UNIVERSITY
of
GLASGOW

Department of Computing Science

University of Glasgow

Congestion Control for Video-conferencing Applications

MSc by Research

Alvaro Saurin

December 15, 2006

Abstract

In the Internet, transmission systems must deal with congestion in order to keep the stability of the network. However, the model used for congestion control determines some important properties of the traffic. The most important algorithm currently used, found in the TCP protocol, has characteristics that make it unsuitable for videoconferencing systems.

The aim of this dissertation is to provide an insight into the field of congestion control for such systems. In particular, this work examines one of the most promising alternatives available, *TCP-Friendly Rate Control* (*TFRC*), to answer the question *"is TFRC suitable for interactive videoconferencing applications?"*

This dissertation presents the results obtained with TFRC, focusing on some practical aspects and providing recommendations for the implementation of such a *rate-based* congestion control system. This work examines the scenarios where TFRC is an adequate solution, exposing the behavior that can be expected and suggesting future improvements.

The thesis also presents the experiences of integrating TFRC in the *UltraGrid* videoconferencing application. It shows the difficulties found, demonstrating that this integration requires an significant amount of support from the application, and questioning the suitability of TFRC in some situations.

Acknowledgments

First and foremost, I would like to thank Colin Perkins for his expert guidance throughout this project, and for his patient, friendly, and unfailing support over the last year. Many thanks to Ladan Gharai for her keen insight and thoughtful reflection on this work. Many thanks also to Peter Dickman for his assistance and help.

Special thanks to Raquel for understanding my work and for her constant support and encouragement even at the most difficult times. Finally, I would like to thank my parents and family for their assistance and for sponsoring part of my work over the last few months.

This work was supported by the US National Science Foundation under grant No. 0230738.

Contents

Chapter 1

Introduction

The aim of this project was the implementation of a congestion control mechanism and its integration in an existing videoconferencing application, *UltraGrid*.

This thesis covers previous work in this area, the approach taken by this project to tackle the problem, and the design, implementation, testing and evaluation techniques used during the course of the project.

1.1 Motivation

The rapid deployment of broadband technologies in our homes, via cable modem or *Digital Subscriber Line*, *DSL*, has resulted in a significant increase in the use of multimedia applications and, in particular, videoconferencing and streaming applications. In next years, users will change their cable/DSL for optical connections, allowing even higher transmission speeds. This will probably lead to new uses of the network for video streaming, *video-on-demand*, videoconferencing and a wide range of new applications. The need for bandwidth by these systems will probably make them important players in the future Internet. In this new context, the development of a robust and reliable framework for the transmission of media content is extremely important.

Although some multimedia applications work acceptably with current Internet technologies, we can not say the same for *interactive* real-time transmission, where timing constraints acts as an additional handicap. Although users can tolerate some loss and quality degradation, they will notice any rate change, choppiness or late reception of frames. These goals are just the opposite to what the *Transmission Control Protocol*, *TCP*, the most important transport protocol, provides. Timing is not the main objective of TCP, which is characterized by strong throughput variations produced by the congestion control algorithm. The solution used by some application is simple: to use a different transport protocol.

When applications avoid the use of TCP, they are also avoiding the use of the congestion control that TCP provides. In order to evade the danger of a congestion collapse, their sending rate should be guided by a congestion control algorithm, and this is not a trivial task. Accordingly, the implementation of a congestion control system has been seen as a secondary objective by software developers, and an increasing number of applications are skipping this step.

This is a concern: the lack of congestion control could lead to congestion collapse, but these applications need ready-to-use solutions that can be easily adopted. As we will see in the following chapters there is a need for effortless, media-friendly and TCP-friendly congestion control frameworks but, although the number of algorithms available in the literature is more than enough, it seems that there is a shortage in real implementations and real experiences.

The aim of this project is therefore to contribute to the development of new congestion control mechanisms for interactive multimedia applications, focusing on the real utilization of these systems and their relation with the applications.

1.2 Thesis Statement

This thesis focuses on one of the most promising *TCP-Friendly* congestion control algorithms: *TCP-Friendly Rate Control*, *TFRC*. It seeks to answer the question *"is TFRC suitable for interactive videoconferencing applications?"*.

I will show that, although it has some good characteristics for a videoconferencing application, like a smooth throughput variation, TFRC presents some practical issues in real environments and systems. I show the details of this congestion control system from an implementation point of view, performing analytical and empirical studies of the protocol and showing some problems of TFRC in the real world. I also study the issues that arise when it is used in a videoconferencing application, showing the dynamics that it imposes in the system and the limitations of the algorithm in such an environment.

1.3 Work Outline

This dissertation starts with a detailed literature survey in Chapter 2, covering the topics that surround this subject, analyzing the scenario of interactive multimedia transmission and providing an overview of some videoconferencing systems. The rest of the thesis is then divided in two main parts: the *TCP-Friendly Rate Control*, *TFRC*, and *UltraGrid*.

The first part begins with the fundamentals of TFRC in Chapter 3. This chapter presents an in-depth description of the protocols, showing all the different components of the algorithm and explaining all the mechanisms that guide this congestion control system. The discussion continues in Chapter 4, where I show the main aspects of any TFRC implementation, focusing on the most problematic issues and risks, and presenting some of the difficulties and design decisions that have been made. The results obtained with TFRC are presented in Chapter 5, showing the behavior of our TFRC implementation in a broad variety of scenarios, comparing it with the expected results and highlighting some conclusions.

The second part of this thesis is focused on *UltraGrid*, the videoconferencing application where TFRC has been integrated. Chapter 6 introduces the application, providing an overview of a videoconferencing system and the details of the TFRC integration. Results obtained using UltraGrid with TFRC are presented in Chapter 7, where we discuss some problems of TFRC as a congestion control for interactive media traffic.

The dissertation concludes in Chapter 8 with a summary of the content presented in the thesis, conclusions and suggestions for potential future work in this area.

Chapter 2

Background and Related Work

This chapter introduces the real-time interactive multimedia problem. It will focus on the traffic characteristics, the congestion control systems and the relation of these elements with the application level, analyzing some of the most important alternatives and providing a deep overview of the related work.

This chapter is divided in two main parts, as the information available on videoconferencing applications can also be classified in two broad groups. The first part is centered around transmission and congestion control. It starts in Section 2.1 with some background on the most important transmission protocols, followed by a discussion on congestion control and the TCP solution (in Section 2.2). The discourse continues with an analysis of multimedia traffic characteristics (Section 2.3), analyzing the suitability of TCP for this scenario (in Section 2.4) and introducing the family of TCP-Friendly solutions in Section 2.5.

The second part of the chapter (Section 2.6) is focused on real-time media systems and, in particular, videoconferencing applications. Section 2.6.1 starts with an overview of the current technologies used in codecs, their design principles and how they interact with applications, while Section 2.6.2 examines some of these applications, listing the characteristics of the most important systems that have appeared in the last years. The chapter finishes in Section 2.6.3 with an study of how congestion control is currently used in the real-time multimedia scenario.

2.1 Transport Protocols for Real-Time Multimedia

The transmission of media traffic involves several layers of the protocol stack. Each of these layers contributes in a different way to the final characteristics of the resulting traffic. Figure 2.1 shows the stack of protocols that can be used in a real-time multimedia application.

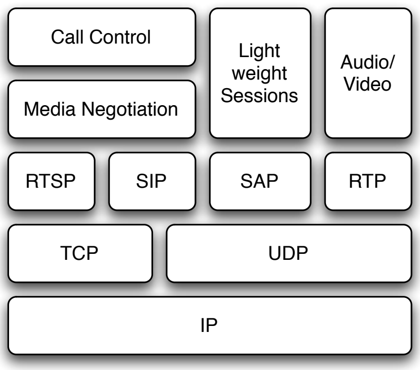


Figure 2.1: Media protocol stack.

The basis of the protocol stack is the *Internet Protocol*, *IP* [77]. IP forms the basis of the current Internet, and it provides the basic mechanisms that govern the network, like routing or fragmentation. However, IP is not directly used by applications in most cases, and higher level protocols like TCP or UDP are used instead.

The *Transmission Control Protocol*, *TCP*, was described in [78] as a protocol for the reliable and ordered delivery of packets. TCP brings a higher abstraction level for applications, acting as a transport layer and multiplexing different connections between hosts. The protocol includes a loss detection and retransmission system, providing a guaranteed delivery of packets to applications at an additional cost of longer delays and *delay jitter*.

One of the most important characteristics of TCP is the congestion control and avoidance mechanisms. This feature, that we will see in detail in Section 2.2.1, has made possible the Internet as we know it now by enabling the fair use of the network resources and avoiding congestion collapse. However, TCP is not the most frequently used transport protocol for multimedia content. A connectionless and unreliable protocol, UDP, is the preferred one instead.

The *User Datagram Protocol*, *UDP* [76], is a best-effort transport protocol, designed as a message-oriented service for the timely and efficient transmission of datagrams without the reliability of TCP. UDP offers a simple service where the sender does not keep any state information once a message has been sent. It provides unreliable delivery of packets, without any flow control or error recovery. Applications that use UDP, such as videoconferencing applications, media players or mutiuser games, typically prefer to send data within a known period of time in a context where retransmission is not necessary and where timely response is desired. Due to this timing requirement, UDP is a more suitable protocol for real-time media communication.

From the point of view of a multimedia application, TCP implements a non-essential retransmission system that can disrupt application timing. In contrast with UDP, TCP provides a congestion control mechanism that should be used by every application in order to keep the stability of the Internet. In Section 2.5.3 we will see some examples that try to provide the best of both worlds.

Other protocols can also be used for the transport of multimedia content. Nowadays, the preferred system is the *Real-time Transport Protocol*, *RTP* [87]. RTP is a media oriented protocol that provides timing recovery, media framing and loss detection. RTP provides a mechanism for the delivery of real-time media content using an unreliable transport layer, and it is generally used with the best-effort delivery given by UDP.

The RTP framework comprises several elements. The *Data Transfer Protocol* provides payload identification, media sequencing and time recovery, and it is responsible for the transport of application data units, either audio or video, using separate streams for each media type. The *Control Protocol* performs two basic functions: first, it carries reception quality feedback, informing the sender of details like the packet loss rate or the timing jitter; second, it is used for lip synchronization by mapping the media clock to an external time reference. The mapping between codecs and RTP is given by the *Payload Formats*, and there are several standards for popular codecs like the *Moving Pictures Expert Group* codecs, MPEG-1/2 [43] and MPEG-4 [50], the *Digital Video* codec, DV [51], or JPEG [9]. An RTP *Profile* provides a namespace for these payload formats, as well as other defaults for particular scenarios.

RTP is only used for carrying multimedia data, and other protocols must be used for call setup, session control or tear-down. For example, RTP is currently used in streaming systems with protocols like the *Real-Time Streaming Protocol*, *RTSP* [88]. This protocol implements a set of commands for this kind of systems (ie, *play*, *pause*, etc.), using simple requests like in the *Hypertext Transport Protocol*, *HTTP*.

Other domains where RTP is used include videoconferencing and push to talk systems. In these environments, session negotiation and call setup is usually performed using protocols like the *Session Initiation Protocol*, *SIP* [85], or the *Session Announcement Protocol*, *SAP* [40]. SAP is mostly used for public sessions, specially in multicast environments, while SIP provides a superset of the features found in public switched telephone networks (ie, calling a number, hearing ringback tones, etc.), but based on a simple peer-to-peer architecture. Even although this simplicity was the most important characteristic of SIP and the main difference when compared to *H.323* [67], the protocol has evolved and has become more complex framework [53].

2.2 Congestion and Congestion Control

When the transmission resources of a network reach their limits, the network suffers congestion. Congestion is generally produced at the buffers that routers use for queuing packets. When the input rate is higher than the output rate, buffers become full and no more packets can be processed. The router is then congested, and the most common solution is to simply drop those packets that cannot be handled. When this happens, transmission endpoints must detect the situation and reduce their sending rate in order to alleviate the problem. Afterwards, they can retransmit packets if they consider it necessary.

Although congestion situations are something normal in a network, there must be some system that controls the traffic and reduces the sending rate when congestion occurs. During the 1980's, the need of such mechanism was evident when the Internet suffered some serious congestion situations [48]. The lack of an effective control produced several *congestion collapses*: livelocks where, while routers were dropping packets, senders were keeping their transmission rates constant and intensified the collapse by retransmitting lost packets.

Congestion Control systems are mechanisms used in computer networks for avoiding congestion collapse. Congestion control systems should follow basic rules in order to keep a stable and sustainable network operation. The basic principles that any congestion control mechanism must follow have been defined as fairness, robustness, scalability and feasibility [56].

Assuming that the congestion control is a function linked to the limited amount of resources (and, in particular, to the bandwidth), we can study how these resources can be controlled and, more specifically, where to implement this management. There are two main approaches to this problem: the implementation of the management function at the routers, using some mechanisms that isolate flows from the rest of the traffic, or the shift of this functionality to the traffic endpoints.

An example of the first model is the *Differentiated Services* architecture, or *DiffServ* [10]. DiffServ is a widely deployed framework that tries to guarantee the quality of service for bulk flows of data. DiffServ is based in the idea of groups of routers that form a *cloud*, all of them obeying the same policy. Packets entering the cloud will be classified and they will be treated by all the routers of the cloud depending on the priority assigned.

Another alternative that follows the same model is the *Integrated Services*, *IntServ* [12]. This approach tries to provide some *Quality of Service*, *QoS*, guarantees for each flow by using a special resource allocation protocol, the *Resource Reservation Protocol*, *RSVP* [101] [13]. Then some parameters can be set on a per-flow basis where an assured bandwidth, loss rate or delay can be established for a single connection.

In this model, there is no congestion control at the network level, but they provide something that can make it unnecessary. This approach gives the ability of setting some parameters in advance, like the minimum bandwidth or the maximum packet loss that a flow will see. For multimedia traffic, this knowledge can be extremely valuable and, as we will see in Section 2.3, it can lead to a constant frame rate or buffering reduction in the application.

Even when this model can be feasible in a controlled environment, it presents a deployment problem when we look at the Internet as a whole. The heterogeneous and decentralized nature of the Internet make this approach quite difficult to use as a global management solution. It is difficult to predict what kind of congestion control we can expect of a connection traversing several networks, making rather hard to maintain a consistent control of traffic that travels through different continents. In this context, perhaps it would be better to lighten the network control and use an end-to-end approach for the congestion control.

Using the end-to-end model, we can provide an affordable flow control and avoid a congestion collapse problem. The role played by routers is reduced (although they can *incentivise* flows with better congestion control [29]) and, in consequence, we can not enforce some parameters that only a router-based solution can provide, such as bandwidth reservation.

Systems following the end-to-end congestion control model can be divided in two main groups: *window-based* and *rate-based* congestion control mechanisms.

- In window-based congestion control, a window is used to limit the amount of data that the sender can send. The window is then increased at a particular rate when there are no losses, or decreased or reset when losses are detected. Window-based congestion control systems provide good adaptation for traffic changes, but this responsiveness results in frequent variations of the sending rate. For application that look for smoother changes in the sending rate, rate-based congestion control systems are more suitable.
- In contrast, applications using rate-based congestion control send packets at a periodically calculated sending rate, depending on the current network parameters. However, this smoothness results in a less aggressive search for new bandwidth available than window-based solutions.

In the following section, I will focus on the most important window-based congestion control algorithm: the TCP congestion control system. Then, I will present some alternatives in Section 2.5.2, showing a mixture of window-based and rate-based schemes.

