there is a strict timing requirement. This leads to the question why not apply the proposed control mechanisms in the application? One of the reasons could be that the congestion control techniques are only proven to work in simulations, or the benefit is not substantial enough to convince the application writers to use them when codecs are so powerful. This chapter answers this question using results from real-world experiments and analysis using the Vic video conferencing system [12].

6.1 Introduction

We have conducted simulation studies in Chapter 4 that TFWC can generate fair and smooth throughput while being responsive to changes in the bandwidth. In this chapter, we evaluate TFWC (our proposed mechanism) and TFRC (the proposed standard mechanism in the IETF) using the Vic application. The objective of these experiments is to study and discuss the two protocols' behavior in interactive video conferencing system (e.g., Vic) and verify the applicability of these congestion control mechanisms. To the best of our knowledge, this is the first research work at present to apply the two types of congestion control mechanisms (i.e., rate-based vs. window-based) in a real-world application and examine the overall systems performance.

6.1.1 Methodology and Environments

The methodologies used for the real-world experiments are similar to those used for simulation studies, which are explained in Section 4.1. We start by validating the functional blocks of the TFWC within the Vic tool. Similar to the simulation studies, the validation includes the AckVec mechanism, the ALI computation, the *cwnd* computation, and the timers. Some detailed validation results are presented in Appendix A.3.

For the test environments, we run the Vic tool over HEN testbed [5]. The HEN testbed provides flexible and reconfigurable experimental network topologies that can be easily controllable to meet our objectives. The testbed network topology we use is a simple dumbbell structure shown in Figure 6.1. When constructing the test environments, we create TCP cross traffic sources using the *iperf* tool [7],

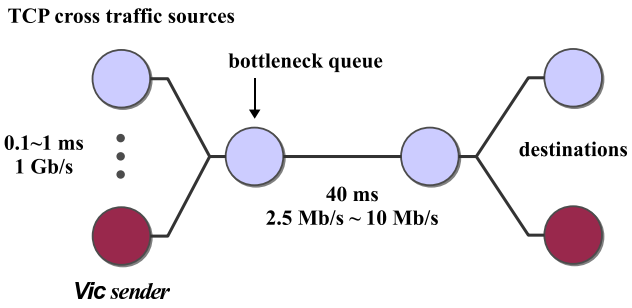


Figure 6.1: HEN Testbed Topology

and configure bandwidth, delay, and the bottleneck queue size using *Dummynet* package [4]. Each node is equipped with an AMD Opteron™ 248 CPU (2.2Ghz) with a 2GB of memory running Linux (*kernel 2.6.23 patch level 12*). We have also used an Apple Macbook computer equipped with a built-in iSight® USB webcam. Table 6.1 shows the specifications of machines that were used in our experiments.

With this environment, we evaluate the metrics introduced in Section 4.1. However, considering the different nature of real-world video streaming with the sources used in the simulations, we will re-interpret them and introduce new performance metrics in this chapter. The performance metrics can be classified into two categories: namely, *network oriented* and *end-user oriented*. While the four performance metrics used in Chapter 4 fall into the network oriented measures, we have yet to discuss suitable measures that can directly reflect the end user's impression. For example, in order to measure the received video quality it is hard to conclude which one of the four metrics represents the perceived quality directly. Thus, we introduce PSNR and MOS when evaluating the TFWC and TFRC protocols in the video conferencing system to measure the end user's perception. In the simulations, we did not need to use these user-oriented metrics as we used sources that possess a FTP-like characteristic – it did not necessarily emulate a multimedia video characteristic (e.g., in the simulations the data packets were always available at the sender – see Section 4.2.1). There is a relevant work that enables transmitting pre-recorded image sequences over *ns-2* simulator [42], but it is unclear as to how well the systems can best reflect the dynamics of real-world applications. In summary, we need to re-define the metrics that were used in the simulations, and also introduce a new set of measures for the performance study in this chapter. In the following sections, we discuss further the two categories of the performance metrics used with the Vic application.

6.1.2 Performance Metrics: *Network oriented*

We have introduced the four metrics in Section 4.1.2: fairness, smoothness, stability, and responsiveness. All of these are, to some extent, network-friendly metrics that did not consider the characteristics of the

Table 6.1: Inventory of the test machines used for experiments.

Model	CPU	No. of CPUs	Memory (GB)	Operating System
Sun X4100	AMD Opteron® Proessor 248 2.2Ghz	1	2	Linux 2.6.23 patch level 12
Apple Macbook	Intel Core 2 Duo® 2.26 Ghz	2	2	Mac OSX 10.6.4

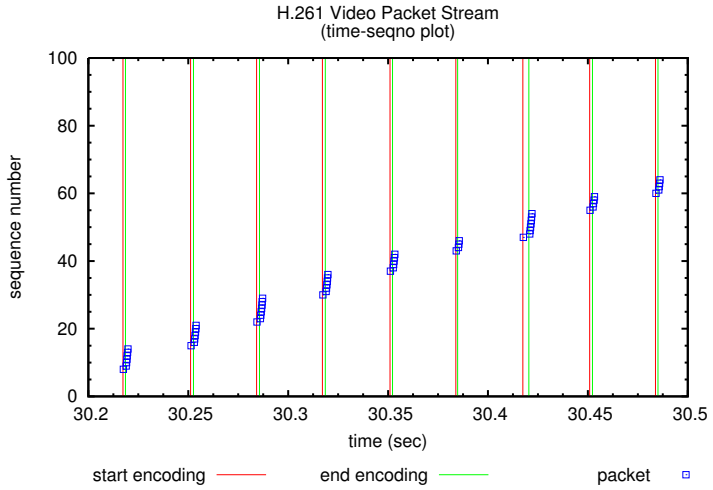


Figure 6.2: An Example of H.261 Video Packet Streams without Congestion Control

media content and its underlying transmission system which also affects the user's final impression. These metrics are defined under the two assumptions, which are:

- data availability – application data is always available.
- packet size – packet size is fixed.

The above assumptions need to be questioned when it comes to the real-world. For example, the protocol's smoothness may not be ideally achieved in the real-world applications as in the simulations, because the application data may not be always available to be sent even though congestion control modules have established such a smooth rate (i.e., the data would have been sent if it were available). It is partly due to the fact that the media encoder normally encodes frames when the media contents become available, resulting in the output bit rate being highly bursty. Therefore, although congestion control mechanisms can provide a very smooth transmission rate, the actual transmission rate may not be as smooth as one would have hoped, because it can be limited by application data. Another difference that we consider in this chapter is the packet size. Unlike the simulations, the packet size (and frame length) can be highly dynamic depending upon media content and encoding bit rate. Normally speaking, the bigger the frames the longer the packet sizes. In the next subsections, we describe the practical cases of the above two assumptions in terms of validating of how well or ill-fitted they are in real cases.

6.1.2.1 Data Availability

Let's take a look at a practical example for the first assumption. Suppose there is a video streaming application using the H.261 codec in an environment where the receiver is behind a typical DSL-like Internet link and the server is located in a university campus network. There is no congestion control mechanism applied in this example. Figure 6.2 shows the time vs. sequence number plot (so called, time-seqno plot) at the sender. It shows that the application transmits the packets immediately as the frames are encoded into the packets (the two adjacent vertical lines stand for the start and end of encoding, respectively), resulting in a bursty send rate overall. Thus, a rate-based congestion control algorithm would space out the packets to produce a smoother transmission rate, and a window-based congestion control would

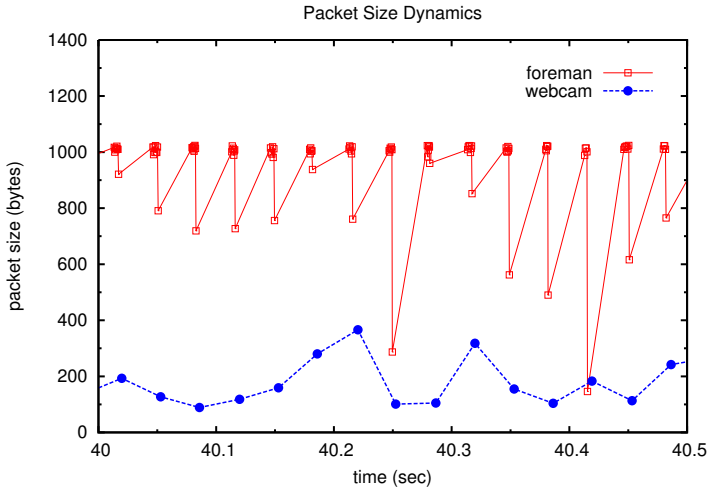


Figure 6.3: Packet Size Dynamics

adjust the window size to have similar effects in the send rate. Nevertheless, the application's send rate might not always be as smooth as the calculated rate by a congestion control mechanism. Thus, it is less meaningful to evaluate the smoothness of the *application send rates*, because these rates are often limited by data availability.

6.1.2.2 Packet Size Variability

Let's consider another example to show how the packet size can vary in practice. As briefly mentioned, the packet size of multimedia applications can be remarkably variable, especially when the source material is dynamic. Before mentioning the packet size variability, we start by revisiting the multimedia transmission scheme briefly. A typical encoding and transmission procedure can be explained with the following steps: compression, packetization, and transmission. So to speak, the video device grabs raw images from the source and feeds them to a codec: for example, the H.261 codec in the Vic application. Then, the encoder performs per frame image compression and packetization, where each frame is partitioned to small blocks and only the changed blocks are transmitted. In this scheme, when the source is highly dynamic, the frame size[1] would become larger (because almost every partitioned block has to be transmitted), generating larger packets.

Now, consider two video image sources that have high and low motion complexity, respectively. We have chosen one of the well-known CIF image sequences, `foreman.yuv` (352×288 pixels)[2], to demonstrate the high motion case, and the Apple Macbook built-in iSight™ USB webcam for the low motion case. Both video sequences are coded at 30 fps and transmitted using the H.261 codec in the Vic application. The video sources are run in a local LAN environment separately, and we plot them together for a half a second period. When streaming video using the built-in iSight™ USB webcam, the user did not move much in front of the camera apart from natural facial movements to emulate low motion complexity. Figure 6.3 shows the packet size dynamics for the two different video sources. As you can see in this graph, the `foreman` source has generated packets of 1000 bytes with approximately

[1] In this thesis, we mean the *frame size* as the total number of encoded bits for that frame.
[2] A brief summary of `foreman.yuv` can be found in Table B.1. Also, the sample video sources can be downloaded from [13].

Table 6.2: Mean Opinion Score [15]

MOS	Quality	Impairment
5	Excellent	Imperceptible
4	Good	Perceptible, but not annoying
3	Fair	Slightly annoying
2	Poor	Annoying
1	Bad	Very annoying

10 packets per frame[3], whereas with the USB camera the source has generated only one packet per frame and the packet size is extremely small (200 bytes roughly) compared with the `foreman` source. Although it did not appear in Figure 6.3, both the packet size and frame length did increase notably when a user moved significantly around in front of the camera to create high motion complexity.

As a result, the two practical examples shown in Section 6.1.2.1 and Section 6.1.2.2 showed that our earlier assumptions are no longer valid, and this must be taken into consideration in the real-world experiments.

6.1.3 Performance Metrics: *End-user oriented*

So far, we have investigated the performance metrics from a network-oriented view, mainly when using *ns-2* simulator. The four metrics that we repetitively speak of (fairness, smoothness, stability, and responsiveness) are related to attributes of a protocol's behavior over various network environment without directly considering users' satisfaction. Of course, when the networking research community initially developed these metrics it is widely considered that they are closely related to users' contentment to a certain degree. The development of TFRC and its variants is due to this consensus: for instance, users prefer a smoothness in the send rate to abrupt changes. However, the smoothness alone is insufficient to quantify users' satisfaction. To address the quality of multimedia video streaming services from a user's point of view, we introduce the two commonly used measures in this section, which are:

- Peak Signal-to-Noise Ratio (PSNR)
- Mean Opinion Score (MOS)

Broadly speaking, there are two categories in the quality measures, namely subjective quality and objective quality measures. The subjective quality metrics capture how the video is perceived by users and designates users opinion on a particular video sequence. However, subjective video measures are extremely expensive: highly time consuming (to prepare and run), with a high level of human resources and special equipment also being needed. Such subjective measures are described in ANSI [2], ITU [17, 18], and MPEG [38]. The main idea of measuring subjective video quality is calculating MOS values of the transmitted video sequences at the receiver side. The MOS tests for voice was specified quite some time ago in [15] (shown in Table 6.2).

As the subjective measures are quite expensive and complex, objective metrics have been developed to emulate the quality impression of the human visual system. In [65], the authors had extensive discussion on various objective metrics and their performance compared to subjective test cases. Never-

[3]The tailing packet size was always smaller than the packets in the middle of the encoding loop.

Table 6.3: An Example of PSNR to MOS mapping

PSNR [db]	MOS
> 37	5 (Excellent)
31 - 37	4 (Good)
25 - 31	3 (Fair)
20 - 25	2 (Poor)
< 20	1 (Bad)

theless, the Mean Square Error (MSE) and PSNR are the most popular difference metrics in networked video transmission systems: MSE measures image difference and PSNR measures image fidelity (i.e., how closely an image resembles a reference image). The popularity of these metrics is rooted in the fact minimizing MSE is equivalent to least-squares optimization (e.g., minimize the sum of squared residuals[4]), and their calculations are very easy and fast. The PSNR compares the maximum possible signal energy to the noise energy, which has shown to result in a higher correlation with the subjective quality perception. Then, the PSNR of two image sequence, I and $\widetilde{I}$, can be computed by:

$$\mathsf{PSNR}(I, \widetilde{I}) = 10 \log_{10} \frac{m^2}{\mathsf{MSE}(I, \widetilde{I})}, \tag{6.1}$$

where

$$\mathsf{MSE}(I, \widetilde{I}) = \frac{1}{XY} \sum_{x} \sum_{y} [I(x, y) - \widetilde{I}(x, y)]^2 \tag{6.2}$$

for pictures of size $X * Y$ pixels, and m is the maximum value that a pixel can take (e.g., 255 for 8-bit images).

However, the PSNR is a controversial metric; some studies have shown that PSNR to poorly correlate with subjective quality [35, 64]. Recently, therefore, PSNR has been at the center of a constant debate as to whether it is completely irrelevant for evaluating the quality of digitally compressed video or the performance of a video codec. In spite of the ongoing debate and discussion, it is still widely used when evaluating codec performance (e.g. as a measure of gain in quality for a specified target encoding bit rate), video codec optimization or as a comparison method between different video codecs (e.g., [10]). Thus, we can say PSNR is arguably considered to be a reference benchmark for developing perceptual video quality metrics. The validity of PSNR is particularly agreed when used carefully; it is only conclusively valid when it is used to compare results from the same codec (or codec type) and content. Given the fact that the PSNR measure is a reasonable measure to evaluate the end-user's impression with the same codec and content, we calculate the MOS values to map the PSNR percentages of the received frames as shown in Table 6.3.

6.1.4 Video Codec

The choice of codecs plays an important role in the overall performance of multimedia transmission systems. To select a suitable video codec for our experiments, we have considered the following criteria:

[4]The residual is a difference between an observed value and its reference.

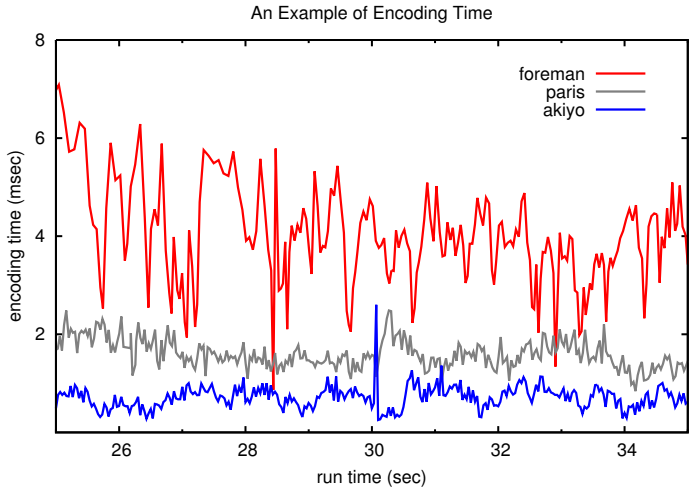


Figure 6.4: An example of elapsed time for the H.261 codec with sources of `foreman` (high motion complexity), `paris` (medium motion complexity), and `akiyo` (low motion complexity), respectively. Each video source runs independently at an LAN like environment and plot them all together. They are coded at 30 fps with a bit rate of 3 Mb/s roughly.

- The codec must be as simple (and as fast) as possible to minimize complexities for the overall systems performance analysis
- The video compression principles must be similar to a modern codec (e.g., H.264 [16, 19]) – block-oriented motion-compensation coding scheme[5]

While considering the criteria, we have not counted on the coding efficiency as it is simply not our goals to study issues around the codec's performance. To meet our requirements as closely as possible, we have chosen to use the H.261 [14] as a reference video codec in this thesis. the H.261 codec is fairly simple and, therefore, a choice quite suitable to serve our purpose for studies on the interactions between codecs and congestion control mechanisms. The Vic tool also implements a derivative of the H.261 codec. As explained in [49], the H.261 codec in Vic performs *intra-mode* only using the so-called *conditional replenishment* procedure, mainly to overcome severe decoding impairments and to enhance compression efficiency especially when packet loss occurred. Figure 6.4 illustrates a typical encoding time using the H.261 codec using an Apple Macbook fitted with an Intel Core 2 Duo™ 2.26 Ghz CPU and 2 GB of memory. Of course, the encoding time will largely depend on the overall system load and media contents, it seems reasonable for our experiments.

6.1.5 Section Summary

We have described our test environments and performance metrics in this section. To summarize, we will construct various network environments over the HEN testbed at UCL Computer Science using *iperf* (to create cross TCP traffic sources) and *Dummynet* (to regulate bandwidth, link loss rate, and bottleneck queue size). We will evaluate two performance categories, network oriented metrics and end-user oriented metrics, in the rest of this chapter, using the H.261 codec implemented in the Vic tool.

[5] Although the H.261 on the Vic only runs intra-mode only, it applies a special algorithm, called the *conditional replenishment* procedure, which is quite similar to the block-based motion-compensation encoding scheme.

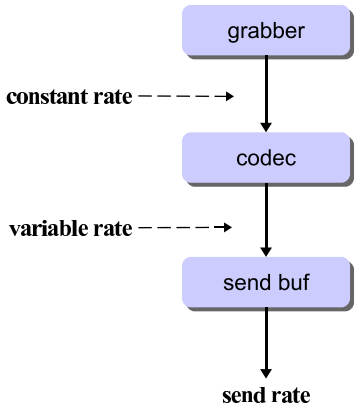


Figure 6.5: Vic Data Streams

6.2 Send Buffer Control

In this section, we examine how the send buffer affects the overall dynamics in the Vic system. As briefly explained in Chapter 5, the packet streaming output rates are determined mainly by relations between two components in the system: grabbing rate by video device, and codec bit rate (which then becomes an input rate to the send buffer), as illustrated in Figure 6.5. As the video device grabs image frames (at a constant rate, namely *frame rate*), the grabbed images are fed to the codec which transforms them into a series of packets (namely *frame size*), and then inputs them to the send buffer (this rate is variable). Once the packets are stored in the send buffer, the system transmits them with a rate established by congestion control module. So, the system dynamics can be determined by the interactions between these components. Our aim, then, is to control the send buffer length (to minimize end-to-end packet latency and to ensure there are always available data to transmit) while generating the best image quality as much as possible (by increasing codec bit rate and/or grabbing rate) with the aid of congestion control mechanisms. The role of congestion control mechanisms is to give useful information to the application (specifically to codec and grabber) so they can alter the target bit rate or frame grabbing rate on the fly. Consequently, the application adjusts the *frame rate* and *frame size*[6] to control the send buffer for the requirements of the real-time *interactive* video conferencing systems. Therefore, instead of using a fixed frame rate and size a priori throughout the entire life time of the connection, Vic varies the frame rate and size on the fly to control the send buffer length and to achieve the best possible image quality while maintaining the application requirements. There are two parameters that can control the frame rate and size: fps in the video device and the *quantizer* in the codec (so-called, q factor in Vic). We can achieve our best frame rate by automatically adjusting fps on the fly and also frame size by manipulating the q factor within the encoding loop. In summary, the send buffer can be controlled by:

- frame rate control – adjusting fps
- frame size control – adjusting q factor

[6] As mentioned, the *frame size* means the total number of encoded bits for that frame. Thus, the frame size control in this thesis means controlling the output encoding bit rates – i.e., it does not mean image sizes in $X * Y$ pixels.

In the next two subsections, we detail how we can control the send buffer with the support from congestion control mechanisms.

6.2.1 Frame Rate Control

Let's consider an example of the dynamics of the send buffer size when no congestion control techniques are applied. Figure 6.6 shows a typical send buffer dynamics without congestion control. The flow runs approximately 1.5 Mb/s with a frame rate of 30 fps. Also the codec's q factor is set to 10 for the entire life time of the flow. Each frame size is 5$\sim$7 packets roughly, and all packets associated with a frame have transmitted just after finishing an encoding loop (similar to Figure 6.2), which results in keeping the send buffer length between 0 and the number of packets for that frame (the right hand side of y-axis represents the send buffer length in packets). It may look good at first glimpse because Vic maintained the send buffer length always smaller than a frame size, which drives the end-to-end latency as low as possible. However, it may not fully utilize the available network resources, because the packets would have been transmitted if they were in the send buffer. Decreasing q factor simply cannot be a solution because it may lead to building up a persistently increasing queue in the send buffer. Therefore, we require some sort of control mechanism.

Now consider we apply TFWC in a similar environment in the Vic system. With this network environment, the approximated *cwnd* size is going to be quite large (using Equation (4.5) *cwnd* in this example is roughly 150 packets, assuming the bottleneck queue is 50 packets – i.e., the default *Dummynet* queue size). Then, the dynamics would be similar to Figure 6.6. However, *cwnd* will decrease as packet loss occurs. For example, if *cwnd* is 3$\sim$4 packets, the send buffer will grow forever (average number of packets per frame was 5$\sim$7 packets), hence the need to control the frame grabbing rate at some point. In fact, we actually skip grabbing frames when send buffer builds up[7].

In this network environment, our target is to keep the send buffer length to less than approximately 5 frames[8], the maximum number of frames which satisfies the timing requirement (e.g., 200 ms) for

```
1 On Grabbing Image Sequence
2   /* check send buffer length and decide suspend grabbing */
3   sbuf = get_sbuf_len();
4   if sbuf > 5 frames then
5     suspend_grabbing();
6 end
```

Algorithm 4: Skip frame grabber for the send buffer control

[7] We have illustrated the detailed mechanisms how to measure the send buffer in Appendix C.

[8] The total elapsed time for the i^{th} frame (t_i) transmission can be computed roughly as:

$$t_i = \frac{n_i\, s}{l} \quad \text{(sec)}$$

where:

- n_i: the total number of packets for the i^{th} frame
- s: packet size in bits
- l: link speed in bits/sec

Then, it takes $(7 \text{ packets} * 1000 \text{ bytes} * 8)/1.5$ Mbps $\cong 0.037$ (sec) to fully transmit a frame. Therefore, in order to meet the timing requirement for the interactive video applications (which is 200 ms), the buffer can hold up to 5 frames. Note this reference

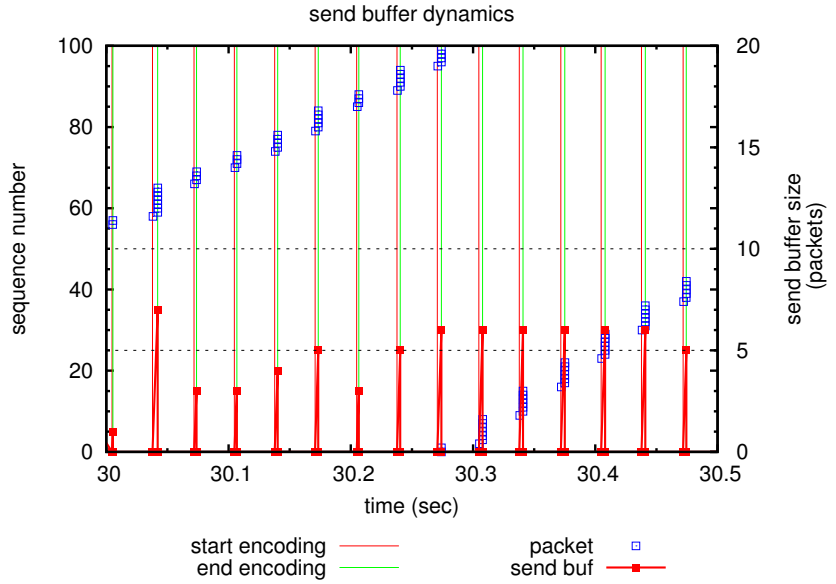


Figure 6.6: Dynamics of Vic send buffer length. We have used foreman.yuv over HEN testbed. The bottleneck bandwidth is configured to 10 Mb/s with a delay of 40 ms using *Dummynet* - a fairly similar network environment between a DSL to an ISP network.

real-time and interactive video sessions, as shown in Algorithm 4.

To demonstrate how we control the send buffer enabling TFWC this time, we introduce a small loss rate to push down *cwnd* value: note, without a loss, *cwnd* will grow only if the TCP window validation [37] is not used. Figure 6.7 uses a dumbbell topology just like the one used for Figure 6.6 over HEN testbed: 10Mb/s of bandwidth with 40 ms link delay using foreman source. We have introduced $1\sim2\%$ of random packet loss in this experiment. We plot the send buffer dynamics during a half a second period. As the send buffer builds up it skips grabbing frames (shown by the dashed line in the graph) when the send buffer exceeds its length containing 5 or more frames, so the Vic system drains the send queue upon the next `Ack`-clocks: packets are transmitted in `Ack`-clocking basis. Here the average *cwnd* size is approximately 10 packets. Roughly speaking, *cwnd* is inversely proportional to the square root of the loss event rate. The first term of Equation (3.2) is dominant at low loss rates whereas the second term will be dominant in a high loss rate regime. Hence, Equation (3.4) can be further simplified:

$$\acute{w} \leftarrow \frac{c}{\sqrt{p}}, \tag{6.3}$$

where $\acute{w}$ is an approximated *cwnd*, c is a constant, and p is a loss event rate. Then, $\acute{w} \cong 1/\sqrt{0.01} = 10$.

As a result, the video device suspended grabbing image sequences as explained, resulting in the frame rate of 20 fps. However, adjusting frame rate alone is rather a crude control (i.e., when it skips a frame, it ends up missing whole packets associated with the frame). In the next subsection, we investigate a finer control technique to manage the send buffer.

is based on a worst case scenario where *cwnd* becomes a very small in a DSL-like environment. Therefore, this reference buffer size could be considered as a soft lower bound.

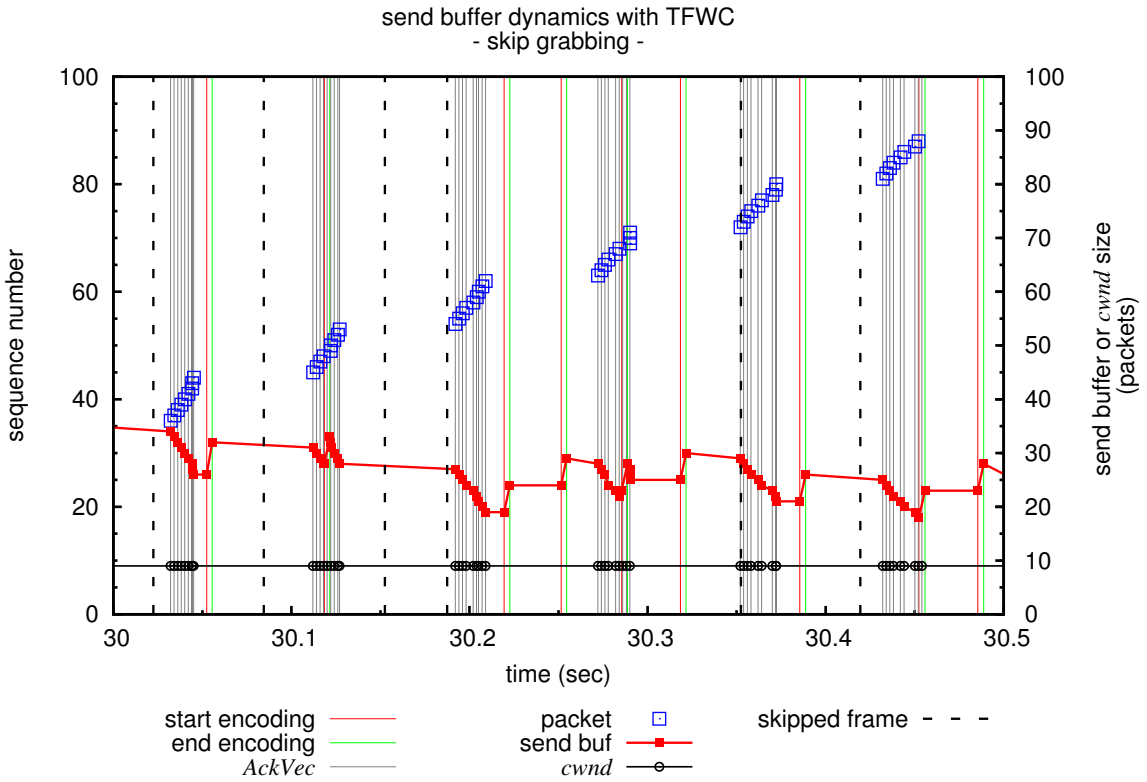


Figure 6.7: Dynamics of Vic send buffer length with TFWC congestion control mechanism. The bottleneck is configured for 10 Mb/s with a delay of 40 ms using *Dummynet*. In this case, we introduced a loss rate of 1∼2%, resulting in *cwnd* size roughly 10 packets.

6.2.2 Frame Size Control

As illustrated in Figure 6.5, the send buffer management can be accomplished in two separate ways: control grabbing rate and codec's output rate. In the previous subsection, we have explained the grabbing rate control mechanism when the buffer significantly builds up. Whilst this method can manage the send buffer length effectively to some degree, it has its limitations. First, when video device skips grabbing a frame, it misses all packets associated with the image sequence, in which some have actually been packetized and transmitted in time successfully – possibility of abrupt image quality degradation. Thus, when the send buffer exceeds unacceptable range (e.g., a upper bound), it would be better if we:

- have a finer granularity control scheme instead of frame-by-frame, when necessary.
- reduce the codec's output bit rate, leaving the input frame rate unchanged.