2.2.1 Congestion Control in TCP

The TCP congestion control has evolved since it was originally defined in [48]. Although this system was been improved with the inclusion of some extensions like the *Explicit Congestion Notification*, *ECN* [80], the *Selective Acknowledgment Option*, *SACK* [61] (and *D-SACK* [28]) or *Forward Acknowledgments* [60], some of these extensions are still not widely deployed and they add a complexity that is far beyond this work. Instead, I will focus on some basic characteristics of the congestion control mechanism found on a standard *TCP* implementation and their effects on the transmission of multimedia content.

The TCP congestion control algorithm provides an essential mechanism for the stability of the network, detecting congestion situations and controlling the transmission rate at the ends when this happens. The algorithm is based on two basic principles. First, it uses a *congestion window* as an estimate of how much data can be outstanding in the network without packets being lost. Second, it assumes that a loss is a sign of congestion, and it requires that, when the sender detects this loss, it must reduce the transmission rate.

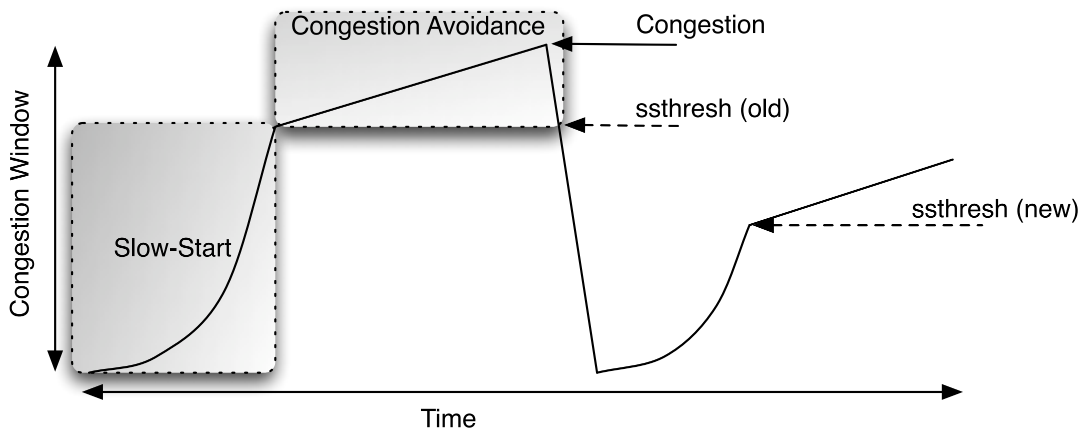


Figure 2.2: Stages in the TCP congestion control.

In consequence, an important part of the congestion control functionality depends on the loss detection system. The mechanism used for detecting losses is based on *acknowledgments*, *ACK*, sent by the receiver on the reception of every segment. Using these *ACK*s, the sender can detect losses in two different ways:

- First, the sender can use a retransmission timer. If no new data is acknowledged for some time, the timer will expire and the data can be considered as lost. This waiting time is specified in the *retransmission timeout*, *RTO*, and it is indirectly calculated from the mean *Round-Trip Time*, *RTT* [71].

- Second, since the TCP receiver acknowledges the highest consecutive segment number received, the sender can also detect losses when duplicated acknowledgments arrive. This retransmission, triggered by three successive duplicate *ACK*s, is known as the *fast retransmit*. After the fast retransmit, the sender follows the *fast recovery* algorithm until the receiver acknowledges all the segments in the last window. During this phase, the sender transmits a new segment for every *ACK* received, and the congestion window is increased for each duplicate acknowledgment.

The congestion control algorithm uses the idea of a *congestion window* as the main control mechanism for the data in transit. This window is represented by *cwnd*, the number of segments that the sender is allowed to send at one time, and its behavior is controlled in two different phases: the *slow-start* and the *Additive-Increase Multiplicative-Decrease*, *AIMD*, congestion-avoidance phases. Figure 2.2 depicts these stages and some basic functions of the congestion control in TCP.

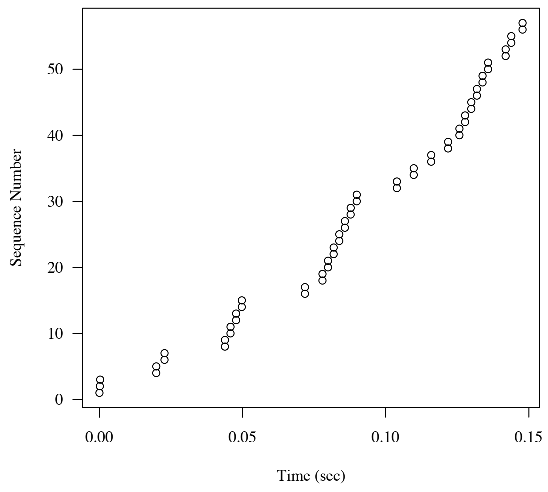


Figure 2.3: Packets in Slow-Start.

In the slow-start phase, the sender tries to fill up the link by exponentially increasing the congestion window, sending more and more data. The *cwnd* is initialized to 1 or more [3] and is increased in one packet for every *ACK* received. The slow-start phase continues until the first loss is detected or the congestion window reaches a predefined threshold, *ssthresh*.

Figure 2.3 shows an example of the slow-start phase in TCP. In this figure we can see the sequence numbers of the packets sent during the slow-start phase. We can observe how the number of packets per unit of time changes noticeably during this short period of time, as the aim of this stage is to fill up the link and to try to find the bandwidth of the bottleneck.

However, there is an upper limit for the congestion window. In order to avoid losses due to insufficient memory at the receiver, the receiver announces to the sender how much free space is available. The sender will keep a copy of this information and use the minimum between this value and the current congestion window for the next number of segments sent. We can see this effect in Figure 2.4, where we can see how the window size can changes with time, but it never goes beyond this limit set by the receiver.

When the first loss is detected, the congestion window is reduced by half. The *ssthresh* is then updated accordingly, and the sender can restart the slow-start but with a lower threshold. However, if there is no loss and *cwnd* reaches *ssthresh*, the sender enters the

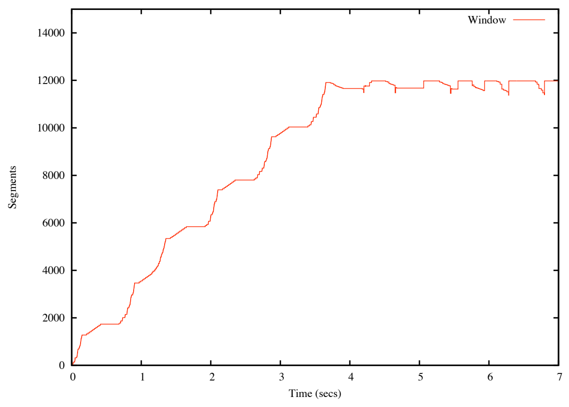


Figure 2.4: TCP window for a connection between two Linux boxes with *dummynet* ($20ms\ RTT$, $3Mbit/s$ bandwidth, $7,5Kb$ buffering).

congestion avoidance phase. In the congestion avoidance stage, the congestion window is additively increased at a rate of one segment in a round-trip time, and halved when a loss is detected.

In general, we can describe a whole family of algorithms that follow this mechanism with a formula. We can characterize an AIMD-based algorithm by two parameters, a and b, corresponding to the increase and decrease parameters of the algorithm [7]. For a AIMD(a, b) algorithm, the congestion window is reduced to $(1 - b)cwnd$ after a loss, or it is increased from $cwnd$ to $cwnd + a\ pcks/RTT$ when no loss is detected. When TCP does not use delayed acknowledgements, it can be considered as a *AIMD(1, 0.5)* [25]. We will consider alternative AIMD algorithms in Section 2.5.2.

2.3 Characteristics of Real-Time Multimedia Traffic

In order to better understand the relation between the congestion control algorithm and its effects at the application level, we must first understand the characteristics of the traffic created by multimedia applications. Real-time multimedia traffic is affected by three network parameters:

1. Bandwidth

   Multimedia applications have different requirements regarding the traffic they generate. In particular, if we consider the amount of data necessary for a videoconference session we will realize that we will need a considerable sum of bandwidth.

Codecs can balance this need for bandwidth by increasing the compression level but, even for the last generation of codecs, the minimum level required for an acceptable quality can produce a rather fat stream of bytes that must be transmitted through the net.

However, multimedia traffic is more affected by the bandwidth variation. As we discuss in Section 2.4, rapid changes in bandwidth can significantly affect media quality.

2. Delay jitter

The end-to-end delay that a packet experiences may not be the same for different packets. Queuing in intermediate buffers or routing instabilities can be some of the causes of this effect. This *delay jitter* can be a problem for applications where timeliness is more important than reliability. In videoconferencing applications, for example, the timely presentation of frames results in a more natural and pleasant display [16], and it depends on the reception of frames at the right moment.

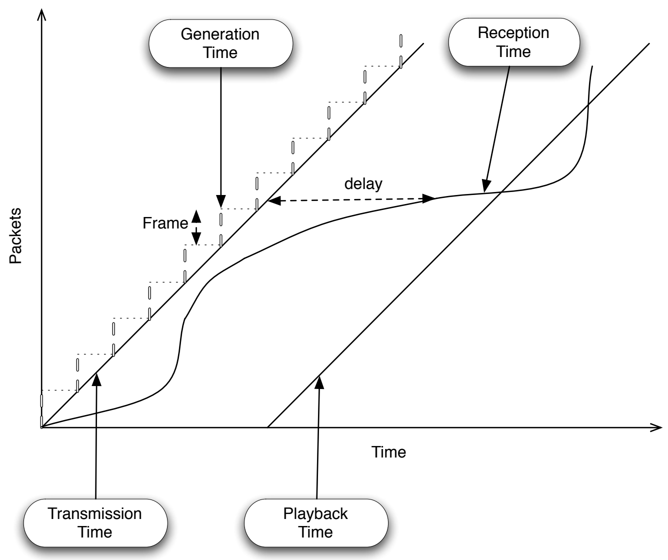


Figure 2.5: Delays in media transmission.

We can see in Figure 2.5 how this *jitter* affects the transmission of media traffic. For each media element (*i.e.*, a frame), packets are generated (at logically the same time), and sent sequentially to the receiver. However, the network delay is not constant, and the reception time for packets changes. The receiver then needs to store plackets in a playout buffer, giving some time to gather all the packets of the same frame together before the playback time of the frame is due.

3. Packet Loss

   Internet flows suffer some degree of packets lost. This can be caused by failures in the transmission systems, but the main source of losses is the presence of congestion in the network resources. In general, intermediate routers have a limited amount of buffering space, and they drop packets when these buffers are full.

   For media traffic, a reasonable amount of losses is not an important problem. A lost packet is something insignificant most of the time, because it can represent a tiny fragment of audio or video. On the other hand, the arrival of these packets in the right moment is important and, most of the time, the retransmission of packets is considered a waste of time and resources if they wouldn't arrive in time to be useful.

The way a congestion control protocol deals with these factors determines the final result obtained at the application level. In the next section we will discuss how TCP deals with these variables, and why it is not an appropriate protocol for interactive multimedia traffic.

2.4 TCP for multimedia traffic

As we have seen in Section 2.2, TCP controls the amount of data sent using a congestion window. When loss is detected, the window is reduced by half, resulting in a drastic change in the sending rate and forcing the sender to reduce the amount of data injected into the network.

The strong throughput changes start with the slow-start phase, when the congestion window is doubled every RTT. After the finalization of this phase, the smoothness of the congestion control system does not improve, and the window size variation results in drastic changes in the throughput of the sender.

The behavior in steady-state can be observed in Figure 2.6, where we can see how the throughput shows frequent rapid changes. It corresponds to a TCP connection in a environment with *20ms* RTT and *3000Kbit/s* of bandwidth[1]. The dynamics of the congestion control mechanism result in drastic changes in the throughput. This variation

[1]This connection corresponds to the Scenario II in Table 5.2, that we will see in Section 5.2.1 (page 69)

forces quick changes in the amount of data that the media application must produce. If the media application produces more data than it is allowed to transmit, this data must be dropped. The sender must try to match the sending rate of TCP by changing the compression applied to the input media. However, due to the high frequency of the variation in the TCP throughput, an output rate that strictly follows the sending rate results extremely variable in quality and, in consequence, is unpleasant for the users [16] [23].

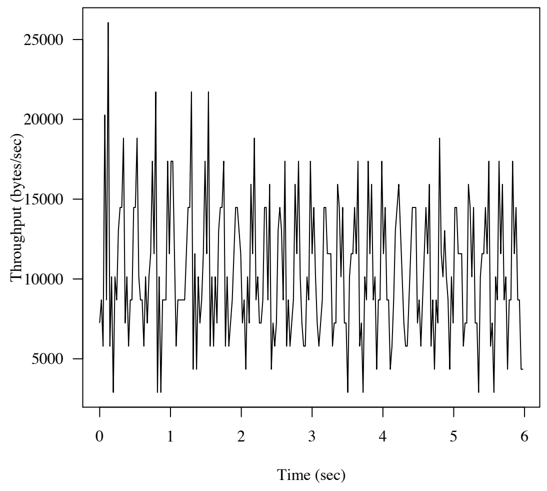


Figure 2.6: TCP throughput in steady-state

The sender can reduce the effects of these rapid changes by using buffering to mitigate the short-term variations and produce data following long-term trend, adapting the output rate more evenly. However, we can not forget that, as we have seen in Section 2.3, the receiver also needs some buffering. Media elements (*i.e.*, frames, audio samples, *etc.*) are not immediately available, and the variability of the TCP throughput added to the delay jitter results in a significant fluctuation of the data that arrives at the receiver. A playout buffer is needed in order to neutralize these effects and provide a timely output.

In consequence, all this system buffering has a side effect: it increases the end-to-end delay and reduces the interactivity of the application. This makes TCP an unacceptable solution for interactive real-time media applications, but it can be feasible in other environments. TCP has been identified as a suitable protocol for media transmission when applications allow some timing flexibility [96], in particular for streaming, where a couple of seconds of buffering is insignificant.

Another problem of TCP for multimedia traffic is the reliability offered by the protocol. It uses a transmission model where data is assured to reach the other end of the connection. While this model is satisfactory for other applications, it is not convenient for multimedia applications with strong timing limits. In this case, the retransmission of data is useless if it will arrive at the receiver after the time when it was supposed to be used, and it consumes resources that could be used for the transmission of more useful content.

Despite all these problems, TCP is a widely used protocol for media transport. We will see in Section 2.6.3 why TCP is one of the most popular transport protocols in the multimedia scene, and how it is used in these environments. Firstly, we will see other alternatives to TCP, how they can be designed in order to coexist with TCP and the benefits they provide.

2.5 TCP-Friendly Congestion Control

In the following sections, we will study other congestion control algorithms available. All of them satisfy a basic requirement: they are *TCP-Friendly*.

2.5.1 TCP-Friendliness

It has been observed that the average throughput of a TCP connection can be inferred from some end-to-end measurements, like the round-trip time and the loss event rate [62]. In fact, the long term throughput of TCP can be calculated using the equation given in [30], a simplified version of the TCP response function shown in [69]. This equation can be expressed as Equation 2.1:

$$X = \frac{s}{RTT\sqrt{\frac{2p}{3}} + T_{RTO}(3\sqrt{\frac{3p}{8}})p(1+32p^2)} \tag{2.1}$$

We can see that this function depends on the values taken by four parameters. The loss event rate, p, is a measure of the lost packets. Other parameters are the mean *round-trip time*, RTT, and the packet size, s. The T_{RTO} is the timeout value in seconds, simplified in [39] as $4 * RTT$.