Second, the frame grabbing rate cannot go faster than a maximum rate that a codec can accommodate, so when the system runs at the maximum frame rate, the only way to increase the bit rate is to dynamically change the q factor – a possible limitation on maximum achievable image quality, if not changing *quantizer* when possible. In this case, the codec would have been able to generate a higher bit rate by decreasing the q factor. Figure 6.8 suggests that the bit rate curves move from one to another as the source's motion complexity varies for different q factor values.

Similar to other video conferencing systems, the Vic tool does not have a direct mechanism to control the codec's output bit rate on the fly – i.e., once the q factor is preset at the beginning of transmission, all images use this fixed q value throughput the entire life time of the connection. So when there is still network bandwidth available while the system operates at the maximum input frame rate (with the prefixed q factor), there are no possible ways to increase the image quality. For example, when the system is already at the maximum input frame rate, the fixed q value would produce a CBR-like output bit rate, assuming the image contents do not vary significantly. What is suitable is to increase both the input frame rate and the bit rate as much as we can whenever possible if there are still available network resources. However, due to the continually changing complexity of pictures in a real video sequence, it becomes less obvious as to which value of the q factor to select. If we set and fix the q factor for an "easy" part of the sequence having slow motion and uniform areas, then the bit rate will go up dramatically when it

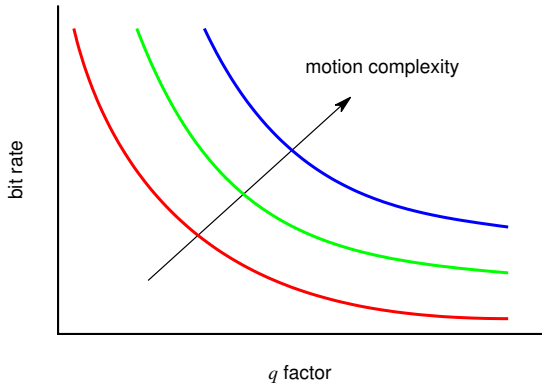


Figure 6.8: Bit rate dynamics as source complexity changes with different *quantizer* values used.

reaches the "hard" (i.e., more complex) parts of the images, resulting in a high send buffer length. There are some modern codecs (e.g. H.264) that control the quantizer dynamically (although the standard does not include dynamic rate control features), but they solely implement a closed-loop rate control algorithm mainly based on the complexity of uncompressed input images, which does not count on network information at all.

What we propose here is combining the congestion control mechanisms with the Vic transmission system (e.g., grabber, codec, etc.) to regulate the application send rate whilst the grabber and codec can alter their output rates (fps and bit rate) dynamically. What we aim to achieve is to minimize the send buffer length and maximize the image quality, in order for Vic to transmit best quality images at all times. To this end, we apply a simple but quite effective conditional function at the encoder as shown in Algorithm 5. This simple routine essentially increases and decreases by observing the send buffer on start of every frame encoding instance. While the send rate is determined by congestion control mechanisms, it monitors the send buffer to see if it can adjust image quality. Figure 6.9 shows a typical example of usefulness of this condition. We have used the Apple Macbook built-in iSight™ USB webcam for the both tests in a local LAN environment (i.e., very short RTT with 10 Mb/s bandwidth). In this environment, the packet loss rate is almost equal to zero, which exhibits *cwnd* large enough to send all packets as Vic packetizes frames. Figure 6.9(a) uses a fixed q factor whereas Figure 6.9(b) uses a dynamic q factor adjustment. As shown in the both figures, the send buffer is always empty (because *cwnd* always allowed sending them), so with dynamic q factor adjustment enabled, it has increased the codec's output bit rate, generating more packets per frame.

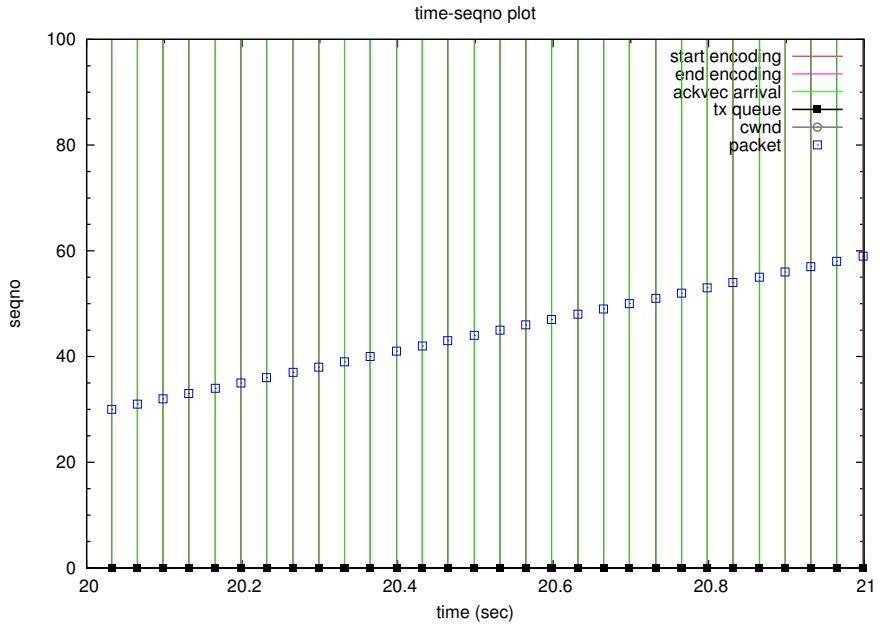
```
1  numAvg(average number of packets in a frame)
2  sbuf(send buffer length in packets)
3  On starting every frame encoding instance
4      if sbuf ≤ numAvg then
5          quantizer− −;
6      else if (sbuf > numAvg ) && (sbuf ≤ 3.5 * numAvg ) then
7          quantizer ++;
8      else
9          quantizer += 2;
10 end
```

Algorithm 5: The *quantizer* adjustment depending on the send buffer length

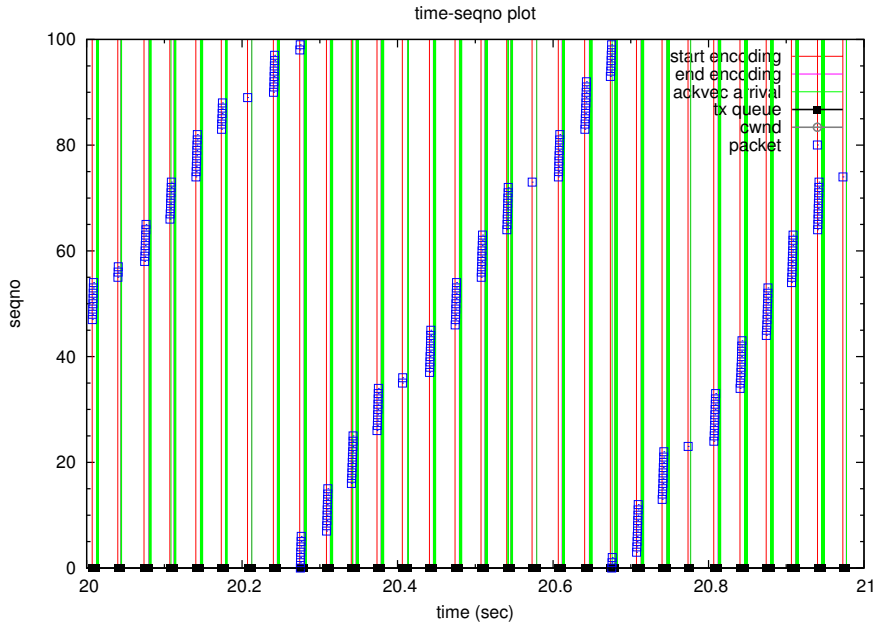
In this subsection, we have described and demonstrated how we can dynamically adjust image quality by manipulating an important codec parameter (e.g., *quantizer*) whilst the overall send rate is still controlled by congestion control mechanisms.

6.2.3 Section Summary

We have investigated how the send buffer affects the overall transmission behavior in the system. We skip grabbing images when the send buffer significantly builds up whereas for the finer control we dynamically adjust the q factor on every encoding loop. In the next sections, we evaluate the performance metrics defined in Section 6.1.



(a) with fixed q factor



(b) with dynamic q factor adjustment

Figure 6.9: Dynamic q factor adjustment.

6.3 Throughput Dynamics: *Ideal situation*

Before we go into the detailed performance analysis, we present how the Vic throughput behaves when TFWC or TFRC is applied in an ideal situation. We have placed a single TFWC or TFRC flow over the test topology as depicted in Figure 6.1. In this test, we have used the bottleneck bandwidth of 10 Mb/s with an end-to-end delay of 80 ms (again, these are typical network parameters that normal users might experience using DSL-like services). We then run each test individually and plot them together for a 30 second period as shown in Figure 6.10. The throughput was measured at the bottleneck node with 100 ms time interval when averaging them.

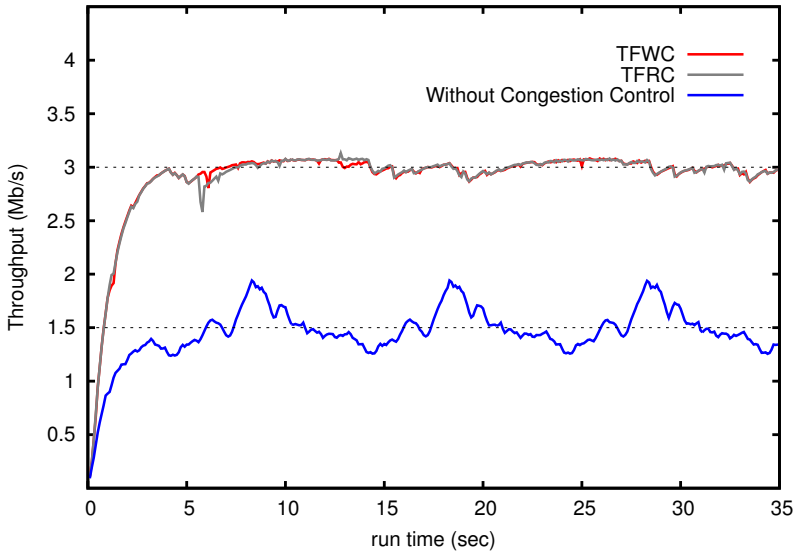


Figure 6.10: Vic throughput dynamics with the foreman image sequences.

Both TFWC and TFRC had almost identical slow start behaviors. When no congestion control is applied, the q factor is the Vic default value, 10, and the frame grabbing rate is fixed at 30 fps, resulting in 1.5 Mb/s throughput on average. On the other hand, with TFWC and TFRC, the throughput reaches 3 Mb/s on average for both cases, and they are much smoother than the one with no congestion control. Note, with TFWC and TFRC, the q factor and the frame grabbing rate have been dynamically adjusted. There is a little drop in the throughput just after the slow-start phase with enabled congestion control mechanisms, especially with TFRC, but these are due to the OS having not scheduled the Vic properly somehow, causing to stop the sending of packets all of a sudden for a few millisecond or so (i.e., they are *not* an artifact from the congestion control mechanisms.). In summary, these are the maximum throughput that the Vic can achieve, so we consider them as a reference behavior in an ideal network status (e.g., sufficient bandwidth with no losses).

Beginning by defining the scope of our experiments, we evaluate the performance of congestion control mechanisms (TFRC and TFWC) over various network environments in the next sections.

Table 6.4: HEN testbed parameters used for the experiments.

Bandwidth	RTT	imposed link loss rate	queue	queue size
5 Mb/s	80 ms	0%∼1%	DropTail	50
2.5 Mb/s	80 ms	0%∼1%	DropTail	50

6.4 Scope of the Experiments

It is obvious that the real-world experiment studies would bring more tangible ideas on using and applying the proposed congestion control mechanisms in practice. However, those experiments necessitate a well defined scope so that the results can be aptly used for the meaningful analysis. Similar rules apply to the simulations as well. As mentioned at the beginning of Chapter 4, the scope of the simulations covered rather a wide range of network parameters. The principal reason for having checked across the wide range of parameters was to explore our design space in an attempt to fathom the protocol's functionalities by understanding its limitations and possibilities, and thus obligated that way. So, one of our goals for the performance analysis with the simulator was to bring out any issues related to its functions over various network settings and verify possibilities for the final usage in real-world applications, which we concluded its options to be quite useful for the real applications.

On the other hand, in this chapter, we will primarily focus on the interested regimes where TFWC and TFRC both have revealed limitations in the simulations, especially when there is low bandwidth and relatively low statistical multiplexing (See Figure 4.3 and Figure 4.4). These environments are particularly coherent to those with the most end-users would experience daily: a more or less DSL-like environment, if we say. Therefore, we rather confine the network parameters in the real-world experiments, addressing each protocol's performance in those network environments. Table 6.4 shows the chosen network parameters used in our experiments for the rest of this chapter.

6.5 Network-oriented Performance Evaluation

This section presents the performance results of TFWC and TFRC with the Vic video conferencing system in terms of fairness and responsiveness (the network-friendly metrics).

6.5.1 Fairness

We investigate the TFRC and TFWC protocol fairness when they compete with the standard TCP flows. As we mentioned, our interested network environments include low bandwidth (i.e., less than 5 Mb/s) and low level of statical multiplexing. To this end, we use a simple dumbbell topology as shown in Figure 6.1 with one TCP flow (created by *iperf* tool) and one TFWC or TFRC, respectively per experiments. Initially starting with TFWC or TFRC flow, we then join TCP connection around 20 sec in the experiments; we have used the `foreman` clip as an input images in the experiments. We then measure the throughput at the bottleneck node using *tcpdump* tool and plot the time-averaged throughput (the bin size, we used, is 100 ms – roughly, an RTT in this scenario.). Figure 6.11 and Figure 6.12 show the fairness of TFWC or TFRC when competing with a single TCP flow. The results indicate that both TFWC and TFRC are quite fair as the competing TCP flow enters the network.

TFWC in Figure 6.11(a) and TFRC in Figure 6.12(a) have reached their full bit rate (around 3 Mb/s) before a TCP has entered the network. As soon as TCP connection has started, both TFWC and TFRC reduced its sending rate and then reached an acceptable level of fair state – TCP in both cases tried to reach a fair share (2.5 Mb/s) continuously, but backed off a little more than that of TFWC and TFRC. When the bottleneck link speed is 2.5 Mb/s, TFWC has converged to a fair state perfectly in a very short time (within 2∼3 sec) as shown in Figure 6.11(b). TFRC, however, still showed a little aggressive when competing to a TCP flow in the same condition. The throughput of TFRC, on the other hand, appeared to be very smooth before TCP flow joined the network (i.e., when it reached the steady state, around 15∼20 sec) as shown in Figure 6.12(b). Despite a little varying appearance, the throughput dynamics experiments substantiate that TFWC and TFRC provide a good level of fairness and smoothness in the measured throughput.