However, it must be noticed that, even when we can have accurate values for these parameters, it is difficult to determine the real throughput of a TCP connection. There is no single standard behavior of TCP in the real world and, in consequence, there are different implementations that react in different ways to network changes, producing different throughputs. Nevertheless, if we use Equation 2.1 as a reference for the throughput of a TCP connection in the long term, we can define a new fundamental property of any flow: the *TCP-Friendliness.*

We can consider that a flow is *TCP-Friendly* (or *TCP-Compatible*) when the *"long term throughput does not exceed the throughput of a conformant TCP connection under the same condition"* [29] [68]. This is an attribute that every flow sharing the net should have: TCP-Friendly flows interact well with other TCP traffic and maintain the stability of the Internet [11]. The lack of TCP-Friendliness not only leads to the unfair share of the bandwidth available, but it could also result in a congestion collapse [29].

However, this definition of friendliness, based on the long-term throughput, forgets the aggressiveness that a connection can have. Even when a TCP-Friendly flow can have a long-term throughput equivalent to TCP, the dynamics of the flow (in particular, the aggressiveness when it increases the throughput) can disturb other TCP flows. This is the reason why other authors [98] use an alternative definition for TCP-Friendliness, where a flow is considered TCP-Friendly when *"it does not reduce the long-term throughput of any co-existent TCP flow more than another TCP flow on the same path would do under the same network conditions".* In the sake of simplicity, we will assume that both conditions are equivalent.

In general, we can establish that a congestion control mechanism is TCP-Friendly if the behavior of its flows are TCP-Friendly. Provided that these conditions are met, any TCP-Friendly congestion control mechanism has the ability to generate traffic that coexists with other TCP traffic in the same network and, at the same time, their flows will be fair to TCP flows on average, with no greater long-term throughput under the same loss and delay conditions.

Keeping in mind all these constraints, new congestion control mechanisms can be designed in order to be TCP-Friendly and, at the same time, achieve different goals. As long as the flows keep their TCP-friendliness, a congestion control system could change the sending rate following a different set of rules. These alternative regulations could be designed with other objectives in mind. For example, some congestion control systems could look for sending rate smoothness, while other algorithms could prioritize the maximization of throughput.

2.5.2 TCP-Friendly Solutions

As we have seen, the idea of TCP-Friendliness brings a new family of congestion control algorithms. I will focus on congestion control systems for unicast applications, as the schemes used for multicast are usually implemented in a different way and with algorithms that are generally more complex: these mechanisms must scale to a large number of receivers and must deal with heterogeneous network conditions.

In Section 2.2 we saw that congestion control algorithms can be classified in two main categories:

- The first group is composed by systems that use the same behavior as the *Additive-Increase Multiplicative-Decrease*, *AIMD*, in TCP but using different constants. An overview of some alternatives belonging to this group can be found in [49], with some remarkable examples like the AIMD algorithm of TCP, *TEAR* [82], *RAP* [81] or the *Binomial* [7] congestion control algorithm.
- The second group is formed by equation-based mechanisms that use a throughput formula that tries to be equivalent to the TCP throughput equation on a medium timescale. In this group we can find *TFRC* [30]).

We can see some examples of these congestion control algorithms in Figure 2.7. Algorithms are classified depending on several characteristics, like the responsiveness to congestion or the level of TCP-compatibility. It must be noticed that some algorithms are characterized by some parameters that modify their behavior. Depending on the values taken by these parameters, the algorithms can show a higher friendliness or a lower responsiveness [21]. For example, values of $b < 0.5$ for an *AIMD(a, b)* scheme correspond to slowly-responsive algorithms.

An example of an algorithm where parameters play an important role is the *Binomial* congestion control system [7]. This mechanism uses the current window as a factor for the increment or decrement of the sending rate. The Binomial congestion control system is a nonlinear generalization of AIMD, characterized by four parameters k, l, a and b. These parameters are present in a general formula that specifies the increment and decrement function for the current window. Depending on the particular values used for these parameters, the Binomial algorithm can adopt different forms. In fact, *AIMD(a,b)* congestion control can be seen as an special case of binomial congestion control, where constant l and k take values 1 and 0 respectively. However, other parameter combinations produce different binomial algorithms, reaching higher smoothness or reducing the

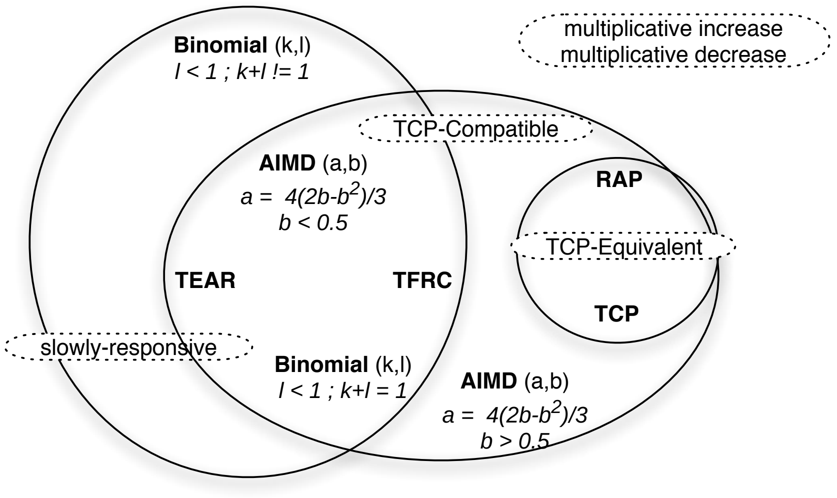


Figure 2.7: CC algorithms (adapted from [21])

TCP-Friendliness of the result.

Another AIMD mechanism is used by the *Rate Adaptation Protocol*, *RAP* [81]. In contrast with the Binomial algorithm, RAP is a simple congestion control algorithm that tries to mimic the TCP behavior but with a rate-based model. It is based on the delivery of feedback from the receiver, used for detecting losses and for the computation of the RTT. The sender updates the sending rate depending on the losses suffered in the last RTT, halving the rate when it detects congestion or increasing by one packet per RTT when it does not, and resulting in a throughput that is very similar to TCP.

In the *TCP Emulation At Receivers TEAR* [82], receivers calculate a fair reception rate using a congestion window similar to the one used in TCP. However, TEAR does not directly obtain the amount of data to send from this window, but calculates the equivalent TCP sending rate, resulting in a rate near to a congestion window per RTT. Once that the sending rate is obtained, it is sent back to the sender but, in order to avoid the abrupt changes of TCP, it is smoothed using a weighted average over a set of previous periods of time. Thanks to this behavior that emulates TCP in the short term, TEAR reaches a TCP-Friendly transmission rate but without the sawtooth-like shape of TCP (as we saw in Figure 2.6).

One of the most popular examples of an equation-based system is the *TCP-Friendly Rate Control*, *TFRC* [30] [39]. TFRC is a congestion control system for unicast applications where the sender changes the sending rate as a function of the loss event rate. TFRC calculates this rate using the TCP throughput equation, resulting in a TCP-Friendly output but with smoother changes over time. TFRC is appropriate for applications that prefer to keep a slowly-changing rate at the cost of less aggressiveness when looking for more bandwidth.

Some studies highlight that TFRC shows better characteristics for multimedia traffic than other TCP-Friendly solutions. It provides a smoother rate variation than the *AIMD(a,b)* family of algorithms, with a maximum increment in the sending rate of 0.14 *pcks/RTT* (or 0.28 *pcks/RTT* when history discounting is used) [30], keeping its friendliness [25] and with a fairness level even higher than TEAR [100].

Even though TFRC is considered among the algorithms with better *friendliness*, some authors have suspicions about the real throughput obtained with the protocol. Several studies show some throughput difference between TFRC and TCP [92] [93], focusing on the conservativeness of the protocol that could result in Equation 2.1 being an upper bound of the real throughput of the protocol. According to these studies, *"TFRC can be TCP-Friendly in the long run and in some case, excessively so"* [93]. However, a prevailing thought is that the average discrepancy between both protocols does not constitute a menace for the stability of the Internet.

Other studies [21] [93] [70] also show that a slowly responsive flow like TFRC can suffer a higher loss event rate than TCP. This can have a negative effect in the throughput difference between TCP and TFRC and result, in some extreme cases, in TFRC using up to twenty times more or ten times less bandwidth than a TCP flow in the same situation [83].

TFRC is a new congestion control algorithm, and there are just a few studies of the real behavior of the protocol on the Internet. TFRC has been used for the transmission of MPEG-4 [24] [94] or as a basis for a TCP-Friendly rate control algorithm for JPEG-2000 video streaming [32]. It has also shown some problems with wireless environments [38], where it has been observed to reach lower throughput than TCP [14].

It must be noticed that TFRC could also be used for multicast applications [99]. In this case, receivers could measure the *RTT* and loss rate, calculate the sending rate that should be used, and report it to the sender. The sender should then use the slowest sending rate reported as the base rate for the multicast transmission.

2.5.3 Other Frameworks

As we have seen, many researchers believe that, in order to keep the stability of the Internet, congestion control mechanisms must be brought to applications or a new congestion collapse situation could happen. New congestion control frameworks are needed, and they must be easy to integrate with the present applications.

Congestion control can be supplied in different forms. Usually, it is available at the kernel level, integrated with the transport protocol (like in TCP). This solution could provide congestion control in a transparent way or with minimum changes to the application. The algorithm most widely available is the congestion control system found in TCP, and it provides reliability and congestion control at the same time but both functions can not be separated. A system where congestion control is kept apart from the transport protocol could provide more flexibility for applications. This alternative could make possible the transmission without congestion control or the change of the congestion system while the transport protocol stays.

The *Datagram Congestion Control Protocol*, *DCCP* [52] is a new framework designed to provide this kind of flexibility. DCCP has been designed to replace UDP, with the addition of congestion control when this functionality is required. DCCP provides an unreliable flow of datagrams with acknowledgments, with a reliable negotiation of options, including the negotiation of a suitable congestion control mechanism. This negotiation systems permits, for example, the selection of the most suitable congestion control system for the application. DCCP currently provides two congestion control systems: a TCP-like congestion control, *CCID 2* [26], with AIMD behavior but without the reliability of TCP, and a TFRC congestion control, *CCID 3* [27].

However, DCCP presents the problems of a work in progress. It is currently partially implemented in the latest Linux (kernel *2.6.x*) and FreeBSD versions, but a wide adoption can not be expected in a near future. Meanwhile, new congestion control profiles will be added and the current ones will be tested. Even although profiles change, the overall value of DCCP as a framework that provides flexible congestion control to applications is considerable.

Congestion control can also be implemented as an external framework at the user level. It can be a library where any of the systems seen in Section 2.5.2 is used for controlling the sending rate. The application has to integrate this systems with the transport protocol used for the multimedia content, that could be UDP or RTP. As we will see through this work, this integration between the congestion control system and the transport protocol can be something difficult to accomplish.

We can find user-level frameworks that solve this problem in a similar way as DCCP does at kernel level. The *Congestion Manager* framework, *CM* [5], is an architecture located between the application and the transport levels, and provides a combination of window-based and rate-based congestion control. It has been proved that the CM provides enough control for the effective delivery of media content (in particular, audio).

2.6 Real-Time Media Systems

There are several elements that define the final experience of a real-time multimedia system. The codec or congestion control used by these applications can be some of the most important design decisions, with a direct impact on what the user perceives from the system. I will now describe some of the most important technologies available in this field.

2.6.1 Codecs

Compression is a key factor in the transmission of multimedia content. Without the compression applied to the media, audio or video would need huge amounts of bandwidth in order to obtain an effective interactive experience. However, compression algorithms can make the flow more sensitive to losses. In fact, a packet loss causes a worse quality degradation on multimedia applications when compression algorithms are used. Some algorithms use a motion-compensating predictor to predict blocks in the current frame from previous frames, and then they code the residual prediction error. Because the prediction is based on the decoded signal, the model assumes the decoder shares an identical state. A packet loss leads then to a mismatch of the decoder state and quality degrades with each new frame.

To deal with this problem, this kind of codec relies on the transmission of resynchronization frames. At low bit rates, these kinds of frames can be tens or hundreds of frames away. Moreover, when a packet is lost, the system can be waiting a significant amount of time until the next synchronization frame is received, and a recovery of the signal is possible. Therefore, the probability of error in any given interval is high enough that the decoded signal could be considered not error free.

Codecs that use this scheme produce three different classes of output: intra-frames, forward predictive frames and bidirectional predicted frames. As we have seen, intra-frames are complete input images, used for resynchronization purposes, while the last ones are frames predicted from previous or next images in time order. This is the scheme followed by one of the most widely used family of codecs: *MPEG*.

The *Moving Pictures Expert Group* (*MPEG* [65]) developed their first codec, commonly knows as *MPEG-1* [33], in 1991. Since then, MPEG codecs has been successfully used with RTP [43] for the transport of media content. The last MPEG codec released is *MPEG-4* [79], which provides object-based coding, and supports the presence of synthetic and interactive content, as well as video streaming with variable bit rate.

Another popular alternative has been *H.263* [46]. This codec was originally designed as a compression system for videoconferencing on the internet, and later improved in the *H.263+* [18] standard. The latest addition to this family has been the *H.264* codec [37] (technically identical to the ISO/IEC MPEG-4 Part 10), capable of producing good video quality at substantially lower bit rates than previous codecs.

When the main objective is to increase the resistance to losses, there are two possible solutions. First, the encoding system can use some kind of packet loss recovery technique [22], or it can reduce the space between the synchronization frames and, in an extreme case, reduce this interval to a single frame. This is the model used in Motion-JPEG, where each frame is coded as a JPEG image and transmitted independently of the others. The main benefit of this technique is the latency reduction (frames can be quickly decoded, as they do not depend on future frames), but with lower efficiency than other compression systems that consider the temporal coherence of the stream.

2.6.2 Videoconferencing systems

A videoconference system is a system where all the participants of the communication can act as senders and receivers at the same time. Therefore, each member must perform capture and encoding tasks for the multimedia data captured from the local system, and this must be accomplished while data from other members is decoded and shown.

One of the most influential videoconferencing systems has been the video conferencing tool *vic*. *vic* [64] was a flexible application framework originally designed for the MBone [58] as a successor of *nv* [31]. It uses RTP or RTIP [6] for the media transport, and H.261 [44] or Motion-JPEG (with the hardware support) as codecs. However, *vic* only provides the video system, and an audio tool like *rat* [42] must be used for a full videoconferencing system.

A new application in this field has been *UltraGrid* [74]. UltraGrid is a high-definition interactive video conferencing that has been used in real-time environments, providing low latencies and high data rates [73]. These characteristics, added to the highly modular design, had made UltraGrid the ideal platform for testing TFRC in a real videoconferencing system. All these details will be provided in later chapters, starting with a description of the application in Chapter 6.

2.6.3 Congestion Control for Multimedia Applications

As we have seen, multimedia applications are bandwidth intensive and they must fairly share the available resources with other connections. The traffic they inject in the network is sometimes really significant (*i.e*, a *MPEG-2* stream is 1-2*Mbps* and *DV* can be in the 25-30*Mbps* range) and, as any other traffic, their transmission system must cope with congestion problems that happen.

This is the reason why the congestion control system is so important for the performance of multimedia applications. As we saw in Section 2.4, TCP does not always produce the best throughput stability for a multimedia system, leading to the use of different strategies in the current scene:

- Even with its limitations, TCP is a commonly used protocol for the transmission of media content. This is attributable to the resistance of TCP to firewalls and *Network Address Translation*, *NAT*, something that makes it a popular solution for streaming applications.