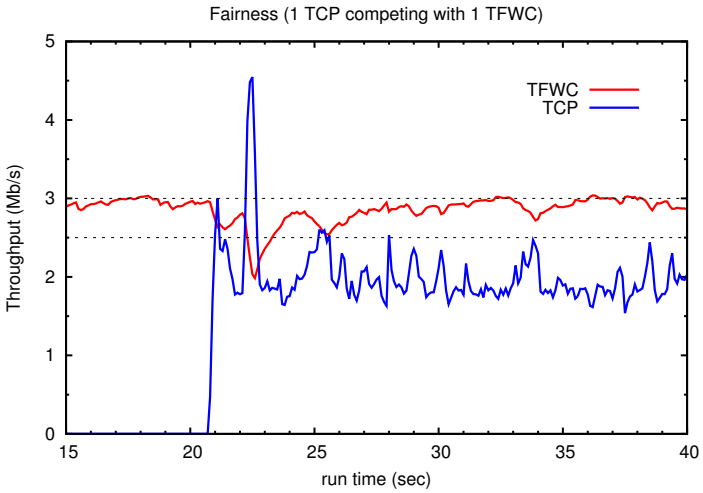
6.5.2 Responsiveness

As discussed in Chapter 4, it would be easy to create an arbitrary congestion control protocol that can achieve smooth throughput property extremely well (even similar to a CBR-like throughput) but at the expense of losing responsiveness. Such excessively slow-responsive congestion control protocols would not cut the sending rate properly when packet losses occur, resulting in a persistent high loss rate in a longer term. It is therefore difficult to say these protocols serve the function for controlling congestions. Furthermore, we cannot give up the responsiveness, particularly for the interactive applications. It is because the usefulness of congestion control mechanisms comes from a real-time feedback on the network conditions to the multimedia codecs so they can change necessary parameters appropriately. When congestion control protocols forget the responsiveness, simply trying hard to achieve smoothness, then the codec cannot change parameters in time, resulting in a poor performance overall. In addition it must be noted that the modern codecs (e.g,. H.264) compress the original image frames such that frames relay on the previous frames, thus, a persistent packet loss (due to unresponsiveness) can severely degrade the overall performance. Similarly, when there is a sudden network bandwidth increase, one would like to fully utilize them as they become available in order to improve image quality, etc. Without responsiveness, it would require prolonged time to reach a full capacity.

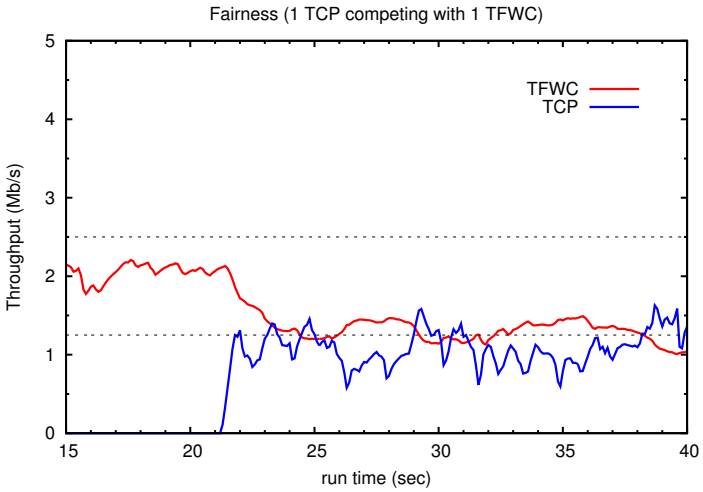
In this section, we verify the responsiveness of TFWC and TFRC by studying an impulse response using on-and-off CBR sources. We create two groups of experiments. First, TFWC and TFRC run at their full bit rates (i.e., 3 Mb.s) in a bottleneck of 5 Mb/s over the HEN testbed (shown in Figure 6.1). A CBR connection then joins the network taking up 80% of bandwidth for the 4 seconds; with 5 Mb/s of bottleneck, the CBR rate becoming 4 Mb/s in this scenario. The responsiveness results of TFWC and TFRC are shown in Figure 6.13(a) and Figure 6.14(a).

For the second group of experiments, we use the same topology as shown in Figure 6.1 with a bottleneck link speed being 2.5 Mb/s. In this test scenario, one CBR flow enters the network with the rate of 1.25 Mb/s (50% of the bottleneck bandwidth), and runs 4 seconds of duration. The results are shown in Figure 6.13(b) and Figure 6.14(b), respectively.

The two groups of experiments showed the responsiveness of TFWC and TFRC to be quite reason-

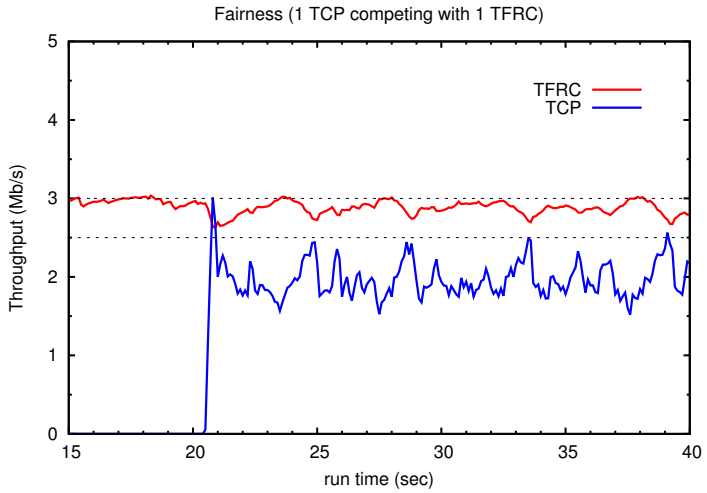


(a) one TCP and one TFWC fighting for a bandwidth of 5 Mb/s with an RTT of 80 ms.

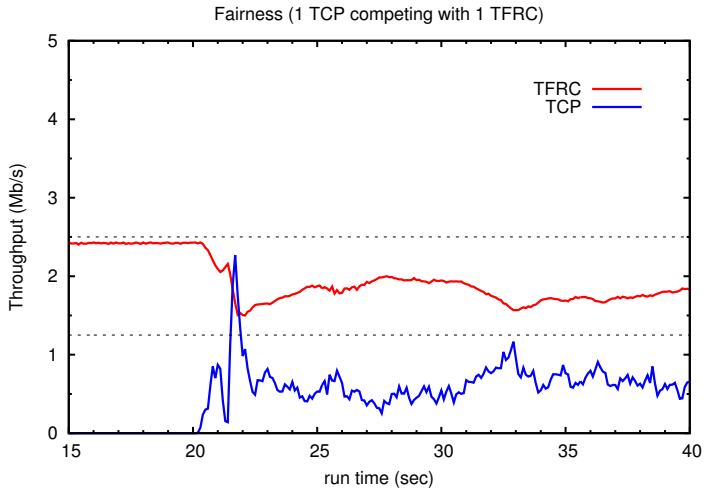


(b) one TCP and one TFWC fighting for a bandwidth of 2.5 Mb/s with an RTT of 80 ms.

Figure 6.11: Fairness – one TCP flow competing with one TFWC flow



(a) one TCP and one TFRC fighting for a bandwidth of 5 Mb/s with an RTT of 80 ms.



(b) one TCP and one TFRC fighting for a bandwidth of 2.5 Mb/s with an RTT of 80 ms.

Figure 6.12: Fairness – one TCP flow competing with one TFRC flow

able – they cut the rate in a second manner and come back to the full capacity quickly when they can. When the first impulse entered the network, their responsiveness revealed to be a little slower than when the second CBR entered. It is due to the fact that the relatively large history information in the averaging filter led them to being a little insensitive in reducing the sending rate than those when the second insertion of the CBR flow. Nevertheless, they largely appear to have similar responsiveness behaviors: both TFWC and TFRC use the same equation after all.

6.5.3 Section Summary

In this section, we have evaluated the network-oriented metrics, chiefly on the protocol's fairness and responsiveness aspects using the Vic tool. Both TFWC and TFRC achieved the required functions in that they acquired a flow-level throughput fairness while being responsive over a sudden change in the available bandwidth. It is worth noting that they appeared to be quite unfair in similar network environments in the simulations, especially for TFRC, whereby TFRC still being a little aggressive in the real-world experiments. While the level of fairness is acceptable, with the high statistical multiplexing, this side effect of TFRC would be diminished, attaining to a fairness level akin to the simulation cases.

6.6 User-oriented Performance Evaluation

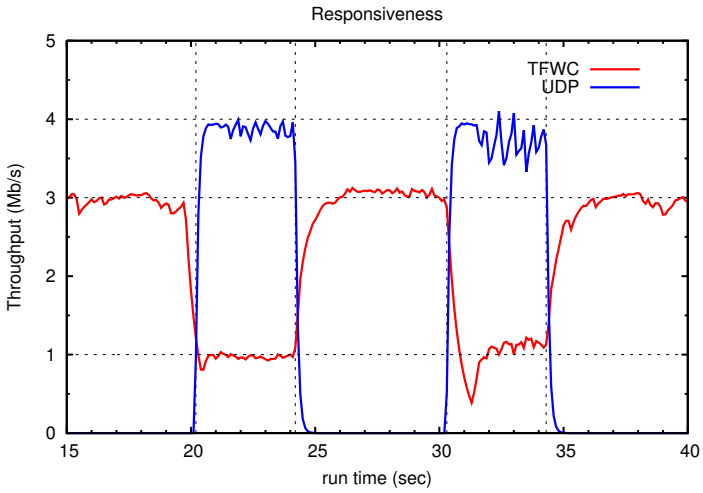
In this section, we evaluate TFWC and TFRC using the performance measure that better reflects the perceived end-user quality in the video conferencing system. To this end, we select two commonly used metrics, PSNR and MOS, for the discussion and evaluation.

6.6.1 PSNR

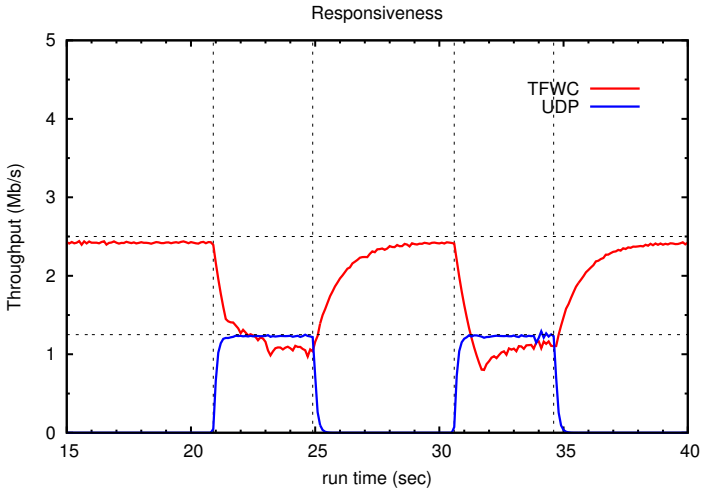
It is widely known that the PSNR has limitations to be used for the video quality measures, and we have briefly discussed them in Section 6.1.3. As explained in the section, the validity of PSNR is still largely agreed when used with the same code and source material. In our experiments, we use the H.261 codec with foreman video clip for all of the experiments, making it useful to evaluate two protocols using the PSNR metric.

The experiments use the topology illustrated in Figure 6.1 as well, but this time there is no competing flow. We set up four scenarios for the experiments for the PSNR evaluation. The first and second experiments use the bottleneck of 5 Mb/s and 2.5 Mb/s with an RTT of 80 ms, respectively, whereas the third and forth cases use 5 Mb/s of bottleneck link with a link loss rate of 0.1% and 1%, respectively. We then compute the PSNR using Equation (6.1) when the Vic receiver renders each frame, then plot them together with the measured throughput. The PSNR results for TFWC, TFRC, and the bare Vic (i.e., without a congestion control mechanism applied) are shown in Figure 6.15, Figure 6.16, and Figure 6.17, respectively.

The best results for the average PSNR are achieved when used with the bottleneck of 5 Mb/s for all three cases (i.e., with TFWC, TFRC, or without congestion control), as shown in Figure 6.15(a), Figure 6.16(a), and Figure 6.17(a). The average PSNR is close to 40 dB in the case of TFWC and TFRC, whereas PSNR with no congestion control indicates 35 dB on average. Although this average PSNR still represents good quality according to Table 6.2, it must be noted that the average throughput is

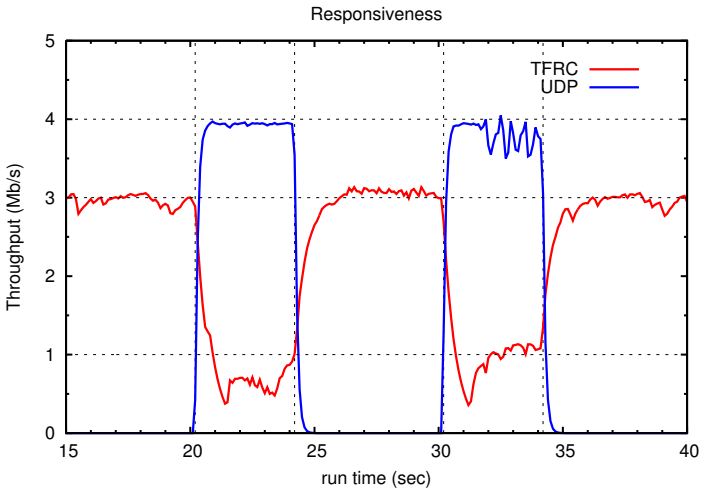


(a) TFWC Responsiveness – 5 Mb/s

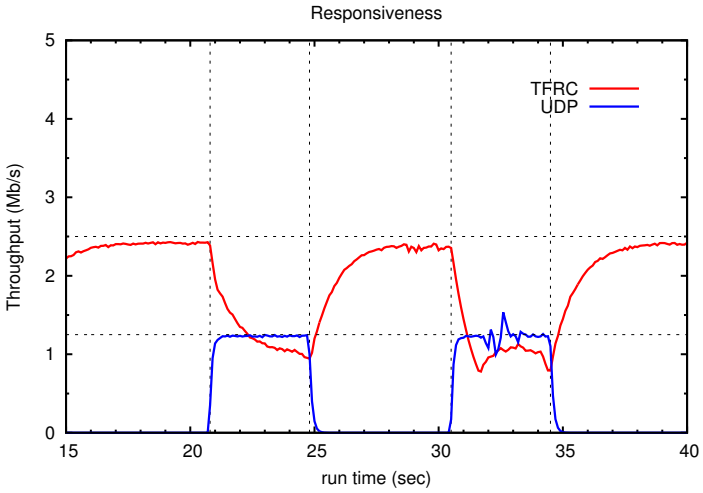


(b) TFWC Responsiveness – 2.5 Mb/s

Figure 6.13: TFWC Responsiveness



(a) TFRC Responsiveness – 5 Mb/s



(b) TFRC Responsiveness – 2.5 Mb/s

Figure 6.14: TFRC Responsiveness

approximately half of those seen in the TFWC and TFRC cases when congestion control is not applied. Therefore, although the image fidelity between the references and the received frames remains somewhat at high marks, the overall perceived image quality would be worse than those with TFWC and TFRC options.

On the other hand, the worst results are shown in Figure 6.15(b) and Figure 6.16(b), which use 2.5 Mb/s bottleneck link. Although the average PSNR comes into 35 dB roughly, they fluctuate severely during the entire experiment, resulting in a disrupted image quality. The main reason for this result is due to the fact that a subtle change in the q factor can have a significant impact on the encoded bit rate for the image frame, whereby our q factor adjusting method (Algorithm 5) turned out to be a little crude in this scenario. Thus, the product of changing the frame rate and the q factor simultaneously has resulted in drastic changes in the send buffer length, meaning the send queue filled up quite quickly with a decreasing q factor, causing it to skip a few consecutive frames thereafter, but suddenly the send queue being drained because *cwnd* allowed the transmission of packets all at once. We can have two options to mitigate this artifact:

- Implement more accurate control mechanisms in changing the q factor.
- Use just one control method when controlling the send buffer – for example, fix the frame rate and manipulate the q factor only.

For the first option, one can apply a more sophisticated control theory to control the send buffer. For example, any combination of Proportional-Integral-Derivative (PID) controllers might just solve the problem. Nonetheless, to devise such a control loop could be another research question – i.e., it is not our research goal to answer the question in this thesis. For the second option, we have fixed the frame rate to be 15 fps throughput the experiment, applying the q factor control method alone. The results shown in Figure 6.18 indicate that the PSNR sustained quite stable, as high as 40 dB on average, for both TFWC and TFRC cases. There is a related work on the evaluation of changing the frame rate and *quantizer* [50], explaining that frame skipping (e.g., effective frame rate down to 15, 12 or even less fps values) does not have a considerable effect on the acceptability of the encoded image quality. This work is based on a pre-determined q factor, thus, our approach (i.e., increase/decrease *quantizer* on the fly) appear to be of less impact to the user's final impression with these lower frame rates. From this study, it is worth noting that a slight q factor change can lead to substantial variations on the encoded bit rates, hence, can significantly affect the final image quality as well[9].

We also have conducted when there are link losses. The results shown in Figure 6.15(c), Figure 6.16(c), and Figure 6.17(c) used the bottleneck link with 0.1% random losses, whereas Figure 6.15(d), Figure 6.16(d), and Figure 6.17(d) with 1% of link loss rate. With 0.01% loss rate, the PSNR values remained at 40 dB on average when TFWC and TFRC were applied. When the loss rate is a little higher (1% loss rate), the overall PSNR indicated 32 dB on average for the both cases. Without applying congestion control mechanisms, the PSNR remained standing at similar values, but the video

[9]Unlike MPEG image compression algorithm, the H.261 codec simply uses a universal *quantizer* value to all frequency components of images in Discrete Cosine Transform (DCT) computation, which results in the drastic bit rate changes on a subtle q factor tuning.

bit rates were much lower than those with TFWC and TFRC, which represent much poor image quality.

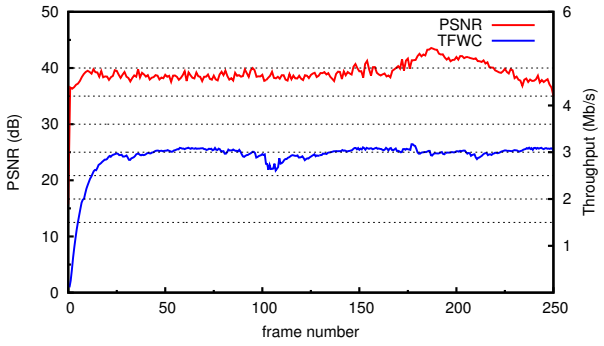
6.6.2 MOS

In this subsection, we present Mean Opinion Score (MOS) by mapping the calculated PSNR values using Table 6.2 to better illustrate the perceived image quality. We extracted all PSNR values from the same experiments to those with Figure 6.15, Figure 6.16, and Figure 6.17. The MOS results are shown in Figure 6.19, where TFWC1 corresponds to the case shown in Figure 6.15(a), and, similarly, TFRC1 in Figure 6.19 to the case shown in Figure 6.16(a) and so forth. The x-label explanations are summarized in Table 6.5.

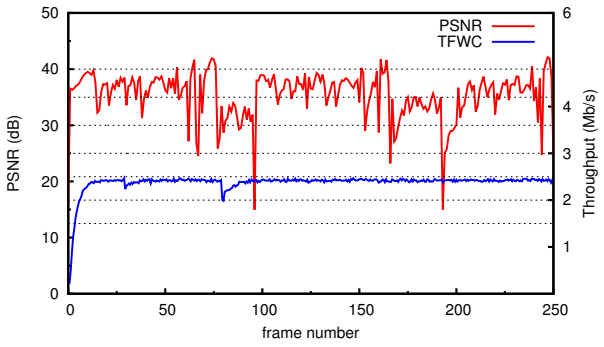
As we can see, the PSNR remained best for the TFWC1 and TFRC1 cases – most of the received frames indicated extremely high value (above 37 dB). The Vic also revealed relatively high PSNR (between 31 and 37 dB). However, as we mentioned, the received bit rate was about half of TFWC and TFRC, resulting in the user's final impression being deteriorated. Overall, TFWC appeared to be slightly better than those of TFRC in all of the four experiment scenarios[10]. However, it is difficult to claim that TFWC outperforms TFRC in this regard as TFRC produced the excellent PSNR results as well.

From the PSNR and MOS studies, the benefit of using congestion control mechanisms is apparent in that they certainly helped the application to deliver better image quality in the experiments, while still being soundly fair and responsive to the competing TCP flows.

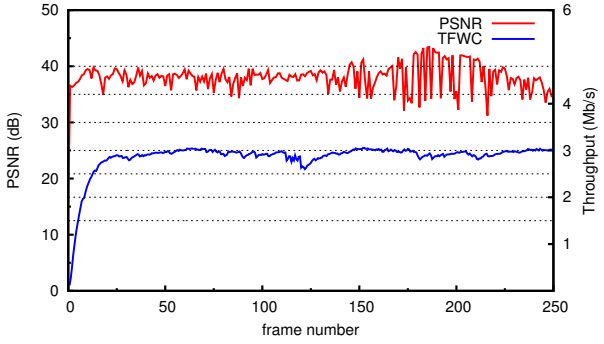
[10]They have reached almost equal bit rates in all cases, thus, we might be able to say TFWC provided a better image quality.



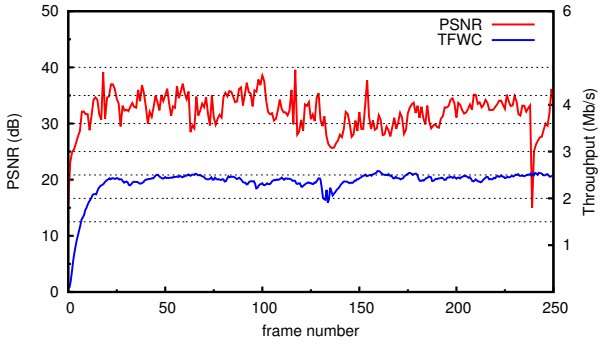
(a) TFWC PSNR – 5 Mb/s



(b) TFWC PSNR – 2.5 Mb/s

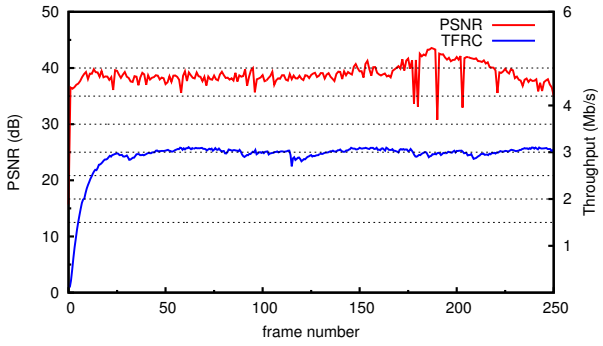


(c) TFWC PSNR – 5 Mb/s with 0.1% link loss rate

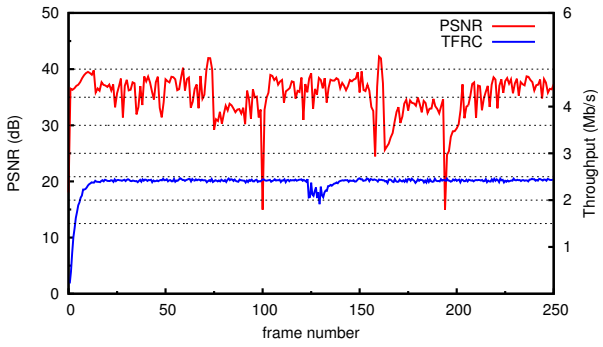


(d) TFWC PSNR – 5 Mb/s with 1% link loss rate

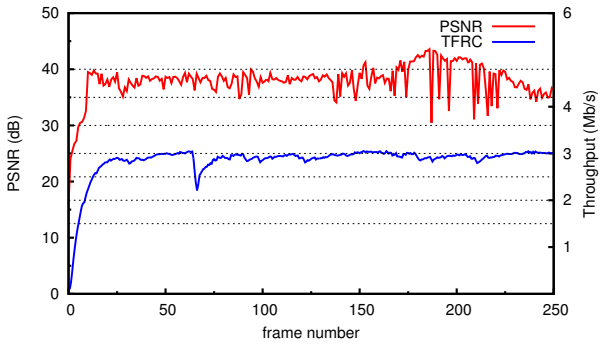
Figure 6.15: TFWC PSNR



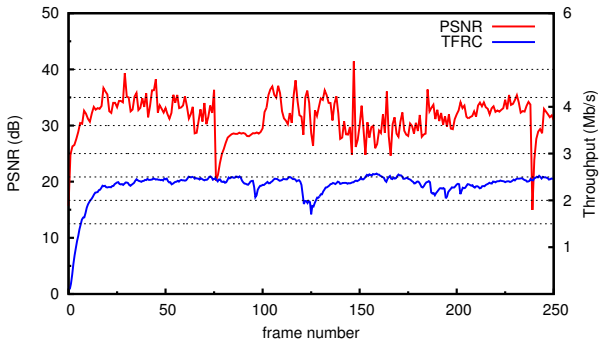
(a) TFRC PSNR – 5 Mb/s



(b) TFRC PSNR – 2.5 Mb/s

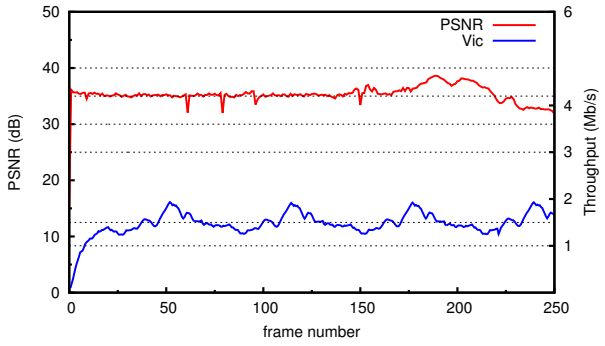


(c) TFRC PSNR – 5 Mb/s with 0.1% link loss rate

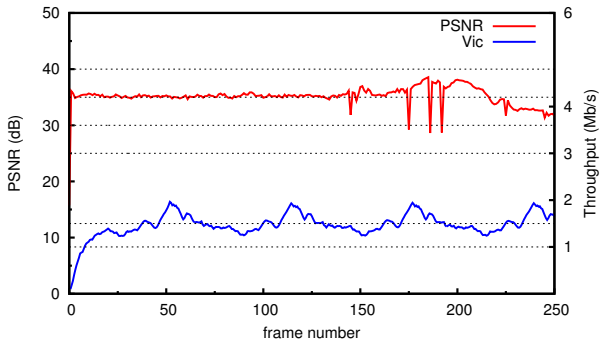


(d) TFRC PSNR – 5 Mb/s with 1% link loss rate

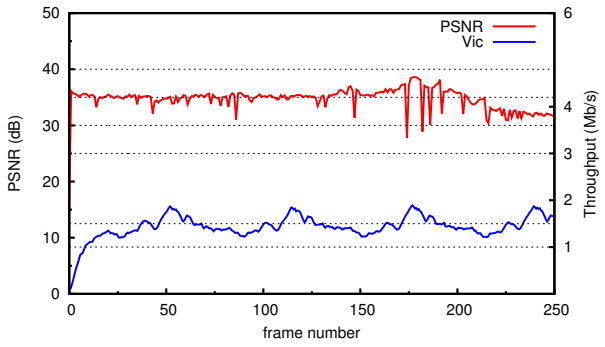
Figure 6.16: TFRC PSNR



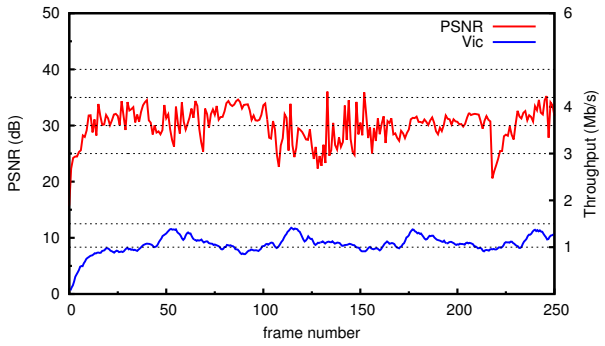
(a) Vic PSNR – 5 Mb/s



(b) Vic PSNR – 2.5 Mb/s

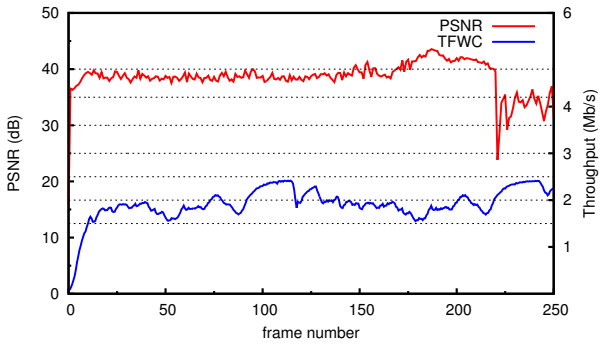


(c) Vic PSNR – 5 Mb/s with 0.1% link loss rate

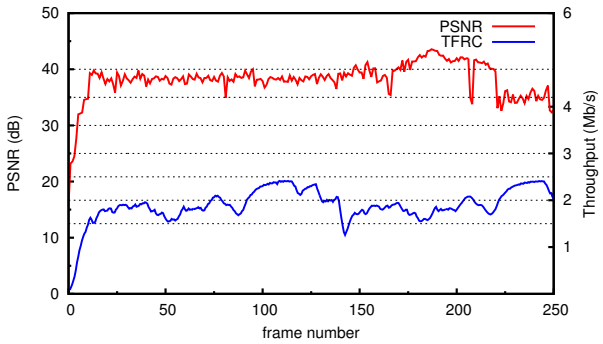


(d) Vic PSNR – 5 Mb/s with 1% link loss rate

Figure 6.17: Vic PSNR without Congestion Control



(a) TFWC PSNR – 2.5 Mb/s



(b) TFRC PSNR – 2.5 Mb/s

Figure 6.18: TFWC/TFRC PSNR with the fixed frame rate (fps = 15).