  As we have seen in Section 2.4, this can be acceptable for streaming, and produces satisfactory results when buffering is not a problem. TCP has been proved to be useful for media streaming with some support from the application [54], and it has been widely used for popular transmission applications [89].

  However, the buffering requirement for transmission makes it an impractical solution for interactive real-time multimedia or videoconferencing. Applications can overcome this problem if they can use several flows in parallel for transmitting the content. This enables a faster response to throughput variations but at the cost of higher redundancy and need for bandwidth. This strategy also requires some support from codecs: they must produce a hierarchically encoded output where some layers contain more information (or more important) than others. In this way, layers can be attached or removed to the transmission flow according to the sending rate that the congestion control system specifies.

- A considerable number of multimedia application use UDP, but they usually implement congestion control as a simple interchange of codecs or they do not use any congestion control at all. These application should use a congestion control implementation on top of UDP, but this is too complex or too expensive for a lot of them. Taking into account that reliability is not a requirement and fairness is not a commercial priority, these applications just do not pay attention at the network congestion and accept packet loss as something natural.

  Congestion control is completely inexistent for the UDP traffic produced by some popular applications like *Skype* [8], or some of the most important streaming platforms [57] like the *Real* [88] or *Windows Media* [41] [66] players.

  Applications using UDP must also determine if they are behind a firewall or *Network Address Translation*, *NAT*, and they must overcome the limitation using appropriate measures. The use of protocols like the *Simple Traversal of UDP through NATs*, *STUN* [86], or the *Traversal Using Relay NAT*, *TURN* [47], provides information that can be used for this purpose.

In conclusion, congestion control is something rare in the field of multimedia transmission. This is an issue ignored by software developers, even for application with more relaxed timing requirements (*i.e.* streaming). Although we have seen some solutions in Section2.5, none of them is broadly deployed. There is a need of *ready-to-use* congestion control systems and, in particular, for interactive real-time media transmission.

2.7 Summary

In this chapter we have seen an overview of the most important technologies for the transmission of real-time multimedia content. We have analyzed the characteristics of this traffic, the transport protocols used, and the problems of congestion control for multimedia applications. We have concluded that there is a need for TCP-Friendly congestion control algorithms, something fundamental for the stability of the Internet, and we have seen an overview of some alternatives to the congestion control used by TCP.

In the following chapters, I will focus on one of the TCP-Friendly solutions described:

the *TCP-Friendly Rate Control*, *TFRC*. One of the most important advantages of this protocol has been said to be a very smooth sending rate variation. This could be an ideal characteristic for a videoconferencing system where it could enhance the interactivity of the application and improve the overall user experience.

I will use TFRC as a congestion control mechanism for UDP traffic, avoiding the use of complex frameworks (*i.e.*, DCCP, CM, *etc.*, seen in Section 2.5.3) in order to study the basic properties of this new system. I will start by exploring some basic properties of the protocol, looking into the implementation issues and showing some interesting results obtained when it is used in a real system.

Chapter 3

TCP-Friendly Rate Control (TFRC)

This chapter presents the *TCP-Friendly Rate Control* protocol, *TFRC*, as it has been defined in [39]. After an overview of the main protocol features, the following sections will examine the basic characteristics of TFRC, detailing some aspects of the behavior of this congestion control algorithm.

3.1 Overview

TFRC has been defined in [39] as *"a congestion control mechanism designed for unicast flows operating in an Internet environment and competing with TCP traffic"*. TFRC does not specify a complete protocol. Instead, it is a congestion control system that can be used with any transport protocol, offering more flexibility for applications by freeing them from the necessity of a specific protocol. In this way, applications can choose between different transport protocols, like UDP or RTP, depending on their suitability.

One of the requirements that the TFRC protocol defines is the information that must be interchanged between the sender and the receiver. However, the protocol does not define how this data is really encoded or interchanged. This is something that must be specified by upper levels, either by the application or by other protocols (*i.e.*, DCCP [52] [27]).

The other part of the protocol specifies the guidelines for obtaining this information. TFRC provides the set of algorithms needed for the calculation of these parameters and, even when the protocol details the recommended algorithms for them, the application is free to obtain some parameters in more convenient ways (*i.e.*, the application could use alternative methods for calculating the round-trip time).

3.1.1 TFRC for Multimedia Traffic

As we have seen previously, TCP does not provide a satisfactory sending rate for interactive media applications, and the necessity of an alternative congestion control standard has been an imperative for a long time. The design of TFRC has been motivated by this need of an adequate congestion control system for multimedia traffic.

The main advantage of TFRC for multimedia traffic is the slow variation in the throughput of the sender. TFRC provides a smooth sending rate while maintaining the same average throughput as TCP running under the same circumstances.

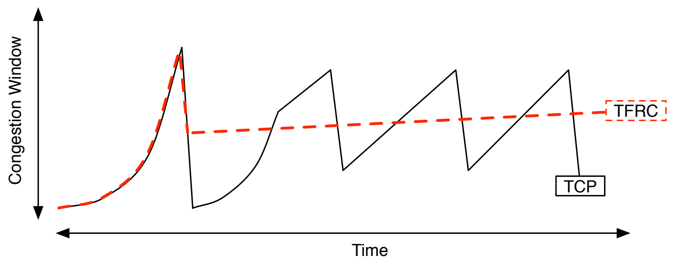


Figure 3.1: TCP and TFRC throughput.

Figure 3.1 illustrates the idealized graph of the throughput obtained by one flow using TCP and another flow using TFRC. While the TCP congestion control mechanism produces an oscillatory sending rate, TFRC tries to obtain a smoother sending rate after the slow-start phase.

This characteristic has an important implication for real-time multimedia transmission. With TCP, the system needs large buffers in order to cancel the effects of the throughput oscillation. However, with TFRC, the amount of buffering needed is reduced, resulting in a shorter end-to-end delay and an increased level of interactivity for applications.

Another advantage of TFRC for multimedia is the workload distribution. TFRC is mainly a receiver-based congestion control mechanism: as we will see in the next section, most of the operations are performed at the receiver. This is an useful feature for the transmission of media content, where the sender is supposed to be busy performing expensive processes like compression and encoding.

Nevertheless, this media encoding becomes more complicated due to a characteristic of TFRC. At a fixed input rate, codecs usually prefer to produce packets of variable size, due to the change in the amount of information in the input. The current version of the protocol uses fixed-size packets, a requirement that sometimes can be difficult to achieve when media is encoded. However, a new version of TFRC for variable packet size is currently being developed [97].

3.2 Congestion Control with TFRC

TFRC is based on the division of tasks between the sender and the reciever. Figure 3.2 depicts the main information flow between both parties, as well as the most important functions performed by them.

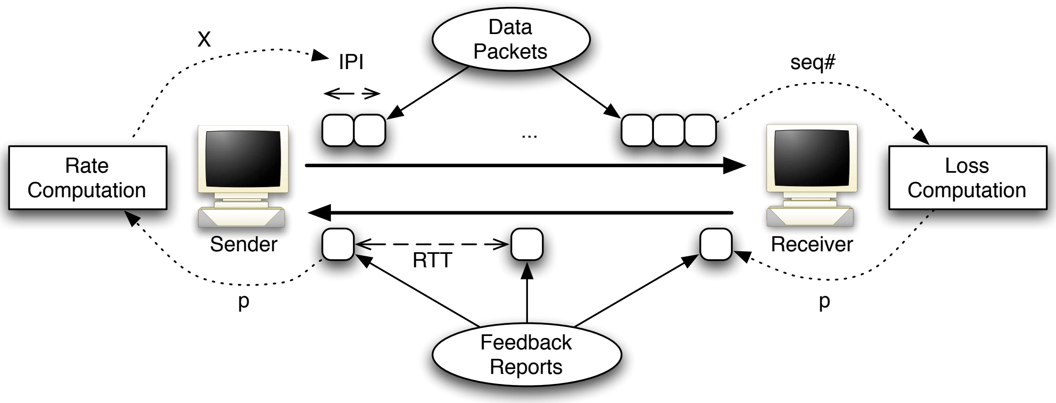


Figure 3.2: TFRC overview.

The basic responsibilities of the receiver consist in computing the loss event rate, p, and reporting it to the sender at least once per *Round-Trip Time*, RTT. Other information attached includes the time-stamp of the last packet received, the sending rate observed in the last RTT, X_{recv}, and the amount of time used for generating this report, t_{delay}.

With these details, the sender can compute the RTT and update the expected sending rate, using the TCP throughput equation. A new inter-packet interval will be calculated, and the application will space packets accordingly. It is responsibility of the sender to assign unique sequence numbers and timestamps to each packet and to attach an estimate of the current RTT for the receiver.

In the following sections we will see in more detail how all these parameters are calculated and used by both the sender and the receiver.

3.2.1 TFRC Steady State Sending Rate

The most important output of any congestion control algorithm is the sending rate. In the case of TFRC, this rate is obtained from the TCP throughput equation, Equation 2.1, that we have seen in Chapter 2. This equation specifies the throughput as a function of the loss event rate, the RTT and the packet size.

The first parameter, the loss event rate, will be calculated by the receiver and reported back to the sender. The second parameter is the round-trip time, and will be calculated by the sender using these feedback reports of the receiver. The last variable, s, is the packet size, and is known by the sender, either because all packets have the same size or because it is the mean value.

The first task that the sender must perform is to calculate the RTT. For this calculation, the sender relies on the receiver reporting the timestamp of the last packet received in the feedback packets. Using this timestamp, the sender can measure the current RTT sample, RTT_{sample}, and calculate an average RTT using Equation 3.1, resulting in RTT estimate that changes smoothly.

$$RTT = q * RTT + (1 - q) * RTT_{sample} \tag{3.1}$$

Once the new sending rate, X, has been obtained, the sender calculates the corresponding *Inter-Packet Interval*, *IPI*, as follows

$$IPI = \frac{s}{X} \tag{3.2}$$

The application must try to keep, on average, this spacing between packet. In the next chapter, we will see some issues regarding the constancy of this packet spacing, the implementation strategies and the consequences of any error.

An additional step can be introduced before the *IPI* calculation. In environments with low mutiplexing level, the increment and decrement in the network buffer length will produce fluctuations of the RTT that will result in an oscillation of X. In order to avoid this oscillatory behavior, the sender must obtain the *IPI* from an instantaneous sending rate, X_{inst}. This new rate is calculated using X and the difference between the last RTT_{sample} and an estimate of the long-term RTT.

$$X_{inst} = X * \frac{RTT_{sqmean}}{\sqrt{RTT_{sample}}} \tag{3.3}$$

where the RTT_{sqmean} is updated with every new RTT_{sample} as

$$RTT_{sqmean} = q_2 * RTT_{sqmean} + (1 - q_2) * \sqrt{RTT_{sample}} \tag{3.4}$$

where q_2 is a constant (with a recommended value of 0.9).

However, there is an upper limit in the sending rate, and it is obtained from the receiver. In order to avoid any possible overflow of the network capacity, the sending rate can not go beyond the double of the X_{recv} reported by the receiver. As X_{recv} represents the sending rate that has been observed at the receiver in the last RTT, this works as a safety mechanism when the network is buffering.

As we have seen, the spacing that the sender must use between packets is the final result of Equation 2.1. Knowing that the other components of this equation, s and T_{RTO}, are both constants, the last variable that determines the smoothness of this equation (and, in consequence, of the inter-packet interval) is the loss event rate, p.

3.2.2 Loss Event Rate Calculation

The loss event rate is a loss measurement calculated at the receiver. A loss event is defined as the loss of one or more packets in one RTT, and it is important to realize that, even when this rate is a measure of the lost packets, it is not equivalent to the packet loss rate.

The way TFRC estimates the loss event rate is critical. This loss event rate should change smoothly in an environment with a steady-state loss rate, but it should also show good responsiveness when new loss events are detected.

The basis of the TFRC loss event rate computation is the *loss interval*. A loss interval is composed by the number of packet received between two loss events, registering any group of losses that happens in the same RTT as only one loss.

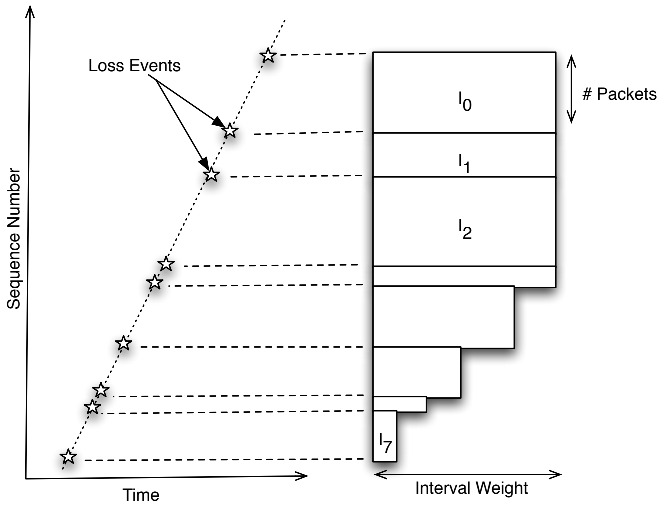


Figure 3.3: Loss intervals (adapted from [30]).

To estimate the loss event rate, TFRC calculates an average loss interval. This is the weighted arithmetic mean of the last n loss intervals, where a value of $n = 8$ is recommended in [39]. These eight intervals, from the most recent I_0 to the oldest I_7, are weighted in such a way that the four most recent are set to one and the rest decrease towards zero. Figure 3.3 illustrates this by showing the loss intervals and the graphical representation of their weights in the final calculation.

The loss event rate is obtained using two different estimations of the average loss interval. The first estimate, $\hat{I}$, is obtained as

$$\hat{I} = \frac{\sum_{i=1}^{n} w_i I_i}{\sum_{i=0}^{n-1} w_i} \tag{3.5}$$

while the second one, $\hat{I}_{new}$, uses a more recent range of intervals, from I_0 to I_{n-1}

$$\hat{I}_{new} = \frac{\sum_{i=0}^{n-1} w_i I_i}{\sum_{i=0}^{n-1} w_i} \tag{3.6}$$

and the loss event rate is finally calculated as

$$p = \frac{1}{max(\hat{I}, \hat{I}_{new})} \tag{3.7}$$

This calculation of the loss event rate has several advantages. First, it is a straightforward and light calculation, where a simple history of the last packets is needed. Second, the weight of the intervals gives more significance to the most recent events (those representing the current state in the network), increasing the responsiveness of the protocol. Finally, the shape of the weighting function, with its smooth decay, provides a soft change in the final value of the loss event rate.

Nevertheless, the calculation of the loss event rate sometimes leads to a wrong estimation of the current situation in the network. The n intervals used by TFRC are weighted in such a way that the most recent intervals have more weight than the older ones. A recommended weighting function is specified in [39] where the last four intervals have the same power, but this weighting can be improved for some cases when loss rate decreases.

In fact, if we imagine a situation where the flow has not suffered any loss since long time ago, the most recent interval, I_0, could be proportionally longer than the rest of the intervals. In contrast, the weight assigned to I_0 would be the same as the weight assigned to the rest of the four most recent intervals. In this case, the loss event rate calculated would be higher than what could be expected, showing a situation that does not correspond to the last experiences observed by the connection.

In order to avoid this effect, TFRC has an optional feature called *history discounting*. This is a *de-weighting* system that tries to reduce the significance of previous intervals when their length is far less than the average length, and long intervals can have more weight when they are more recent.

3.2.3 Slow-Start Phase

TFRC starts a transmission with a slow-start phase, similar to the algorithm used by TCP [48], where the sending rate is doubled every RTT until the first loss is detected. During this stage, the only limit for the increment applied is the double of X_{recv}.

When the first lost is detected, the receiver must calculate the first loss event rate that will be reported to the sender. However, due to the rapid change of the sending rate during slow-start, the receiver can not use the number of packets received until the first loss for computing the initial value of p. Instead, TFRC uses the X_{recv}, the receive rate for the most recent round-trip time, to synthetically calculate p and, in consequence, the number of packets that should be in the first interval. TFRC does this by finding a value, p_{synth}, for which the throughput in Equation 2.1 is X_{recv}.

However, as Equation 2.1 is difficult to invert, a simplified version of the throughput equation is used in order to obtain p_{synth}. For small values of p, Equation 2.1 can be replaced by a more simple form

$$X = \frac{s}{RTT} \frac{c}{\sqrt{p}} \tag{3.8}$$

where c is a constant in the range of 0.87 to 1.31 [62]. Using this equation, the initial p reported by the receiver will be

$$p_{synth} = \frac{s^2 c}{RTT^2 X_{recv}} \tag{3.9}$$

and, using this value, the first loss interval will be initialized as

$$I_0 = \frac{1}{p_{synth}} \tag{3.10}$$

New losses will shift this synthetic interval and, eventually, it will be discarded.

3.3 Summary

This chapter has provided a high level overview of the TFRC protocol. However, it must be noticed that not all the details of the protocol have been described in this chapter. Further details should be obtained from the reference documents.

TFRC is a new Internet standard and, as such, it is evolving and needs more experiments. The number of project where TFRC has been used in the real world is very limited, and there is a long way until it is fully tested. In this time, TFRC will most likely change, and some improvements will be probably needed in order to adapt the protocol to new scenarios.

In the following chapters I will try to bring more information about TFRC, focusing on some issues for the use in the real world. First, an analysis of the protocol will be performed from the point of view of a system designer, followed by the integration of TFRC in a videoconferencing application.

Chapter 4

TFRC Implementation

In the previous chapter, we have seen the basic elements of TFRC. Any system using TFRC as a congestion control mechanism must perform several essential tasks. Figure 4.1 shows the basic layout of any TFRC system, representing the basic tasks as boxes and information dependencies as simple lines.

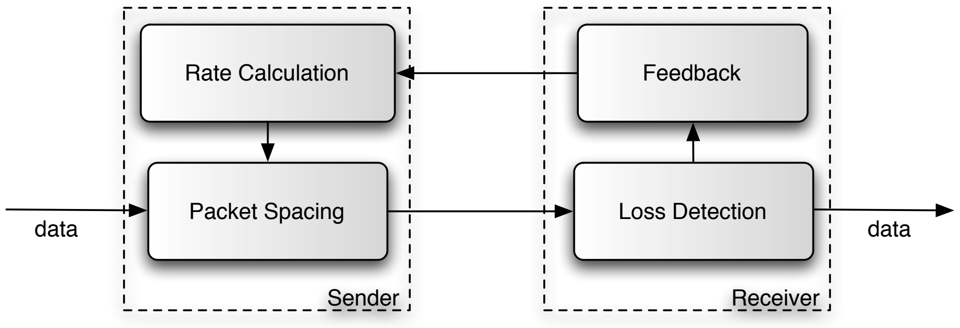


Figure 4.1: TFRC tasks and information flows.

First, the sender must receive the loss event rate calculated by the receiver and use it to obtain the sending rate. Using this rate, the sender must calculate the *Inter-Packet Interval*, *IPI*, and space data accordingly.

The receiver must perform other essential tasks. At first sight, it seems that it does not need as much precision as the sender, so all these tasks could be implemented in a single thread: to receive and process packets, to detect losses and to pass the data to the application. The loss event rate calculated must be reported to the sender as soon as possible, being the only function where time plays an important role.

In this chapter I will try to explain some of the peculiarities of TFRC and how they lead to a wide range of difficulties when it must be implemented. Some of these problems apply to any rate-based congestion control system, like the correct packet spacing and some performance issues. I will focus on the problems found on the implementation of TFRC, highlighting some problems that arise from the way the protocol is designed, *i.e.* dependencies with the operating system.

I will start the chapter with an overview of the accuracy needed by the different tasks of TFRC. In Section 4.1, the sensitivity and requirements of the protocol will be described, as well as the problems that arise when these requirements are not achieved. The discussion will continue in Section 4.2, where I will focus on implementation issues of how basic tasks as performed, and the problems that arise when they are implemented as separate threads or when they are functions of the same process. The chapter will finish with the description of the system implemented in Section 4.3, giving details of the internal design and the decisions that led to them.

4.1 Accuracy of Processes

The implementation of a congestion control system is a difficult task. Besides the complexity of the mechanism, a rate-based algorithm adds some performance issues that can be difficult to overcome. The congestion control system should have some resistance to these performance errors and should not depend on errors produced by external factors.

In the following sections we will find that TFRC is extremely strict in some situations. We will study the accuracy needed by the algorithm and how the different processes can be implemented for better performance and precision.

4.1.1 Timing Errors

A rate-based congestion control system is based on the idea of packet spacing. It uses the time between packets as the main technique for modulation of the sending rate. Consequently, it is a key point for this kind of algorithm to avoid any long periods of unexpected inactivity, both for the sender and the receiver: implementations must have a good level of *awareness* in order to generate, send and process packets or acknowledgments at the right time.

An important part of this mechanism is based on *sleep* operations. For example, if we focus on how the sender can implement the sending process, we can see that it can adopt one of these different strategies in order to space packets:

1. The sender can use functions provided by the operating system for sleeping between packets. The use of this sleeping function, `sleep()`[1], will interrupt the execution of the sending thread for t_{sleep} seconds. The operating system adds some error to this call, and the program execution will really be resumed after $t_{sleep} + t_{error}$ seconds, where t_{error} is the error in the operating system due to system latencies and other possible limitations in the timer resolution of the hardware. Our experiments show that we can expect a mean error, $\overline{t_{error}}$, near $5ms$[2].

   Small errors will not only force the urgent delivery of some packets, but they also have another consequence: when `sleep()` has t_{error} seconds of error, the sender is not sending data for t_{error} seconds. As we will see in Section 4.1.2, this can result in wrong values for X_{recv} and a following reduction in the next X calculated.

2. The sender can avoid any sleeping function and send packets ignoring the *IPI*. In this case, the sender should periodically check the partial sending rate in order to verify that it is not sending too fast. When it detects that it must reduce the sending rate, the sender must stop and sleep for some time, resulting in the same problem that we have seen in the previous strategy. We will also see some problems related to the measurement of the real sending rate in Section 4.1.4.

3. The sender can avoid the use of the `sleep()` function provided by the operating system and implement its own version using a *busy-wait* loop. In this case, the sender can consume too much CPU for the inter-packet spacing[3], and the maximum throughput of the system can be eventually limited.

In conclusion, the sender can send packets at the right rate using some kind of controlled, delayed loop or an alarm. In any case, the sender depends on the accuracy of the system timing (unless CPU consumption is not important). Any sleep or alarm can easily be performed when the inter-packet interval, *IPI*, can be measured in milliseconds. However, with shorter RTTs and higher rates, the *IPI* can be of just a few milliseconds or even microseconds, and a sleep error can result in no packets being sent in one (or

[1]This function will be implemented with functions like *nanosleep()* or *select()* in Unix systems.

[2]This value highly depends on the operating system, hardware and system load. Mac OS X shows the highest accuracy, with an error in the range on microseconds. However, we can expect more than $10ms$ error from a Linux machine with a medium load.

[3]In our experiments, it consumes 99% of CPU in a *Pentium IV* at *2.8Ghz*

even more) RTTs.

The same happens for the receiver: it must try to achieve the timing accuracy when it sends feedback reports and be responsive and process packets quickly. The same problems apply here: it is easy to do for long periods of time, but it gets harder when, for example, the RTT gets shorter.

4.1.2 Sender Errors

The main information, from an application point of view, provided by the TFRC system to the sender is the *IPI*. The inter-packet interval represents the sending rate as the time that must elapse between packets. If the sender does not send at the required rate, the traffic generated can overflow the network resources or otherwise waste bandwidth that is available.

If we denote by t_{IPI} the *IPI* specified by TFRC, and by t_{ipi} the real time elapsed between two consecutive packets. The difference $t_{IPI} - t_{ipi}$ should be zero in an ideal case, however sleeping errors can produce a persistent difference between the expected *IPI* and the real sending time.

In Figure 4.2 can be seen the current *IPI*, t_{IPI}, and the real time elapsed between two consecutive data packets, t_{ipi}, for a connection with $200ms$ RTT. The experiment is run between two Linux machines, using *busy-wait* sleeps in the sending loop. It can be observed a lag when sending packets, with a mean error ($\overline{t_{error}}$) of a couple of milliseconds. This delay is insignificant for a long RTT like this, where it represents no more than the 2.5% of the current RTT.

The same error can be seen in Figure 4.3, but using a RTT of just $3.5ms$. If we compare this graph with the previous one, we must notice the different time scale of the *IPI* represented here: it is just one or two milliseconds. In this scenario, the sending error can be of up to 50% of the RTT, and the time between two consecutive packets could be longer than the current RTT. Even longer errors in the sleeping system can occur if the sender is resumed too late or the operating system interrupts the process.

This kind of error is insignificant for long RTTs, but it becomes far more concerning when it happens with short ones. In such situations, the following sequence of events can produce an oscillation in the sending rate:

1. First, a relatively high $\overline{t_{error}}$ results in the sending of less packets than expected. In this example, with a $3.5ms$ RTT, a $\overline{t_{error}} = 5ms$ can result in more than one

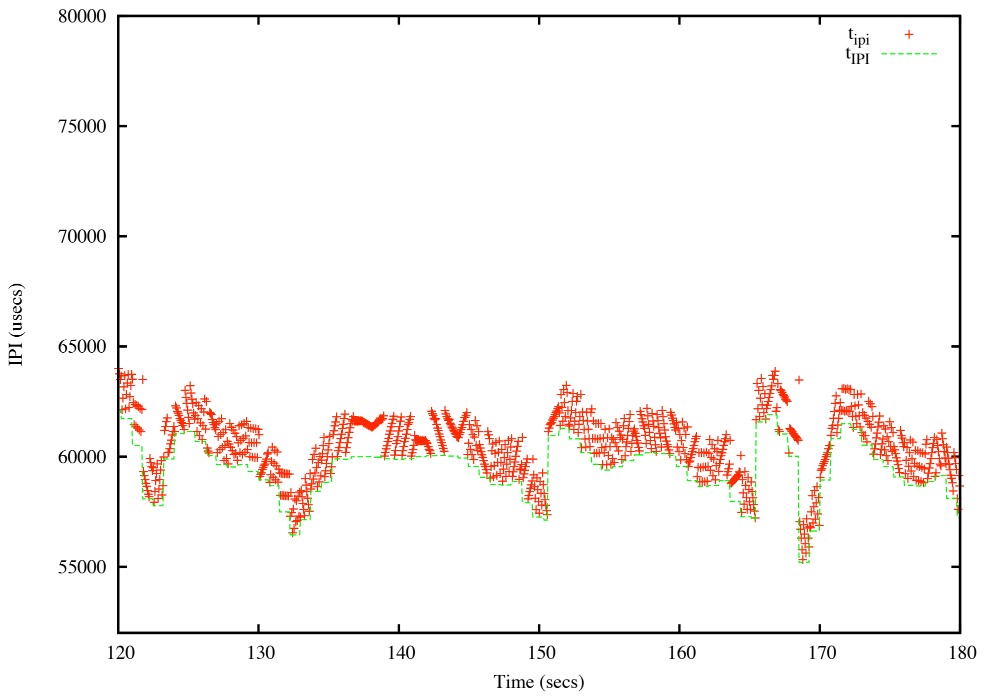


Figure 4.2: Expected IPI (t_{IPI}) and real time between packets (t_{ipi}), with $200ms$ RTT, $200Kbit/s$ bandwidth.

RTT without packets sent.

2. For the next acknowledgment, the receiver computes a very low or null X_{recv} using the amount of data received in the last RTT. In our example, the sender can reports a value of $X_{recv} = 0$ to the sender.

3. The sender calculates the new sending rate, X, but, as there is a limit where $X < 2 * X_{recv}$, a wrong value of X_{recv} limits the new sending rate. For this example, the new sending rate can easily be 0.

In general, this situation will be the direct result of the $X < 2 * X_{recv}$ constraint. Provided that the next sending rate calculated, X', should be not less than the current rate, X, any t_{error} that reduces in more than 50% the amount of data sent in the next RTT will produce a limiting X_{recv} and result in a situation where $X' < X$. When this error happens, the sender tries to recover during several RTTs. This rule is once again a limitation, and if this takes place too frequently, the sender will never recover and the system will oscillate forever.

This can be a problem in some scenarios where $t_{ipi} - t_{IPI} > RTT$. For example, the

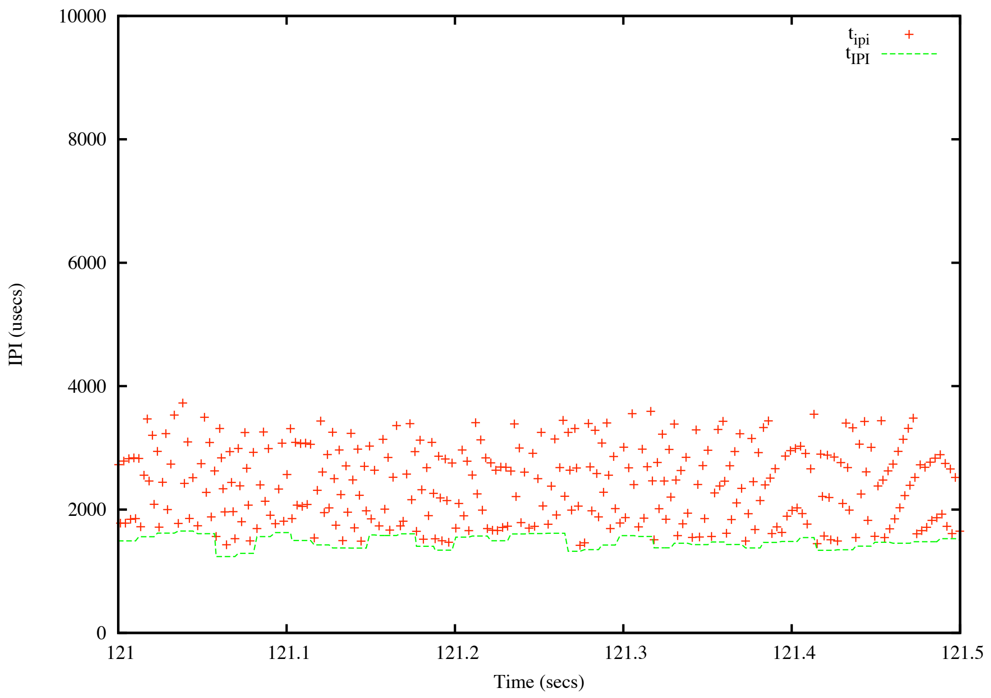


Figure 4.3: Expected *IPI* (t_{IPI}) and real time between packets (t_{ipi}), with *3.5ms* RTT and *8000Kbit/s* bandwidth).

RTT measured for national connections can be less than $10ms$[4], and local area networks are even more susceptible to this problem. In a high speed network, where there is a low level of buffering, a TFRC receiver will be prone to compute a false X_{recv} when the difference $t_{ipi} - t_{IPI}$ is far from the current RTT.