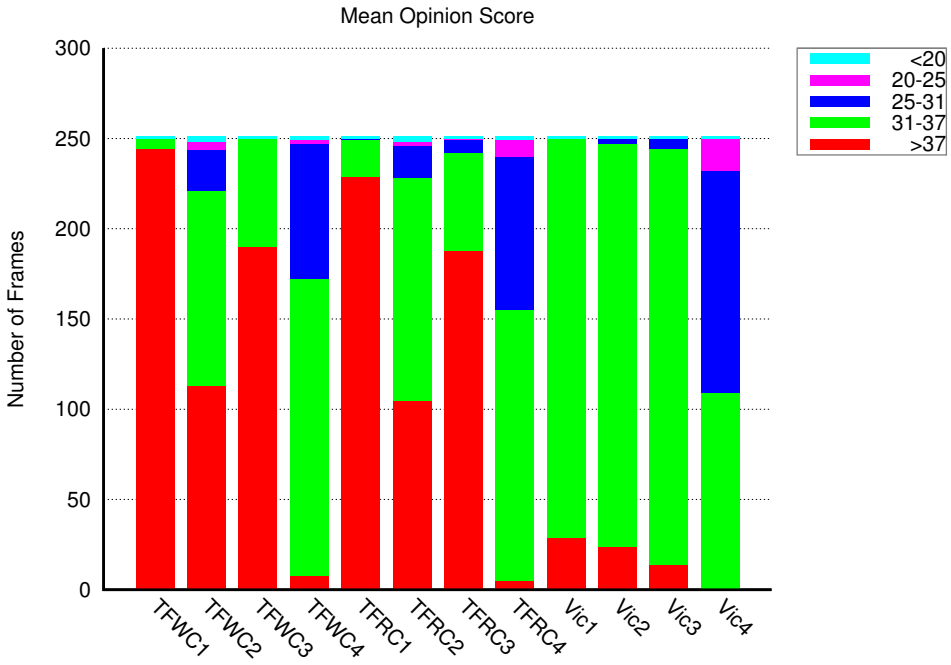


Figure 6.19: Conversion PSNR to Mean Opinion Score

Table 6.5: Explanation of x-label for MOS Figure 6.19

x-label	Corresponding Experiment
TFWC1	Figure 6.15(a)
TFWC2	Figure 6.15(b)
TFWC3	Figure 6.15(c)
TFWC4	Figure 6.15(d)
TFRC1	Figure 6.16(a)
TFRC2	Figure 6.16(b)
TFRC3	Figure 6.16(c)
TFRC4	Figure 6.16(d)
Vic1	Figure 6.17(a)
Vic2	Figure 6.17(b)
Vic3	Figure 6.17(c)
Vic4	Figure 6.17(d)

6.7 Conclusion

In this chapter, we have extensively evaluated our proposed congestion control mechanisms, TFWC, and the IETF standard, TFRC, using the Vic tool over typical network settings that most users might generally experience. To this end, we have introduced two categories of the performance measures: *network-oriented* and *end-user oriented* metrics, respectively.

The main objectives and benefits of congestion control mechanisms chiefly stem from the need in that users demand better quality. Therefore, congestion control mechanisms aim at providing useful information to the multimedia applications effectively within required time limits, so that they can adjust various parameters to deliver better media quality. At the same time, congestion control mechanisms strive to prevent from a congestion collapse by regulating send rate in the form of a rate-based or window-based controller. In short, they fulfill two functions:

- hand over useful information to higher-layer applications.
- prevent from a congestion collapse by avoiding persistent packet losses.

We have validated and discussed the basic functions and requirements as a congestion control protocol for the use of interactive applications in Chapter 4. To cross check the validity of these two protocols over a real-world application (i.e., to inspect the functions whether they can provide useful information to the applications, and to figure out its practicality), we developed two control loops, namely, frame-rate and q factor control, respectively. In fact, the combination of two control methods turned out to be quite beneficial to the overall performance of the Vic tool, although not all cases worked perfectly. There have a certain regime that these control revealed a little crude, but the artifact can be easily diminished by fixing the frame rate, applying the q factor control scheme alone.

The various PSNR results and mapping to MOS showed enough evidence that the congestion control mechanisms have introduced significant improvements on the perceived image quality. However, it is unclear as to which of those congestion control mechanisms outplay the other in the real-world experiments.

Chapter 7

Conclusion

In this thesis, we have proposed and examined a window-based congestion control protocol for real-time interactive multimedia streaming applications. We started with a *curiosity* as to why those congestion control mechanisms are not widely deployed in real applications, while it has been lying at the center of the Internet for the normal data transfer. We then developed an idea that the rate being inaccurate might have failed to bring enough attention to the application writers, which led us to propose a window-based control protocol which can still provide nice properties as rate-based protocol has promised (e.g., smoothness, fairness, stability, and responsiveness). We have also implemented TFWC and TFRC over the Vic tool, and conducted various real-world experiments for the performance analysis in seeking the answers that we asked in Chapter 1.

We showed that our proposed mechanism, TFWC, can outperform TFRC in various networking settings through extensive *ns-2* simulations. A rate-based congestion control faces practical implementation issues with timer granularity when the RTT is small, like a local LAN environment, whereas TFWC works just like TCP with its `Ack`-clocking mechanism and does not suffer from this limitation, giving better fairness (see Figure 4.3 and Figure 4.4). These fairness graphs indicate that TFRC significantly starved TCP sources when the bottleneck bandwidth was small (DropTail queue at the bottleneck), whereas TCP received a little more throughput than TFWC in the same condition. On the other hand, TFWC showed almost perfect fairness when competing with TCP sources (RED queue at the bottleneck), whereas TFRC still largely starved TCP sources. It shows that TFWC is more resilient in reaching a fair share of the link regardless of the bottleneck bandwidth. For the protocol stability analysis, both TFWC and TFRC indicated a high CoV values when the bottleneck bandwidth is small with a large number of sources. This can be seen as a side effect such that the TCP equation did not correctly model the TCP's timeout behavior. Therefore, it does not appear to be the protocols' artifact but the equation simply driving them that way: after all they are using the same equation. For the smoothness, TFWC and TFRC were equally smooth when TFWC enabled the jitter mode. The protocol responsive results shown in Figure 4.9 indicates that TFWC and TFRC cut their sending rate coequally, whereas TFWC exhibited a little faster when catching up the available bandwidth.

The extensive simulation results explain that TFWC can indeed perform better (i.e., fairer in sharing

bandwidth especially with RED queue at the bottleneck, equally smooth, and a little better in responsiveness), giving a reasonable option to the application writers when they have a choice between a rate-based and a window-based. To cross-check its validity, we have implemented both protocols over the Vic tool.

To implement the `Ack` mechanism, we have used the RTCP XR packet format which delivers AckVec and AoA information between a sender and receiver. We also have considered packet reordering, whereby TFWC and TFRC revert the ALI and its timestamp when necessary. For TFRC, the correct measurement of RTT is critically important. As the Vic system and the kernel buffer can potentially add arbitrary delay, we have used the socket option (`SO_TIMESTAMP`) that enables to parse the socket timestamp, a timestamp that indicates the arrival of a packet at the socket buffer. We also have introduced a finer-grain *cwnd* computation mode: *cwnd* in bytes. This gives more control to our window-based congestion control protocol. Finally, we converted the TFRC to a sender-driven mode, partly for a fair comparison.

With this system, we have conducted various experiments over the HEN testbed, and evaluated their performance in two categories: *network oriented* and *end-user* oriented metrics. To this end, we have considered the following metrics:

- ***Network Oriented***
  - ***Fairness***

    Figure 6.11(a) and Figure 6.12(a) shows that TFRC is a little aggressive than TFWC – this is coherent outcomes as in the simulation. On one hand, Figure 6.11(b) and Figure 6.12(b) shows that TFWC is much fairer than TFRC – the bit rate is limited by the bandwidth, but as the competing TCP entered the network, TFWC reached a fair share in a matter of second, whereas TFRC briefly maintained the fair state, but took in majority of the bandwidth thereafter.

    One thing worth noting is in the steady state (Figure 6.11(b) and Figure 6.12(b)), TFWC did not quite fill the pipe, whereas TFRC filled the link in its full capacity.
  - ***Smoothness***

    From Figure 6.11(b) and Figure 6.12(b), the TFRC throughput smoothness in the steady state showed better than those of TFWC.
  - ***Responsiveness***

    Figure 6.13 and Figure 6.14 shows that their responsiveness are not substantially different.
- ***End-user Oriented***
  - ***PSNR***

    It is widely known that PSNR does not capture the quality of video *streams*, whereas it is still used for image-by-image quality measure. In our presentation (in Chapter 6), we also plot the measured throughput together with PSNR so that it could be a little more meaningful. As we explained, although the PSNR values indicate somewhat high in the case of

no congestion control (Figure 6.17), the bit rate it received is half of those with TFWC and TFRC. Therefore, we could argue the congestion control actually helped generate more bits to increase image quality, which then can infer as higher image quality.

- ***MOS***

  The computed PSNR can be mapped to MOS value using Table 6.2. From the PSNR and MOS studies as shown in Figure 6.19, the benefit of using congestion control mechanisms is evident.

Through the various real-world experiments as well as the *ns-2* simulations, we would like to conclude by answering our initial questions:

- Yes, the congestion control mechanisms certainly helped providing better image quality by probing the available bandwidth, or by decreasing the sending rate reasonably well while they are fairly competing with the standard TCP flows.
- Although TFWC seemed a little better than TFRC in terms of giving better fairness in the simulations and experiments, the difference between TFWC and TFRC did not stand out significantly.
  - TFWC was better in terms of fairness (both the simulations and experiments)
  - TFWC was better when catching up available bandwidth (in the simulations)
  - TFRC was better in terms of smoothness (both the simulations and experiments)
  - TFRC was better in reaching its full bandwidth cap (in the experiments)

Therefore, this thesis concludes that with a congestion control mechanism applied, there is high potential to improve overall quality for interactive multimedia applications, but the choice of congestion control protocols remains an open issue. Having our proposed algorithm, TFWC, it assuredly adds options. Among them, what we definitely buy is the easiness of implementation: no longer need to worry about the interrupt timer granurarity on the host.

7.1 Future Work

We have mentioned in Chapter 2 and Chapter 3 about the rate mis-match using TFRC in real situations. This was not demonstrated in this thesis due to the limitations of the infrastructure in our lab facilities. This issue is worth considering as a next part of the work for the comparison between TFWC and TFRC.

Appendix A

TFWC Protocol Validation

This chapter describes the detailed processes in checking the TFWC protocol's functional components. We brief an overview of the components that need validation, and then presents the results both with the *ns-2* simulator and Vic tool.

A.1 Overview

As mentioned in Section 3.2 and Section 3.3, there are fundamental but important components to be validated before conducting the performance evaluation. These are:

- AckVec mechanism
  - Does the AckVec at a receiver side construct the received packet sequence numbers correctly?
  - Does the sender interpret the received AckVec correctly when determining which packet sequence numbers are missing?
  - Does the AckVec trim well once in a while upon reception of AoA?
- ALI computation
  - Does the ALI calcuation from the received AckVec reflect the loss interval correctly?
  - Is the calculated loss event rate (p) correct?
- *cwnd* computation
  - Is the computed *cwnd* correct?
  - Are all the necessary steps (e.g., hybrid win-rate and *cwnd* jitter) applied in the correct way?
- TFWC Timer (to update RTT and RTO)
  - Is the updated RTO and RTT correct?
  - Does the timer go off correctly when it expires?

Among them, AckVec mechanism is critically important as `Ack` essentially determines the ALI and *cwnd* computation. Therefore, we mainly validate AckVec mechanism so that we can cross check the rest of the core procedures. In fact, we test and check ALI information to validate all the necessary mechanisms have worked correctly. To this end, we artificially disturb transmitting specific packet sequence number(s) periodically at the sender so as to emulate deterministic packet losses, as explained in Section 4.1.1: constructed losses.

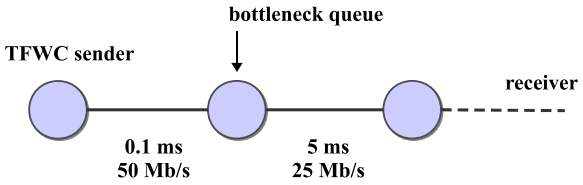


Figure A.1: Simulation Topology for Protocol Validation

A.2 Validation over *ns-2* simulator

In *ns-2*, we validate the protocol's functionality in three-fold: AckVec mechanisms, ALI computation, and *cwnd* calculation. The simulation topology for the validation process is composed of one bottleneck node and a TFWC sender and receiver pair. We assume that the application data is always available. Figure A.1 shows the topology used for the entire validation scenarios. In the following sections, we verify ALI construction mechanisms and *cwnd* computation on various packet loss scenarios.

A.2.1 ALI Validation

In this section, we validate ALI mechanisms in the following scenarios:

- validation on deterministic losses
- validation on multiple losses in the same window
- validation on random losses

A.2.1.1 Deterministic Packet Losses

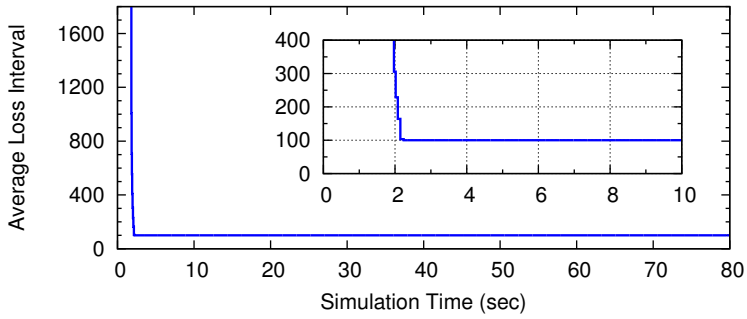
In this subsection, we manually construct a constant packet drop to check if it has been reflected to ALI value accordingly. If we drop the very first packet periodically among one hundred packet transmission, the loss event rate (p) generated by ALI should indicate 0.01 (i.e., 1%). In this test, we do not send the very first packet periodically among a hundred packet transmission. Likewise, we do not send the first packet among 20 and 10 packet transmission, respectively, introducing 5% and 10% packet loss rate. As shown by the test results in Figure A.2, ALI has quickly converged to the values as expected. The validation results prove that ALI mechanisms to be functioning correctly, showing 100, 20, and 10, respectively, according to the loss event rate of 1%, 5%, and 10%.

A.2.1.2 Multiple Packet Losses in a Single Congestion Window

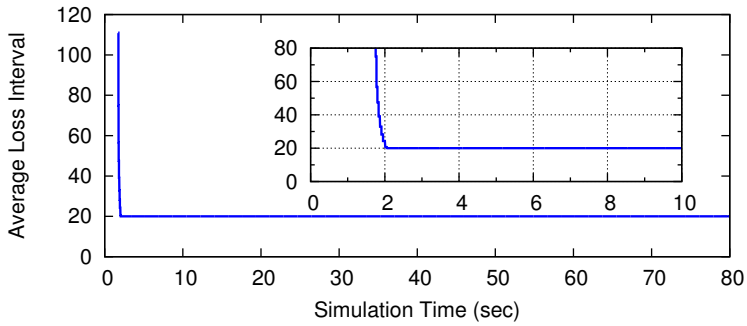
In this subsection, we consider the cases where multiple packet losses occur in the same window. We validate that the sender does not initiate a new loss history when multiple losses occur in the same window. To this test, we create the following two scenarios:

- two back-to-back packet losses in the same window – as shown in Figure A.3(a).
- one packet loss after one successful packet transmission – as shown in Figure A.3(b).

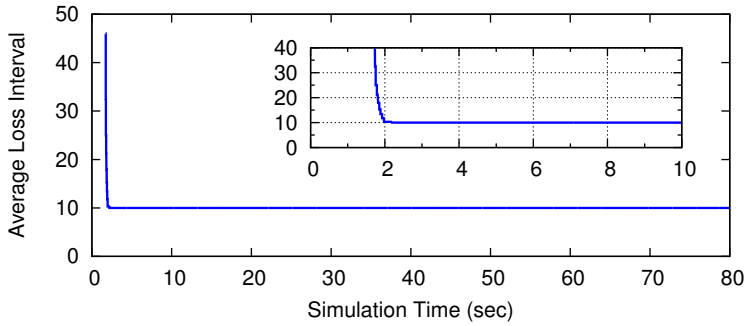
For the back-to-back packet loss case, we artificially drop back-to-back packets among one hundred packet transmissions, resulting in the *cwnd* size of 10 roughly. In this test, we expect ALI indicates 100, just like the 1% of constant packet loss rate case. For the bi-packet loss case, we alternate in sending a packet right after one successful packet transmission.