Another side effect of these sleeping errors is that the sender must be forced to send long sequences of packets. For a constant *IPI*, t_{IPI}, a sleeping error where $t_{error} \geqslant n * t_{IPI}$ (with n some integer number) will force the rapid delivery of n packets. In general, a sender will need $max(X) * max(t_{error})$ bytes of buffering in order to accept this bursty behavior, and intermediate hosts will also need more buffering for this traffic.

4.1.3 Sending Rate Errors

As we have seen, it is not so easy to achieve the desired sending rate. The packet spacing presents some difficulties that can lead to periodic errors. In general, we can distinguish

[4]In our experiments, the RTT measured between Glasgow and Edinburgh is $3ms$, and Glasgow to London can be in the range of $10ms$ to $15ms$.

several types of sending rate errors:

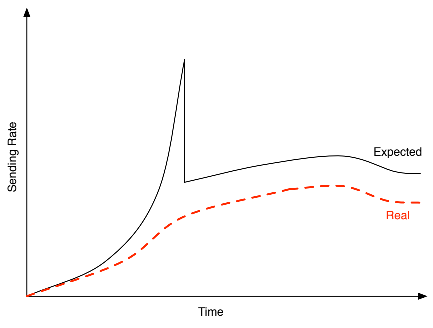
- The sender can have some delay in the sending loop that reduces the speed of the process. This represents a *Type 1* error, as illustrated in Figure 4.4(a).

  As we will see in the following section, this is a very common class of error and is generally produced by some operating system dependencies. The preemption of the current thread, some system call latencies or the time granularity of the *OS* scheduler can be the cause of this periodic error. We will see in Section 4.2 what we can do in order to avoid this problem and how this affects the design of the sender.

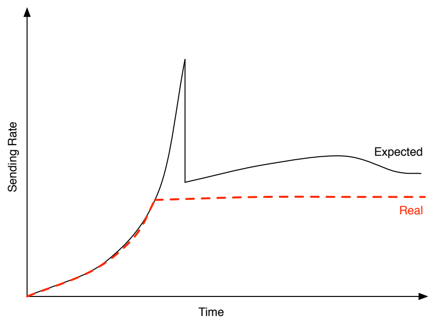
- The sender can have a hard limit in the amount of data that can be transmitted. This is a *Type 2* error, and it corresponds to Figure 4.4(b). This can be a limit established by the *CPU* power or by the network capabilities of the machine.

  Considering the slow start phase, for example, we can see that the increment in the sending rate can lead to quite high speeds that could be unreachable for some machines, or simply out of the physical possibilities (*i.e.*, TFRC can easily set the sending rate at *16Mbit/s* even if the network adapter can only transmit at *10Mbit/s*). Some experiments have shown that this can lead to senders unable to reach the bottleneck bandwidth, and then unable to go out of the slow-start phase.

In conclusion, a rate-based congestion control algorithm must be resistant to these errors. These systems should not depend on the real sending rate or, at least, they should include some kind of correction technique.



(a) Type 1: constant delay



(b) Type 2: hard limit

Figure 4.4: Sending Rate errors.

4.1.4 Sending Rate Correction

The continuous presence of inaccuracies in the sending time could result in a wrong *Inter-Packet Interval*, *IPI*, and, as a consequence, a sender rate that is under or over the expected rate given by TFRC. However, the sender could try to estimate the real sending rate, tracing inaccuracies caused by the operating system behavior. Then it could compare it with the expected rate given by TFRC and, if there is any difference, rectify it.

Several problems arise when the sender tries to calculate the real sending rate, X_{real}.

- The sender could try to measure the real sending rate between packets, obtaining it as s/t_{ipi}, where s is the packet size. However, as we have seen, the high frequency of the process and the sending time inaccuracy produces a highly variable t_{ipi} and, as a result, an unstable real rate.
- The alternative should then be the measurement of the real sending rate at a lower frequency (*i.e.*, every RTT) using the number of packets sent in this interval. This could produce a more stable estimation of X_{real}, closer to the real value, but it could be too different from the expected rate ($X_{real} \ll X$ or $X_{real} \gg X$).

So, even if the sender has a accurate X_{real} approximation, TFRC lacks any correction system where this real sending rate could be used. Simply substituting the expected X by X_{real} could produce too abrupt changes and a possible oscillation. A valid mechanism for adjusting TFRC once we know the X_{real} is still an open issue that must be solved.

In summary, TFRC should include a rectification system designed for the sending rate. It does not consider the case where the sender is unable to reach the specified sending rate, and the lack of a feedback system is something that should be solved in future versions of the protocol.

4.1.5 Receiver Errors

The receiver must report the sending rate seen once per RTT. This rate, X_{recv}, is used for limiting the sending rate, X, at the sender, in such a way that $X < 2 * X_{recv}$. Using the double of X_{recv} as a limit, the sender can ensure that the sending rate does not go far beyond the bottleneck capacity.

Thus, not only the sender must produce the right amount of data in one RTT, but the receiver must also precisely measure the X_{recv} reported. As we have seen in Section 4.1.2, any discrepancy between X and X_{recv} where X_{recv} computes less than a 50% of the data received in the last RTT would result in a report that could limit the next X.

Although an error of 50% can seem a quite high error, there are some situations where this can easily happen:

- At low sending rates, the sender can deliver a small number of packets per RTT. In this case, the precision of the sending process and the network jitter will determine the accuracy of the X_{recv} calculation. Packets arriving late at the receiver will be computed in the next RTT, producing an oscillation where X_{recv} will be underestimated.

  Figure 4.5 shows the number of packets used at the receiver for the calculation of X_{recv}. This was obtained in an scenario with $800 Kbit/s$ bandwidth and $800 ms$ RTT. It can be noticed that some oscillation takes place due to the fluctuation in the number of packets received in every RTT. In general, it is a insignificant fact, as X_{recv} is only used for limiting X. However, this can be concerning when the system in still in the slow-start phase.

  We can see an example of this problem in Figure 4.6. It shows the sending rate at the sender and at the receiver for the previously seen connection. The amount of buffering in the network is artificially high in order to intensify the effect in the slow-start phase.

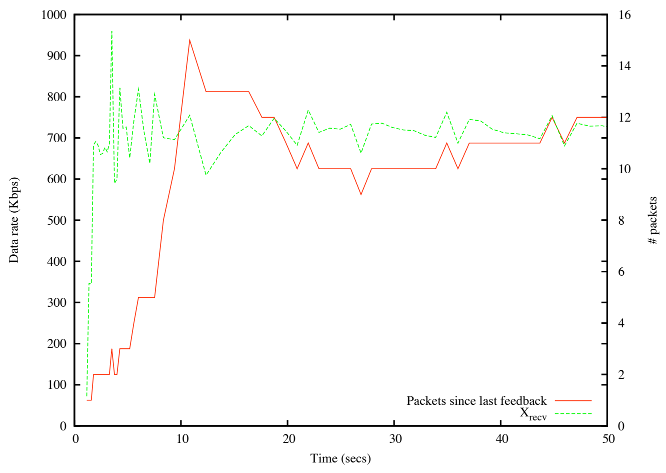


Figure 4.5: X_{recv} and packets used for calculation

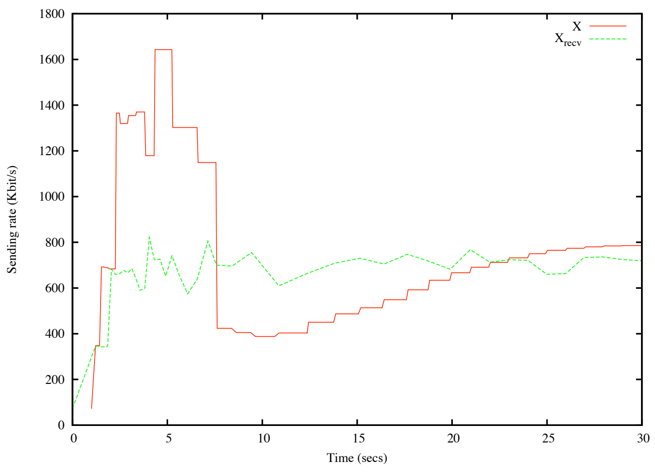


Figure 4.6: Sending rate oscillation in slow-start, due to X_{recv} oscillation

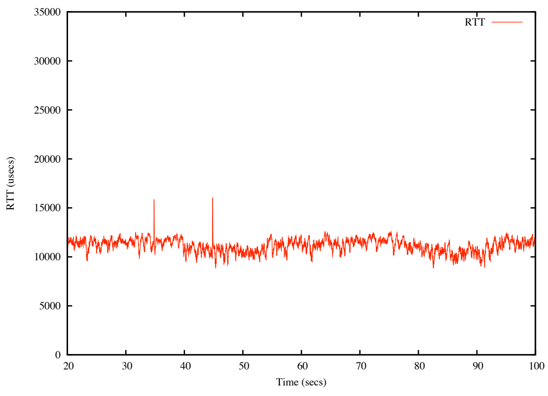
In this case, the variation in the data arrival produces an oscillatory X_{recv} during slow-start. This value is reported back to the sender and continuously limits the new X calculated during slow-start, resulting in an oscillation of X during this phase.

As we can imagine, this problem only appears in very particular scenarios, with long RTTs and short bandwidths. In most cases, the slow-start phase will be too short to appreciate this problem.

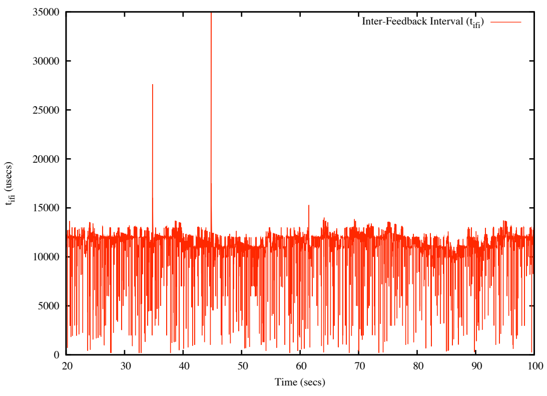
- The receiver can also inaccurately calculate X_{recv} due to late activation. If the receiver computes the X_{recv} every t_{ifi} seconds, where t_{ifi} should be equal to the last known RTT, the difference $RTT - t_{ifi}$ will determine the precision of the X_{recv} reported. In general, we can assume that there will be some mismatch, as the receiver will implement this task as a separate thread or with a sleep operation (see Section 4.2). Once again, the receiver also depends on the accuracy of the sleeping process.

  As we already saw in the sender case (in Section 4.1.2), this problem is insignificant for long RTTs, where the $RTT - t_{ifi}$ difference is almost 0, but it can be a problem for short ones. For a scenario where the RTT is $3.5ms$, the receiver must send a feedback report every $3.5ms$. At this scale, the difference $RTT - t_{ifi}$ can be very unstable due to the sleeping inaccuracies.

  So, with short RTTs, the X_{recv} will not be updated every RTT but at a highly variable frequency. If the sender calculates X_{recv} using the number of packets



(a) RTT



(b) Time between packets (t_{ifi})

Figure 4.7: RTT and time elapsed between feedback packets.

received and the time elapsed since the last feedback, the result will be quite unstable.

The calculation of the right X_{recv} is not the only difficulty for the receiver. The $RTT - t_{ifi}$ difference also shows another problem: the receiver can be late sending feedback too. The same problems found in Section 4.1.2, where the sender could be late sending packets, apply here too. This is illustrated in Figure 4.7, where we can see the time elapsed between two consecutive feedback packets (t_{ifi}) as well as the current RTT reported by the sender.

In a scenario where the RTT is $3.5ms$, the receiver must send a feedback every $3.5ms$, and a $\overline{t_{error}} = 5ms$ could result in no feedback sent in the next RTT. Figure 4.7(b) shows the real time between feedback packets in this case. We can see a large amount of early feedback packets (due to packet losses), but also a periodic late delivery of a couple of milliseconds.

In general, we can set the error limit by knowing that the sender waits for $4 * RTT$. If it does not receive a feedback in this time, it will halve the sending rate. In our example, this sets the maximum error at $14ms$. In Figure 4.7, we can see two peaks in the t_{ifi}, where the receiver is inactive for more than $14ms$ and the sender will consider that a report has been lost, but things could get worse with a lower RTT.

4.2 Threads

In order to work, an application using TFRC must accomplish two different tasks in the sender: the reception of feedback reports and the sending of packets. The receiver also depends on the implementation of the opposite functions: the reception of packets and the generation and delivery of feedback reports. The way these functions are integrated in the system leads to several design choices, depending on the number of threads used.

In the following sections, I will describe how the number of threads affect the TFRC implementation. I will focus on the sender case, and I will just overview the receiver as, basically, it presents the same group of problems.

4.2.1 Multi-threaded sender

At first sight, using more than one thread for the sender could seem a good idea. The functionality could be split in two different concurrent threads, and the sender could increase the accuracy of the sending process, as the time elapsed between packets will be established by a `sleep-until()` function (probably using a *busy-wait* loop). The sensitivity for feedback reports could also increase, as a dedicated thread would wait for them and they would be processed immediately.

Algorithm 4.1: Multi-thread sender: sending thread.

```
while true do
    if sender can send then
        send-packet ();
    end
    next-time = sender-next-time ();
    sleep-until (next-time);
end
```

Algorithm 4.1 shows the basic layout of the sending thread. The sender will perform the packets delivery in one thread, while other thread will handle the reception of feedback reports. Algorithm 4.2 shows how this reception is performed at the sender.

However, some difficulties arise from the timing accuracy of the host operating system.

Algorithm 4.2: Multi-threaded sender: reception thread.

```
while true do
    timeout = $\infty$;
    ack = sender-wait-ack-for (timeout);
    if new ack then
        sender-process-ack (ack);
        sleep-until (next-time);
    end
end
```

- First, in order to improve the precision of the sending process, the sender must use a *busy-wait* loop for sleeping between packets. This will make the sending thread consume all the quantum assigned by the scheduler, yielding the processor only when all the time has been used or when it is preempted by a higher priority thread.

  Under these circumstances, the sender will not be continuously sending packets. As the sending thread is preempted, the sender interrupts the data delivery and these inactivity periods will push it to send long packets sequences in order to keep, on average, the same throughput than TFRC.

  We can see this effect in Figure 4.8, with an *IPI* of $70ms$. There are two parameters represented in this graph: the expected *IPI* (t_{IPI}), and the same variable plus the t_{error} calculated. The result is a scenario where the sender often sends packets at the wrong time.

  Although this can be insignificant with long RTTs, the late delivery of packets with short RTTs could result in a limiting X_{recv} and a subsequent low X. For example, for a standard Linux system, the OS interrupts non-blocked processes every 1/100th second (*1/1000*th in Linux/Alpha), assigning a *10ms* quantum to an alternative thread. Then, for a RTT equal or lower than *10ms* , the sending thread could easily be inactive for one or even more RTTs. Depending on the current buffering present in the network (and the local sending queue), the receiver could see a null sending rate, calculate a $X_{recv} = 0$ and report it in the next feedback.

- The reception thread of the sender will not really process feedback reports continuously. In fact, we must add to the error produced by `select()` the error resulting of the late activation of this thread. If we consider a system where only the sending

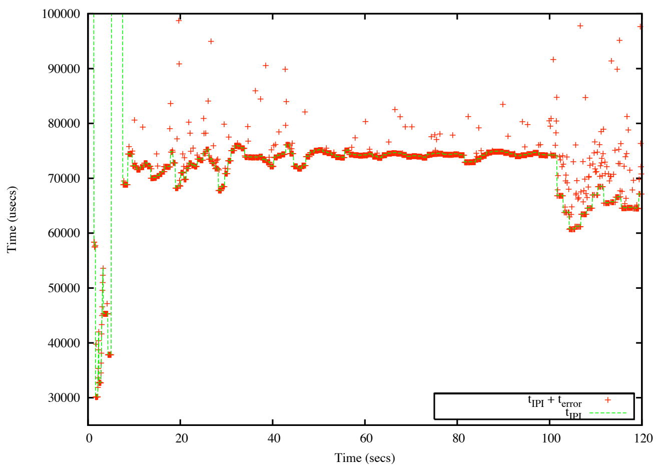


Figure 4.8: Sender delay in a multi-threaded sender.

and the receiving threads are scheduled with a t_q quantum time, and an average error $\overline{t_{error}}$ for every `select()` operation, a feedback report will be processed, on average, with a $(t_q/2) + \overline{t_{error}}$ error.

This error can be excessive for short RTTs. For a $\overline{t_{error}} = 5ms$ and $t_q = 10ms$, the sender will react to any feedback report with a $10ms$ average delay. This is insignificant for a $180ms$ RTT, but huge when the RTT is just $3.5ms$.

In this situation, the only alternative would be to force the check of feedback reports, yielding the processor after sending every single packet. However, this would mean that the operating system must perform a quite large number of context switches and locks, and the associated overhead could limit the final performance of the system. This results in a *Type 1* error (as it has been defined in Section 4.1.3) where the sender would be unable to reach the expected sending rate.

As we have seen, there is no single solution for some of these problems, in particular if there is no special support from the operating system. The implementation of this two functions as separate threads implies a simple consequence: when one is running, the other is stopped. This trivial result seems to be against the use of TFRC with short RTTs.

This demonstrates that a TFRC implementation using a muti-threaded sender could sound a good idea, but it is difficult to successfully implement at the application level.

Even although TFRC implementations at kernel level could get big advantages regarding delays, priorities and threads, there are still some difficulties to overcome when TFRC is used with short RTTs.

4.2.2 Single-threaded sender

With this option, the sender will accomplish both tasks using a single thread, making use of a loop where packets are sent at the right time and polling the reception queue looking for any new feedback report. Pseudo-code for this solution is shown in Algorithm 4.3. However, this approach can present a timing problem, depending on the operating system used and the current data rate.

Algorithm 4.3: Single-threaded sender: basic sending loop.

```
while true do
    if sender can send then
        send-packet ();
    end
    next-time = sender-next-time ();
    timeout = next-time - now;
    ack = sender-wait-ack-for (timeout);
    if new ack then
        sender-process-ack (ack);
        sleep-until (next-time);
    end
end
```

In this solution, `sender-wait-ack-for()` establishes the real time between packets, t_1. If we call t_2 the time slept in `sleep-until()`, the addition of t_1 and t_2 should be equal to the inter packet interval. However, in order to promptly respond to new feedback reports, the sender should spend as much time as possible in the first function, and use the `sleep-until()` function for fine tuning the sending process.

If we focus on how this is done, `sender-wait-ack-for()` will be commonly implemented using a `select()` system call, or any other operating system function that will wait for a maximum amount of time for data in the queue. However, this kind of functions frequently returns before or after the *timeout* has been exceeded. In contrast with `sleep-until()`, that can be implemented using a *busy-wait* loop, the accuracy of this function depends on the operating system and can not be improved with external methods.

If the difference between the timeout and the real time elapsed is called $\overline{t_{error}}$, positive values of $\overline{t_{error}}$ will increase the time in `sleep-until()` where the sender will be unaware of feedback reports for $t_2 + \overline{t_{error}}$ time. In the same way, negative values of $\overline{t_{error}}$ will reduce t_2, even avoiding any sleep at all and producing a late send. Then, for big enough values of $\overline{t_{error}}$, either positive or negative, the system will be either unresponsive to feedback reports or late sending packets.

For example, for a typical $\overline{t_{error}} = 5ms$ average error in modern operating systems, there will be a limit of $5ms$ in the real *inter-packet interval*, t_{ipi}, for a perfect accuracy. Any lower value will make t_2 equal to 0 and send packets with an average $\overline{t_{error}} - t_{ipi}$ error.

Several changes can be made in order to reduce the effect of this error. First, the system must lower the responsiveness to feedback reports, reducing the timeout for the feedbacks reception. Second, we must take into account that `select()` operations can return after $\overline{t_{error}}$ even with a null *timeout*. In this case, we will have a *Type 1* sending rate error: the sender will not reach the expected sending rate due to the constant delay introduced by `select()`. This can only be avoided looping over the sending process, in such a way that the required sending rate can be reached but at the cost of sending packets together.

Algorithm 4.4: Single-threaded sender: enhanced sending loop

```
while true do
    while sender can send do
        send-packet ();
    end
    next-time = sender-next-time ();
    timeout = (next-time - now) - $\overline{t_{error}}$;
    ack = sender-wait-ack-for (timeout);
    if new ack then
        sender-process-ack (ack);
    end
    sleep-until (next-time);
end
```

Algorithm 4.4 shows the new version of the code. In this form, the sender loops trying to send more data. When the condition is no longer satisfied, then it starts the other tasks. The downside is the throughput oscillation that this approach can exhibit due to its bursty nature.

In order to support these sequences of packets, the host *OS* must be able to accept big chunks of data in its network buffers. The TFRC sender must use the formula specified in Section 4.1.2 for the calculation of the amount of buffering needed.

The solution shown in Algorithm 4.4 requires a new calculation. The sender must keep track of the sleeping errors, and compute a mean error, $\overline{t_{error}}$, comparing two successive loops. In order to be early sending packets, the sender must subtract this mean error from the *timeout*. This can improve the accuracy while $t_{IPI} > timeout \gg \overline{t_{error}}$ is true, but there is not much it can do when $t_{IPI} < \overline{t_{error}}$

This approach is still far from perfect. As we said, the sender can be locked for $\overline{t_{error}}$ waiting for feedbacks, even when $timeout < \overline{t_{error}}$. In our example, the sender will be, on average, locked for $5ms$ in `sender-wait-ack-for()`. For scenarios with short *RTT*s and high bandwidth, this would result in very long sequences of packets, and maybe a *Type 2* error.

Figure 4.9 describes this kind of problems as a function of the current *Inter-Packet Interval*, *IPI*. The first case shows a long *IPI* case, where the $\overline{t_{error}}$ is proportionally short and will produce some inaccuracies in the sending time of the next packet, but nothing serious. When the *IPI* gets shorter, the significance of the *timeout* is lower, and the `select()` takes, on average, $\overline{t_{error}}$ seconds to complete. This results in a rapid delivery of packets, but not as concerning as the last case where the *IPI* is really short. In this case, the timeout is ignored, but the check for feedback packets must be done. The sender will need long packet sequences in order to reach the expected sending rate. If we add the time needed to send these packets, and some external factors like the *OS* scheduling, the sender can need very long sequences indeed. In some cases, the sender would even have to forget the feedback reports and *skip* some calls to `sender-wait-ack-for()`.

In the last case, the sender could avoid the constant delay induced by the operating system by interleaving these checks for feedback packets. Now the problem is for how long we can forget about feedback packets. Long periods could lead to very low responsiveness, but shorter values would reduce the performance. Although we know that there should be some relation between the *RTT* and the interleaving time, this is still an open issue that must be answered.

Despite the problems shown for this solution, it is the recommended design for a TFRC

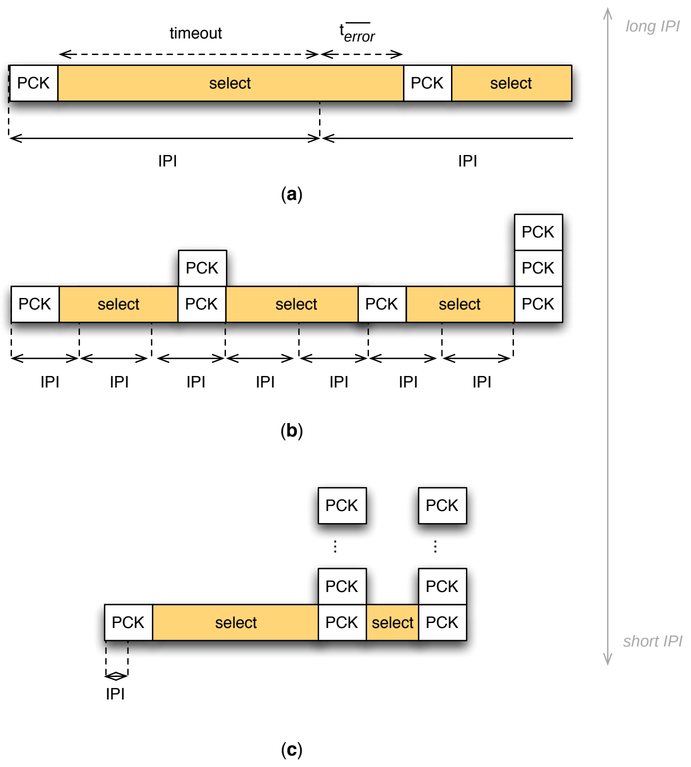


Figure 4.9: IPI and timing errors.

sender. It is the only viable solution where the sender can produce, in most cases, an average rate similar to the rate specified by TFRC.

4.2.3 Receiver

In the same way as the sender must divide the sending and reception tasks in one or two threads, the receiver must be designed considering the same tasks but with packets and reports in the opposite direction. The receiver has to wait for data packets in a loop, and it must generate feedback reports when they are due. So, most of the problems we saw in Sections 4.2.1 and 4.2.2 apply here too.

Taking into account these considerations, we can dismiss as inadequate a multi-threaded design. Reception of packets would be easily accomplished, but latency and scheduling could lead to the late delivery of feedback reports. As a result, the recommended design would be a single-threaded receiver, following the pseudo-code shown in Algorithm 4.5.

Algorithm 4.5: Reception loop

```
while true do
    if receiver can send then
        calculate-feedback ();
        send-feedback ();
    end
    next-time = receiver-next-time ();
    timeout = (next-time - now) - error;
    pck = sender-wait-pck-for (timeout);
    if new pck then
        sender-process-pck (pck);
    end
end
```

This algorithm is optimized for the accurate generation of feedback reports. In this case, to receive packets becomes a secondary task, as unprocessed data will be simply stored in the *OS* buffers. So, the receiver tries to send feedback reports after one *RTT*, and it also uses this value as the timeout for the reception of packets. The mean error, $\overline{t_{error}}$, is subtracted in order to improve the sending time: the receiver could use a *busy-wait* sleep when the next sending time is less than $\overline{t_{error}}$.

However, the same problems seen for the sender can result in the late generation of feedbacks. When the receiver waits for packets, the latency of the `select()` system call will return the execution after $\overline{t_{error}}$ seconds, and this can be a problem for small *RTT*s ($\overline{t_{error}} \gg RTT$).

In conclusion, the receiver must perform two tasks with the highest possible precision. If we compare it with the sender, the functions of the receiver are characterized by longer time scales. Although this simplifies the design, there are still some constraints that must be considered. I have presented a possible optimal solution for the average case in Algorithm 4.5, but we can not forget that it will exhibit serious problems with short *RTT*s.

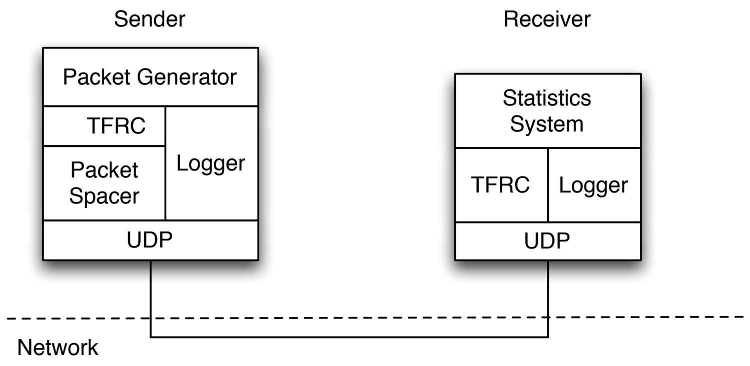


Figure 4.10: TFRC Traffic Generator.

4.3 Traffic Generator

Testing a congestion control algorithm like this requires a traffic generation tool. This generator has been designed taking into account all the problems shown in previous sections. We have focused on a system that follows the specification of the protocol [39], without *history discounting* and with *oscillation prevention*. A modular and flexible design has made possible the evaluation using different alternatives like, for example, the number of threads or the specific tasks performed by each one.

All this information has determined the final design of the traffic generator, looking for a balance between performance and conformance with TFRC. Although this balance should be ideally facilitated by the protocol, the truth is that this is difficult to achieve. However, some of these problems could be solved with some *OS* support or by implementing the protocol at a different level. Figure 4.10 presents the final architecture of the traffic generator used in these experiments.

At first sight, a design with a multithreaded sender could make sense: one thread could be used for the difficult task of packets spacing, and the other thread could do secondary tasks like the reception of feedback reports and any rate calculations. However, and after some testing, this approach was not found reliable. As we have seen in Section 4.2.1, if we want a responsive system with short RTTs, the sending thread must yield the CPU after every packet sent. However, with the overhead of this procedure, the sender could barely generate more than one hundred packets per second in Linux, Mac OS X and FreeBSD. This is a non-acceptable limit in the performance of the sender.

After trying different alternatives, the system has been finally used with a single threaded sender. This configuration can generate enough traffic and, at the same time, be responsive to network changes in an average situation. However, as it was noted in Section 4.2.2, it increases the throughput oscillation and can result in long sequences of data that the network adapter must accept.

At the receiver side, all the tasks have been integrated in a unique thread. A muti-threaded operation mode could also be configured at compile time, but tests have shown that this approach reduces the responsiveness considerably. In the same way as happened with the sender, yielding the CPU has also resulted in a big performance bottleneck and, in consequence, an unacceptable solution.

Finally, at the transport level, the traffic generator uses UDP for the transmission of packets. The TFRC information is encoded inside and transmitted as simple strings, while the rest of the packet is padded with uninitialized data.

4.4 Summary

In this chapter, we have discussed some of the implementation aspects that must be considered when TFRC is implemented. We also have seen some of the problems that arise from a rate-based congestion control algorithm and how they affect the implementation details. In general, we can say that the receiver presents the same problems as the sender, and these problems come to light when we look for a suitable design for a high performance environment.

This duality between sending packets and processing feedback information leads to the hardest problems in implementing TFRC. A lower bound in the feedback reports frequency leads to lower responsiveness when a loss is detected. In contrast, a TFRC sender that does not send a feedback report every RTT results in an unstable system, as I will demonstrate in the following Chapter. Stability and responsiveness seem to be difficult to achieve at the same time in TFRC.

From an implementation point of view, I have demonstrated that a multi-threaded sender is not an acceptable solution. Nevertheless, a single-threaded approach also has a big problem: the sender could not reach the expected sending rate. The reason for this problem comes from the TFRC mechanism that forces the sender to check for a feedback: it introduces a constant delay in the loop, and we can not avoid this check or even interleave it. Otherwise, we would wonder after how many packet we should look for a report or, in other words, how much responsiveness we want for our algorithm.