(a) Loss Event Rate = 0.01

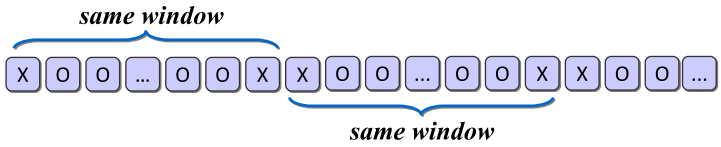


(b) Loss Event Rate = 0.05

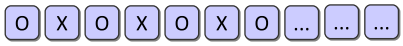


(c) Loss Event Rate = 0.1

Figure A.2: TFWC ALI validation with deterministic losses (1%, 5%, and 10%, respectively).



(a) Two back-to-back packet losses



(b) Bi-packet losses

Figure A.3: ALI test scenarios that manually constructed packet losses, where O indicates successful packet transmission, and X unsuccessful packet transmission, respectively.

Figure A.4(a) shows the ALI result of the back-to-back packet losses in the same window. In this case, *cwnd* should indicate 100: exactly the same as the case in Figure A.2(a). In case of bi-packet losses, the ALI would become 2, representing 50% of loss rate (shown in Figure A.4(b)).

A.2.1.3 Random Packet Losses

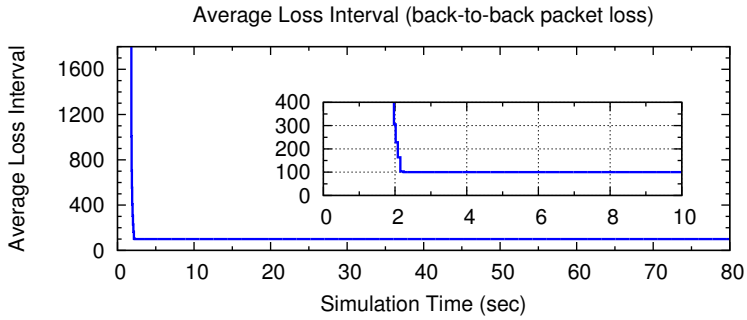
In this subsection, we randomly drop packet(s) to result in the loss rates of 1%, 5%, and 10%, respectively. In this test scenario, for example, we randomly drop one packet among a hundred packet transmission to introduce 1% packet loss rate. This result should be identical to the one presented in Figure A.2(a). Likewise, we drop one packet among 20 packet transmission, and among 10 packet transmission, respectively, to introduce the loss rate of 5% and 10% accordingly. As we can see the results in Figure A.5, they are identical to those in Section A.2.1.1.

A.2.2 *cwnd* Validation

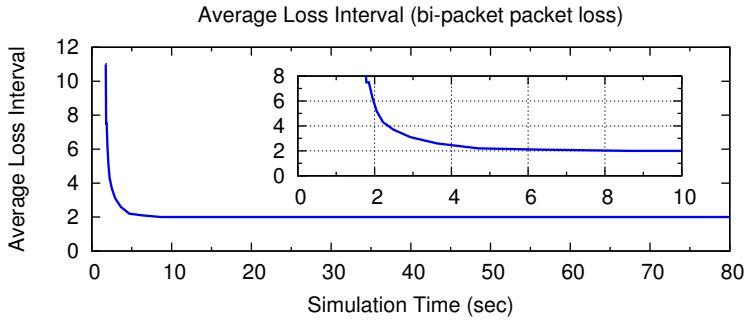
In this subsection, we validate the *cwnd* computation mechanism. As *cwnd* is determined directly from ALI value, we use the same scenarios presented in Section A.2.1.1 and cross compare the two values.

To do so, we first approximate the *cwnd* values for each case. As explained in Equation (6.3) of Section 6.2.1, the dominant term of Equation (3.2) is $\sqrt{\frac{2p}{3}}$ where loss rate is small, so the approximated *cwnd* can be drawn as in Equation (6.3) as follows: $\acute{w} \cong \frac{1}{\sqrt{\frac{2p}{3}}} = \sqrt{\frac{3}{2}}\frac{1}{\sqrt{p}}$. 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 (3.2). So, then, $\acute{w} = 1/f(0.05) = 1/0.271 \cong 3.7$, which reflects our results as in Figure A.6(b). We have rounded up this value to the nearest integer as W can only be an integer value.

When ALI is 100 (i.e., p = 0.01), the *cwnd* indicates 10∼11 packets. Using Equation (6.3), $\acute{w} = 1/\sqrt{0.01} = 10$. Similarly, when ALI is 20 (i.e., p = 0.05), $\acute{w} = 1/\sqrt{0.05} = 4.47 \cong 4$. All these results correspond to our designed mechanisms, hence we can conclude those ALI and *cwnd* calculations to be valid and correct.

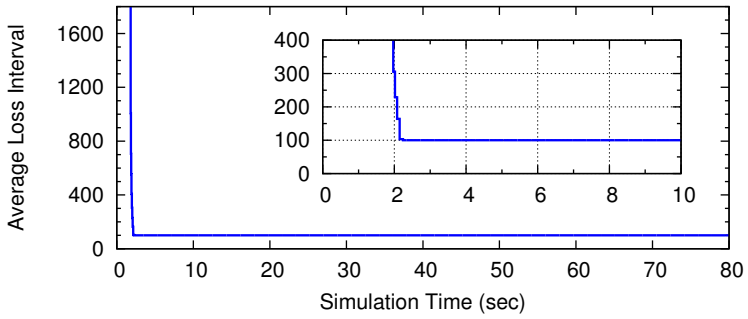


(a) Back-to-back Packet Loss

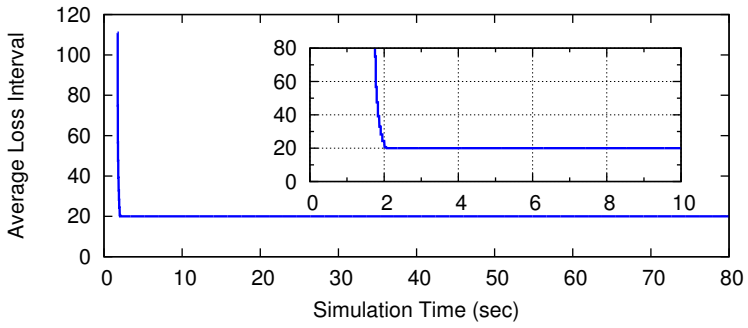


(b) Bi-Packet Loss

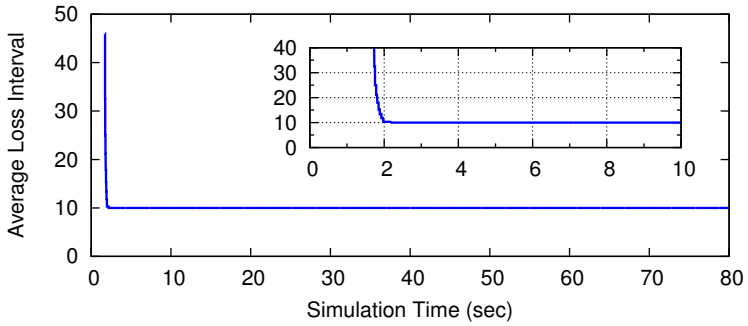
Figure A.4: ALI Validation Test corresponding scenarios shown in Figure A.3.



(a) Loss Event Rate = 0.01

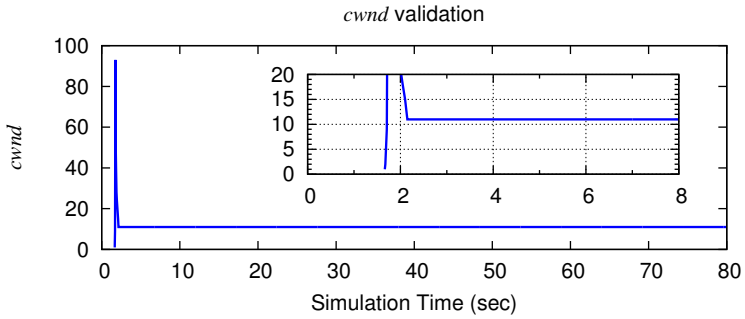


(b) Loss Event Rate = 0.05

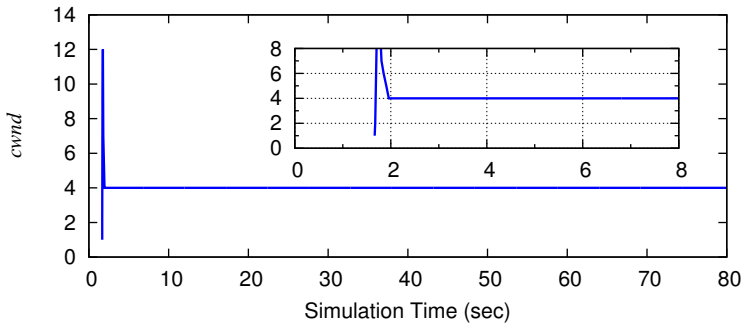


(c) Loss Event Rate = 0.1

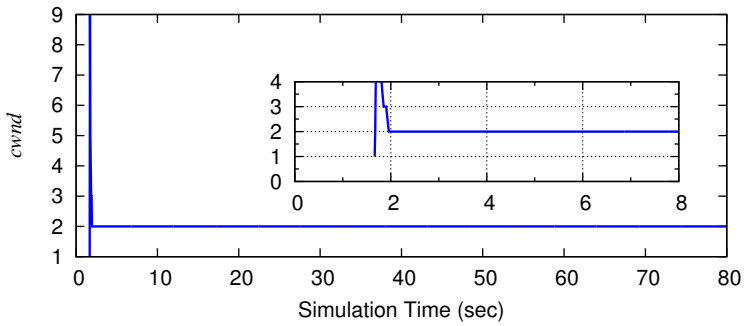
Figure A.5: TFWC ALI validation with random packet losses (1%, 5%, and 10%, respectively).



(a) Loss Event Rate = 0.01



(b) Loss Event Rate = 0.05



(c) Loss Event Rate = 0.1

Figure A.6: TFWC *cwnd* Validation

A.3 Validation over Vic tool

In this section, we validate the ALI mechanism over Vic tool, and verify the *cwnd* calculation is correct. We use the test topology exactly the same in structure shown in Figure A.1, but this time running Vic system. To introduce packet loss rates, as in the simulation studies, we disturb transmitting specific packet sequence numbers at the sender to emulate packet losses as planned.

A.3.1 ALI Validation over Vic tool

Figure A.7 shows the validation results of ALI values per packet loss rates (1%, 5%, and 10%, respectively). The results here are the same as those seen in the simulation cases, shown in Figure A.2. They are:

- when the packet loss rate is 1%, then ALI quickly converges to 100[1].
- likewise, ALI converges to 20 and 10 as per 5% and 1% of packet loss rates, respectively.

A.3.2 *cwnd* Validation over Vic tool

The results in this subsection are from the same set of experiments as in the previous section; we measured the ALI and *cwnd* values together. Thus, the *cwnd* values should correspond to the ALI values when in the same packet loss rates. They are shown in Figure A.8:

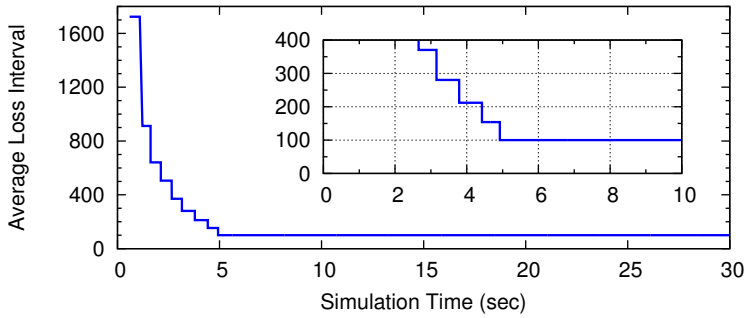
- when the loss rate is 1%, *cwnd* quickly converges to 10[2].
- similarly, *cwnd* converges to 4 and 2 depending on the packet loss rate of 5% and 10%, respectively.

A.4 Summary

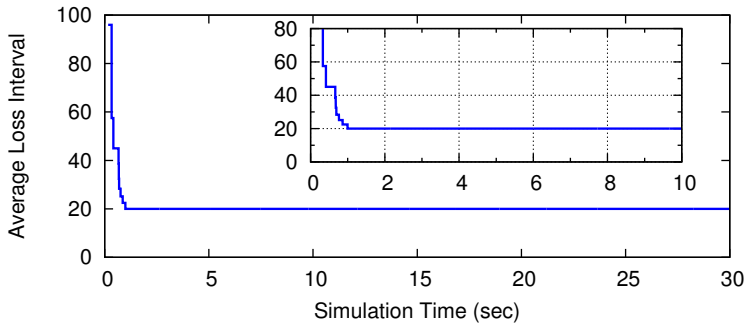
In this chapter, we have validated the core modules (ALI and *cwnd*, and, AckVec) over *ns-2* simulator as well as Vic system. The ALI mechanisms are correctly validated over various deterministic loss rate scenarios, which are also correctly reflected on *cwnd* computation. Because the correctness of ALI computation is due to the correct AckVec mechanisms, we can conclude that the AckVec processes are validated in the given scenarios.

[1]The results correspond to those defined by Equation (3.5).

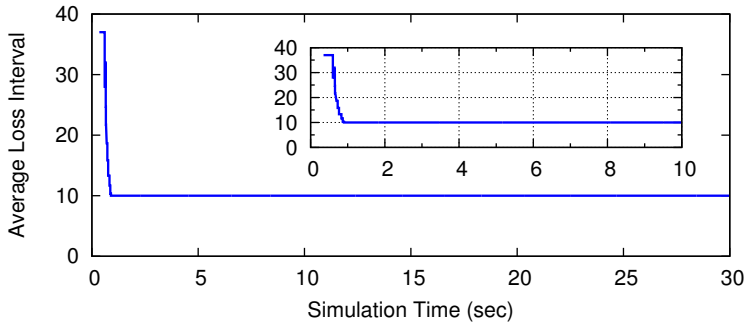
[2]Using Equation (6.3), $\acute{w} \cong \frac{1}{\sqrt{0.01}} = 10$, which is the same result as explained in Section A.2.2.



(a) Loss Event Rate = 0.01

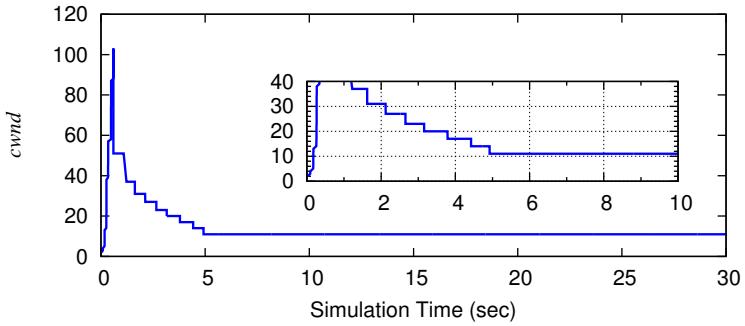


(b) Loss Event Rate = 0.05

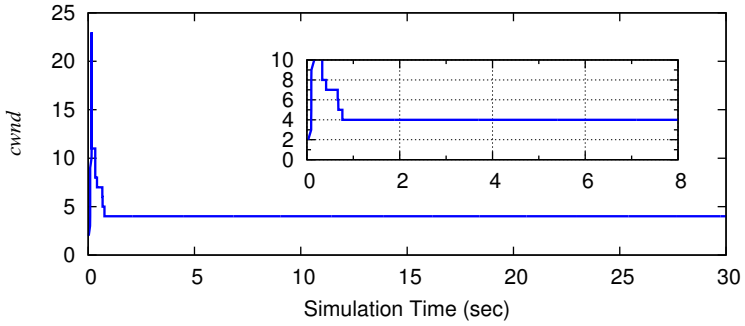


(c) Loss Event Rate = 0.1

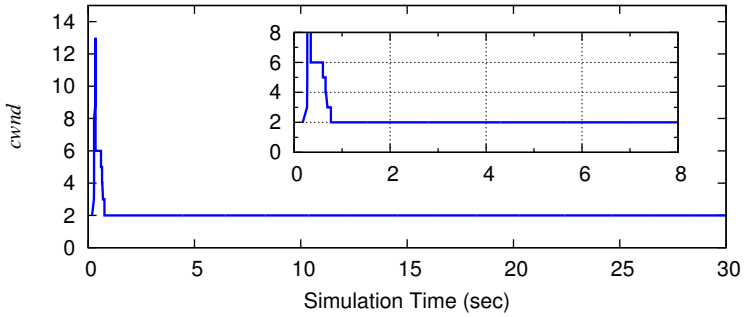
Figure A.7: TFWC ALI validation with deterministic losses (1%, 5%, and 10%, respectively) using Vic tool.



(a) Loss Event Rate = 0.01



(b) Loss Event Rate = 0.05



(c) Loss Event Rate = 0.1

Figure A.8: TFWC *cwnd* Validation over Vic tool.

Appendix B

YUV Image Sequences

This chapter summarizes a brief description of the well-known pre-recorded YUV raw image sequences that have been used in this thesis. These are available for CIF or QCIF[1] image size coded at 30 fps. The descriptions on the content of the files are as given below:

- Akiyo: a Japanese anchor reporting a news
- Bridge-close: a close scene of Charles bridge
- Bridge-far: a far scene of Charles bridge
- Bus: a moving bus
- Claire: a talking woman
- Coastguard: panning of a coastguard ship moving
- Container: a container ship moving slowly
- Football: a high-motion football game
- Foreman: a talking construction worker
- Grandma: a talking old woman
- Hall: an office hallway
- Highway: a scene from a fast car on a highway
- Miss America: a woman talking to a camera
- Mobile: panning of moving toys
- Mother-daughter: Mom and daughter speaking to camera
- News: talking two news reporters
- Paris: two people talking with high-motion gestures
- Salesman: a salesman talking in his office
- Silent: a person demonstrating sigh language
- Stefan: a tennis player
- Suzie: a woman talking over the phone
- Tempete: a moving camera
- Waterfall: a waterfall natural scene

A summary of the above image sequences is shown in Table B.1.

[1] CIF image size is 352 * 288 pixels per frame, and QCIF image size is 176 * 144 pixels per frame.

Table B.1: Summary of YUV images sequences

Name	Resolution	Number of Frames	Motion Complexity
Akiyo	CIF	300	Low
Bridge-close	CIF	2001	Low
Bridge-far	CIF	2101	Low
Bus	CIF	150	High
Carphone	QCIF	382	Medium
Claire	QCIF	494	Low
Coastguard	CIF	300	High
Container	CIF	300	Low
Football	CIF	130	High
Foreman	CIF	300	High
Grandma	QCIF	870	Low
Hall	CIF	300	Low
Highway	CIF	2000	High
Miss-America	QCIF	150	Low
Mobile	CIF	300	Medium
Mom-daughter	CIF	300	Low
News	CIF	300	Low
Paris	CIF	1065	Medium
Salesman	QCIF	449	Low
Silent	CIF	300	Low
Stefan	CIF	90	High
Suzie	QCIF	150	Medium
Tempete	CIF	260	Low
Waterfall	CIF	260	Low

Appendix C

Send Buffer Measurement

In this chapter, we illustrate how to measure the send buffer in Vic tool. In order to calculate the send buffer length at the encoder, we count how many packets are generated in each encoding loop, and how many of them are transmitted during that interval. As we know, every frame size is variable, so the number of packets associated with the frame will also vary. Let's assume the *cwnd* size is 4 packets. As shown in Figure C.1, Vic system allows the transmission of 4 packets while the encoder packetizes the frame (6 packets in this example), leaving 2 packets in the send buffer. So we count the number of packets per frame and also count how many of them are transmitted in that encoding instance: in this case, the send buffer length is 2 by the end of the encoding loop. Then, as `Ack` comes into the sender, Vic will clock out packets and we decrement the send buffer length. In fact, we derive the number of transmitted packets between two encoding instances at the encoder as follows:

$$sent\_more \;=\; end_{(n)} \;-\; begin_{(n+1)} \tag{C.1}$$

Using this method we can approximate the average number of packets per each frame. We maintain a history of these average values (up to K frames), and estimate the average number of packets in a single frame ($\bar{\mathcal{N}}_K$) for the last K history as below:

$$\bar{\mathcal{N}}_K \;=\; \frac{\sum_{n=i}^{K+i} num[n]}{K} \qquad (i = 0,\, 1,\, 2,\, \ldots)\,, \tag{C.2}$$

where K is the maximum history size, and $num[n]$ the number of packets for the nth frame.

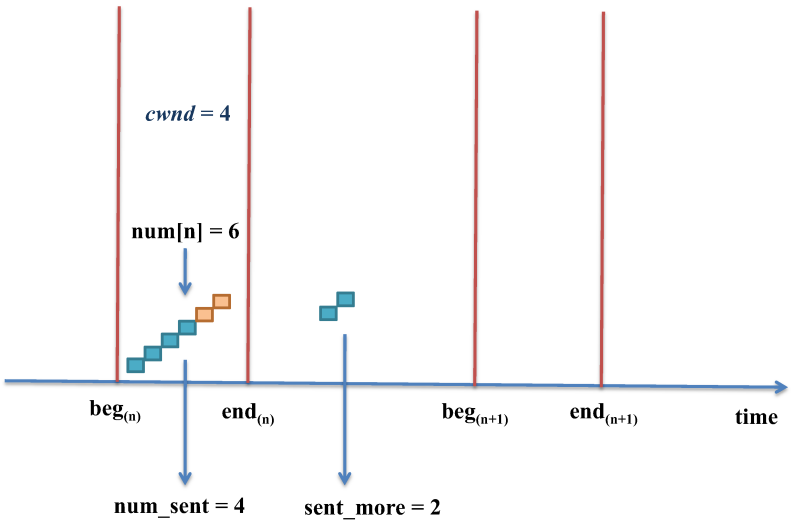


Figure C.1: Send Buffer Measurement: $num[n]$ stands for the number of encoded packets at the nth interval, beg$_{(n)}$ the send buffer length at the beginning of the nth encoding instance. Likewise, end$_{(n)}$ stands for the send buffer length at the end of the nth encoding instance. Two vertical lines represent the start and end of an encoding process, respectively.

Bibliography

[1] AccessGrid. http://www.accessgrid.org/. 66

[2] ANSI Video Quality Research. http://www.its.bldrdoc.gov/n3/video/tutorial/. 81

[3] AVATS. http://www.cs.ucl.ac.uk/research/avats/. 66

[4] Dummynet: *traffic shaper, bandwidth manager and delay emulator* . http://info.iet.unipi.it/~luigi/dummynet/. 78

[5] Heterogeneous Experimental Network, *UCL Computer Science*. http://hen.cs.ucl.ac.uk/. 77

[6] INRIA Videoconferencing System (*ivs*). http://planete.inria.fr/ivs/. 65

[7] Iperf: *TCP and UDP bandwidth performance measurement tool*. http://iperf.sourceforge.net/. 77

[8] Skype – *free Internet calls*. http://www.skype.com/. 21

[9] The Multicast Backbone. http://www-mice.cs.ucl.ac.uk/multimedia/projects/mice/mbone_review.html. 65

[10] Theora (Thusnelda and Ptalarbvorm) Demo. http://people.xiph.org/~xiphmont/demo/. 82

[11] Ultragrid. http://ultragrid.east.isi.edu/. 65

[12] Video Conferencing Tool – *vic* Tool. http://ee.lbl.gov/vic/. 22, 63, 65, 77

[13] YUV Video Sequences. http://trace.eas.asu.edu/yuv/. 80

[14] Video Codec for Audiovisual Services at $p*64$ kbits. *ITU-T Recommendation, H.261*, March 1993. 83

[15] Methods for subjective determination of transmission quality. *ITU-T P.800*, June 1998. 15, 81

[16] MPEG-4 Part 10, Advanced Video Coding. *ISO/IEC 14496-10*, May 2003. 83

[17] Subjective video quality assessment methods for multimedia applications. *ITU-T P.910*, April 2008. 81

[18] Methodology for the subjective assessment of the quality of television pictures. *ITU-R BT.500-12*, September 2009. 81

[19] Advanced video coding for generic audiovisual services. *ITU-T Recommendation, H.264*, March 2010. 83

[20] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *SIGCOMM*, pages 263–274, 1999. 26

[21] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *IETF RFC 2581*, April 1999. 26

[22] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *IEEE Infocom 2001*, pages 631–640, Anchorage, AK, April 2001. 27, 61

[23] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. In *SIGCOMM*, pages 263–274, 2001. 20, 31

[24] B. Briscoe. Flow Rate Fairness: Dismantling a Religion. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 65–74, April 2007. 50

[25] S.-H. Choi and M. Handley. Designing TCP-Friendly Window-based Congestion Control for Real-time Multimedia Applications. In *Proc. of the 7th PFLDNeT*, Tokyo, Japan, May 2009. 20, 31, 35

[26] L. D. Cicco, S. Mascolo, and V. Palmisano. A mathematical model of the Skype VoIP congestion control algorithm. In *CDC*, pages 1410–1415, 2008. 21

[27] L. D. Cicco, S. Mascolo, and V. Palmisano. Skype video responsiveness to bandwidth variations. In *NOSSDAV*, pages 81–86, 2008. 21

[28] F. P. Kelly. Charging and rate control for elastic traffic. `http://www.statslab.cam.ac.uk/~frank/elastic.html`. 50

[29] S. Floyd. *RED: Discussions of Setting Parameters*. `http://www.icir.org/floyd/REDparameters.txt`. 52

[30] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *Proc. of ACM SIGCOMM '00*, 2000. 19, 29

[31] S. Floyd, M. Handley, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. *IETF RFC 5348*, September 2008. 19, 36

[32] S. Floyd and V. Jacobson. Random Early Detection Gateway for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, pages 397–413, 1992. 52, 53

[33] R. Frederick. Network Video (nv). Xerox Palo Alto Research Center. 65

[34] T. Friedman, R. Caceres, and A. Clark. RTP Control Protocol Extended Reports (RTCP XR). *IETF RFC 3611*, November 2003. 15, 66, 68, 69

[35] B. Girod. What's wrong with mean-squared error? *Digital images and human vision, MIT Press*, pages 207–220, 1993. 82

[36] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. *IETF RFC 3448*, January 2003. 19, 29, 30

[37] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. *IETF RFC 2861*, June 2000. 86

[38] T. Hidakaa and K. Ozawaa. Report on MPEG-2 subjective assessment. *ISO/IEC JTC1 SC29/WG11*, 5(1-2):127–157, February 1993. 81

[39] V. Jacobson and M. J. Karels. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, 1988. 25

[40] I. Johansson and M. Westerlund. Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences. *IETF RFC 3611*, April 2009. 69

[41] G. Jourjon, E. Lochin, and P. Sénac. Towards sender-based tfrc. In *ICC*, pages 1588–1593, 2007. 31

[42] J. Klaue, B. Rathke, and A. Wolisz. EvalVid - A Framework for Video Transmission and Quality Evaluation. In *Computer Performance Evaluation / TOOLS*, pages 255–272, 2003. 78

[43] A. Klein and J. Klaue. Performance Study of a Video Application over Multi Hop Wireless Networks with Statistic-Based Routing. In *Networking*, pages 955–966, 2009. 21

[44] E. Kohler, , S. Floyd, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). *IETF RFC 4342*, March 2006. 29

[45] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). *IETF RFC 4340*, March 2006. 37

[46] R. Kuschnig, I. Kofler, and H. Hellwagner. An Evaluation of TCP-based Rate-Control Algorithms for Adaptive Internet Streaming of H.264/SVC. In *Proc. of MMSys*, February 2010. 21

[47] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. *IETF RFC 2018*, April 1999. 26

[48] Matt Mathis et al. Heresy following "TCP: Train-wreck". http://oakham.cs.ucl.ac.uk/pipermail/iccrg/2008-April/000488.html. 50

[49] S. McCanne and V. Jacobson. *vic*: A flexible framework for packet video. In *ACM Multimedia*, pages 511–522, 1995. 65, 83

[50] J. D. McCarthy, M. A. Sasse, and D. Miras. Sharp or smooth?: comparing the effects of quantization vs. frame rate for streamed video. In *CHI*, pages 535–542, 2004. 99

[51] Network Simulator, Version 2.34. http://www.isi.edu/nsnam/ns/. 42, 47

[52] 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. 30, 35

[53] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. *IETF RFC 2988*, November 2000. 36

[54] J. Postel. Transmission Control Protocol. *IETF RFC 793*, September 1981. 26

[55] G. Renker and G. Fairhurst. Sender RTT Estimate Option for DCCP. *IETF Internet Draft draft-renker-dccp-tfrc-rtt-option-01*, August 2010. 31

[56] I. Rhee and L. Xu. Limitations of Equation-based Congestion Control. *IEEE/ACM Transactions on Computer Networking*, 15(4):852–865, August 2007. 20, 31, 61

[57] A. Saurin. Congestion Control for Video-conferencing Applications. *MSc Thesis, University of Glasgow*, December 2006. 20, 31

[58] H. Schulzrinne. Voice Communication Across the Internet. *Technical Report RT 92-50, Department of Computer Science, U. of Mass, Amherst*, July 1992. 65

[59] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *IETF RFC 1889*, January 1996. 69

[60] D. Sisalem and A. Wolisz. Towards TCP-Friendly Adaptive Multimedia Applications Based on RTP. In *ISCC*, pages 166–172, 1999. 21

[61] M. Tallat, M. Koutb, and H. Sorour. PSNR Evaluation of Media Traffic over TFRC. In *International Journal of Computer Networks and Communications*, volume 1, 2009. 21

[62] M. Vojnovic and J. L. Boudec. On the long run behavior of equation-based rate control. In *Proc. of ACM SIGCOMM 2002*, pages 103–116, 2002. 20, 31, 59, 61

[63] Z. Wang, S. Banerjee, and S. Jamin. Media-friendliness of a slowly-responsive congestion control protocol. In *NOSSDAV*, pages 82–87, 2004. 21

[64] Z. Wang and A. C. Bovik. Mean Squared Error: Love It or Leave It? *IEEE Signal Process Magazine*, pages 98–117, 2009. 82

[65] D. Wu, Y. T. Hou, W. Zhu, H.-J. Lee, T.-H. Chiang, Y.-Q. Zhang, and H. J. Chao. On end-to-end architecture for transporting MPEG-4 video over the Internet. *IEEE Trans. Circuits Syst. Video Techn.*, 10(6):923–941, 2000. 81

[66] J. Yan, K. Katrinis, M. May, and B. Plattner. Media- and TCP-friendly congestion control for scalable video streams. *IEEE Transactions on Multimedia*, 8(2):196–206, 2006. 21