The algorithm seems to depend on the endpoint's performance, with their operating systems and hardware capabilities playing an essential role. Some difficulties produced by the late delivery could be solved by implementing TFRC as a *sender based* protocol. Then the only task performed by the receiver would be to acknowledge the data packets with the help of some kind of *selective acknowledgment* system. With all the information centralized at the sender, it could make better decisions based on its own knowledge of the situation. For example, if the receiver reports a null X_{recv} in the last RTT, the sender could simply ignore the report if it has not really sent any data.

A partial solution could be implemented with some changes in the protocol. The system could solve some performance issues using a *selective acknowledgment* system. If the receiver reports the sequence numbers used for updating the p reported, the sender can weight this value depending on the real sending rate used. The sender should keep some history of the packets sent, and maybe a sender-based TFRC would be easier to implement. The sender should also use a more adaptive calculation for the feedback packets *timeout*. We have seen in Section 4.2.3 how a $4*RTT$ timeout can be excessively short: the receiver can simply be late when the RTT is very small.

However, the late activation of the sender leads to another problem. We must realize that the sender can always be late sending packets, and that any TFRC implementation will have to do bursty delivery in some situations. For instance, we can imagine a videoconferencing application where video is transmitted using *Digital Video*, *DV*, format. Although this is a compressed format, the sender will generate a fat stream of $144000\frac{bytes}{frame} * 25\frac{frames}{sec} = 3600000\frac{bytes}{sec}$ of data (with *PAL* format, *720x576*). With packets of $1500bytes$, the sender will have to send 2400 packets every second. Obviously, a TFRC implementation in any current equipment will have to send these packets in long sequences, as it can not properly space them.

Perhaps this can be a very particular example, but there is always going to be a hard limit in the granularity of the sending process. For example, even if the sender can generate packets every $500usecs$, there is always going to be situations where the *IPI* will be less than that. Even with fast equipment, the sender will need to send big groups of packets in environments like local area networks.

Paradoxically, these situations will force TFRC to produce data in a quite similar way as a *congestion window* does. Perhaps at a lower scale than TCP, but TFRC can produce *windows* of packets. Not only the lack of packet spacing can be a problem, but the use of an uncontrolled *congestion window* could result in a higher loss event rate. How will a router in the local area network react to *1Mbytes* bursts produced by a TFRC sender? Do we have any control on these long sequences? How do these *windows* change? Maybe TFRC should smoothly change its behavior towards a *window-based* congestion control system in these scenarios.

TFRC presents some open questions, and some of these points affect to any rate-based congestion control algorithm. In the following chapters we will see some limits of TFRC by using our traffic generator. The results of the experiments are covered in Chapter 5, and we will see how it could be integrated in a videoconferencing application in Chapters 6 and 7.

Chapter 5

TFRC Experiments

This chapter shows the experiments performed with the TFRC implementation. After a short overview of the methodology, I will provide the results of the protocol in several environments, discussing the results and proposing some improvements.

5.1 Evaluation Methodology

The evaluation of a congestion control protocol is a difficult task. It requires the development of a detailed set of tests and the elaboration of complex testbeds. TFRC is not an exception, and testing the protocol in a real environments has been a challenge.

Two main components take part in the experiments that can be done with a congestion control system. The first element is the environment. It must be flexible but at the same time realistic. The highest flexibility can be obtained with a a simulator like the *Network Simulator* [63]. However, the resulting environment can be too synthetic and artificial for a *rate-based* congestion control algorithm, in particular if we want to observe the results in a high performance situation. Moreover, TFRC has been deeply tested in this kind of environment, and this work tries to be more focused on results in the real world. As a consequence, the best alternative has been the use of TFRC in real networks, either local or wide area ones. I will detail all these issues in Section 5.1.3

The second main component of the protocol testing is the group of tests that will be performed. They must show that this TFRC implementation follows the specifications, and they must allow this demonstration without any doubt. However, we have to set the limits of these tests in order to see the boundaries of this work. So, it is not the intention of these experiments to probe basic properties of TFRC. The main goal is to verify the conformance with the TFRC defined in the standard, and any further discussion will be focused on the practical aspects that have been observed.

For example, I will prove that the smoothness of this TFRC implementation is the same as the expected from the TFRC specification, but it is not my intention to discuss if the smoothness variation will disturb other TCP flows or if TFRC is really fair. Those issues have been deeply discussed in the TFRC literature, and their results should be assumed as a consequence of the correct implementation of the protocol.

However, it is difficult to establish a clear limit between the basic verification of the expected properties and the study of other topics in practice. Even though some discussion will be opened from the results of these experiments, the main force behind them will be the same: to verify correctness of the implementation.

5.1.1 Loss Models

Losses are the most important factor in determining the behavior of a congestion control algorithm. They are the main signal of congestion when there is no explicit notification mechanism. It is then a fundamental requirement to test the congestion control under different loss models, created either by congestion situations or by artificial transmission failures.

For the creation of congestion, we can produce some background traffic by using some kind of traffic generator. This traffic serves for two different objectives. First, it increases the complexity of the network by creating a less deterministic and more realistic environment. Second, it allows the comparison between TFRC and the other traffic, analyzing how they respond to the same scenario.

The different loss models used in the experiments can be classified in three main groups:

- The first model will be a periodic, deterministic loss model. TFRC connections will be tested in an environment with a low degree of statistical multiplexing (sometimes, just one flow), and losses will be the result of buffers filling up and overflowing. This is a non-realistic scenario, but shows the basic behavior of the congestion control system in steady state.
- The second model will be a loss process in an environment with a higher degree of statistical multiplexing. Several groups of background traffic will be used, generating groups of parallel TCP connections as well as UDP ON/OFF traffic. This

creates a more realistic scenario where buffers will also overflow, but in a not so deterministic fashion.

- A third model will be used where losses are artificially induced in a router. The same conditions found the previous cases will be used, but the new source of losses will stress even more the congestion control systems for both TFRC and the background traffic.

As we have seen in Section 2.2, overflowing network buffers are responsible for the loss rate experienced by connections. So, the models that we have described will modify their intensity depending on the lengths assigned to these buffers. Shorter lengths will increase the drop rate in the routers, and the opposite will happen when the buffers are enlarged.

In general, I will use the rule of thumb of the *bandwidth-delay* product. To set the buffer size in a router as the average round-trip time of the flows multiplied by the total bandwidth available [91]. Although some authors have observed that this value could be reduced [4], I have followed this recommendation for these experiments and, in fact, I have used more buffering in some occasions.

The problem with this rule is manifested when we use a local area environment. In this case, a high bandwidth and short delay results in very short buffer lengths of maybe a couple of packets. This should not be a problem for environments with a high degree of statistical multiplexing, but in a testbed with a lower number of connections we would need perfectly spaced packets with perfectly scheduled computers. In the real world, this can result in large loss event rates. So, there is a limit in the minimum value of the network buffering, and shorter values produce unusable environments. In the following sections I will provide more details on the minimum values used.

After all these considerations, we can see an important drawback in the application of these models. Due to the lack of control in a shared network like the Internet, these scenarios can only be created in a controlled environment (*i.e.*, a local area network). It would not be a good idea to mix controlled and uncontrolled environments, because we could never be sure if, at the end, the result is a realistic environment. In short, these models must be applied only in a local network, and the use of the Internet must be done without external intervention, using it as is.

5.1.2 Metrics

The evaluation of TFRC depends on the metrics used. As the main goal of these experiments is the verification of the TFRC implementation, I will basically use the same group of metrics used in the TFRC literature [30]. This will provide a good measure of the correctness of the protocol, as it can be verified with the sources.

- One of the most important metrics for a congestion control algorithm is the *throughput*. The throughput of a connection at time t measured with a granularity δ can be defined as the amount of data transmitted during a time interval of size δ. Denoting by $d_{t,\delta}$ the amount of data transmitted by a flow in the time interval $[t, t+\delta)$, the throughput can be defined as:

$$B_{t,\delta} = \frac{d_{t,\delta}}{\delta} \tag{5.1}$$

  and the average throughput of a flow over the measure interval $[t_0, t_n]$ can be defined as:

$$B = B_{t_0,(t_n-t_0)} \tag{5.2}$$

- A commonly used metric for the stability is the *Coefficient of Variation*, *CoV*. The *CoV* is defined as the standard deviation of the time series over the mean of the time series. For an experiment run between times t_0 and t_n, with an average throughput B, the CoV_δ can be calculated as:

$$CoV_\delta = \frac{\sqrt{\frac{\delta}{t_n-t_0}\sum_{i=1}^{\frac{t_n-t_0}{\delta}}(B_{(t_0+\delta i),\delta}-B)^2}}{B} \tag{5.3}$$

  and it depends on the measurement time scale. The *CoV* measures the smoothness of a flow, and must be used at different time scales for fully characterizing the connection.

- Another metric used for the analysis of stability is the *throughput ratio*. This ratio can be defined for a flow as:

$$T_{ratio} = \frac{B_i}{B_{i-1}} \tag{5.4}$$

where B_i is the throughput over the i-th time interval. A T_{ratio} of 1 means that the throughput was the same over two adjacent time intervals.

5.1.3 Test Environments

The TFRC implementation has been tested in several environments in order to verify its correctness. In the first group of tests, I have used a local area network consisting of a group of personal computers. Figure 5.1 shows the basic configuration used, using the well-known *"dumbbell"* topology.

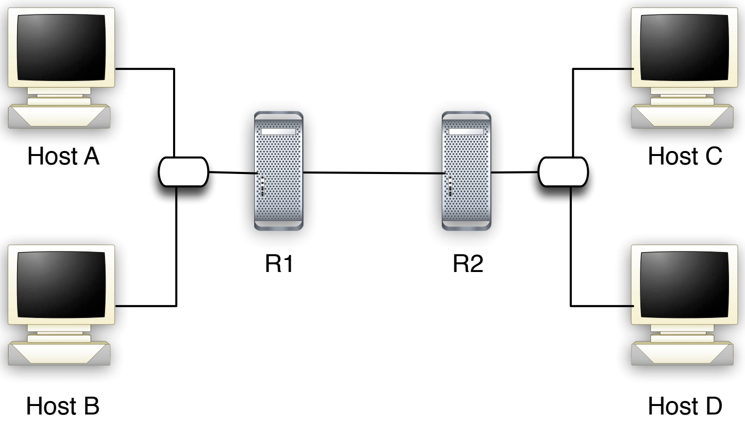


Figure 5.1: Local area configuration.

All systems run Linux (*kernel 2.6*), except *R2* where *FreeBSD 6.0* is installed. Router *R2* has *dummynet* [84] enabled, and it will be used as as the bottleneck for the experiments. By using *dummynet*, we can modify different parameters in the routing code, introducing some propagation delay or losses. Router *R1* will be used for collecting traces of the network traffic using *tcpdump*. These traces will be processed and used for the calculations of the metrics seen in the previous section.

It must be noticed that the propagation delay specified in *dummynet* does not correspond to the real *RTT* experienced by the flows traversing the network. The *RTT* is composed by buffer delays and propagation delays, and buffer delays will be increased when queues utilization increases. So, the delay induced by *dummynet* can be considered as a lower bound of the real value that connections will see.

Experiments have been performed between hosts *A*, *B*, *C* and *D*, generating or receiving TFRC or TCP traffic. TCP traffic was created with *iperf* [90] while TFRC traffic have been produced using the traffic generator described in Section 4.3. The maximum transmission unit, *MTU*, has been set at 1470*bytes* in order to avoid fragmentation. Using this environment, several test sets have been created, with different combinations of delay, bandwidth and loss probability.

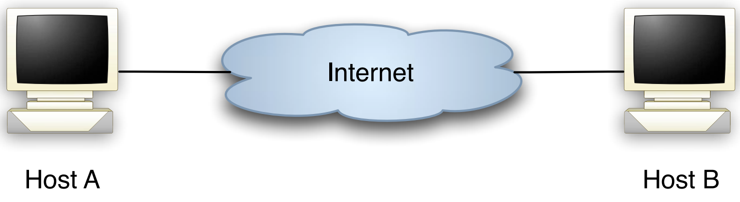


Figure 5.2: Wide-area configuration.

Figure 5.2 shows the configuration used for the wide area experiments. It is a simple setup where one computer acts as the sender and another as the receiver. A broad range of machines have been used for these experiments, trying to verify the behavior of the protocol not only in a real environment like the Internet, but also when it is user on different operating systems. This can provide an insight into the dependencies of TFRC with the underlying system, and the study of some implementation aspects seen in Chapter 4. Table 5.1 lists some of the characteristics of the machines used for these experiments.

Table 5.1: Internet Tests Equipment

Name	*OS*	*Distribution*	*Kernel*	*Speed*	*#CPU*
Alderon	Linux	Fedora Core 3	2.6.5	1GHz	1
Rumi	FreeBSD	FreeBSD 4	4.11	1.80GHz	2
Mediapolku	Linux	Debian Woody	2.2.19	170MHz	1
Curtis	FreeBSD	FreeBSD 6	6.0	3.06GHz	1
Cerrralvo	Linux	Fedora Core 3	2.6.11	2.80GHz	1
Dolgoi	Mac OS X	10.4.4 (Tiger)	8.4.0	1.2GHz	1

The geographic localization of these machines is also diverse. The first two machines, *Alderon* and *Rumi*, are located in Washington D.C., while *Mediapolku* is a PC in Helsinki. The rest of them are personal computers situated in the University of Glasgow, with an average RTT of 110ms (30 *hops*) to *Alderon* and *Rumi*, and 56ms to *Mediapolku* (30 *hops*). The path between *Mediapolku* and the machines in Washington D.C. has 128ms of RTT (36 *hops*).

5.2 Dummynet Results

A local area testbed provides a good opportunity for testing some aspects of the protocol in a controlled environment. We saw in Section 5.1.1 that some loss models can only be set in this kind of environment. In this case, the lack of external traffic becomes something favorable, especially when we want to test the basic behavior of the protocol. In addition, we can include background traffic when we want a more complex scenario.

As I mentioned in Section 5.1, we must focus on some basic properties of a congestion control system like this, and create an adequate set of tests that can demonstrate the correct implementation of the protocol. The group of experiments identified will be focused on the following aspects of the protocol:

- Aggressiveness when starting up and steady-state behavior.
- Fairness with TCP flows.
- Aggressiveness when available bandwidth increases.
- Responsiveness to a new TCP connection.
- Responsiveness to reduced bandwidth.
- Throughput variation and stability.
- Stability under loss.

I will present an overview of the results obtained on these tests in the following sections. Graphs will show typical connections obtained with TFRC (and sometimes with TCP) in these scenarios. However, and in the sake of clarity, only the most explanatory details will be given in the following pages and more exhaustive results should be obtained from Appendix A.

5.2.1 Aggressiveness when starting up and steady-state behavior

The aim of the first experiment is to test the slow-start phase, analyzing the bandwidth utilization and the stability of the TFRC implementation in steady-state.

Table 5.2 shows the set of scenarios used for this test. The values chosen for the RTT and bandwidth represent national, continental and intercontinental connections. The buffer size used in the router has been set at *15Kbytes*, as lower values (corresponding

Table 5.2: Aggressiveness when starting up scenarios

Scenario	I	II	III	IV
RTT (ms)	$3.5ms$	$20ms$	$100ms$	$200ms$
Bandwidth (Kb/s)	8000	3000	600	200

to the *bandwidth delay* product rules) have proved to produce quite unstable sending rates. All the experiments are run for 3 minutes.

Figure 5.3 shows the sending rate observed for the TFRC connection between hosts A and C. The results obtained for Scenario I have been omitted in this graph, but they can be found on page 144 in Section A.1.

After the slow-start phase, TFRC halves the sending rate and tries to reach the bottleneck bandwidth. Considering that X has been limited by the double of this bandwidth, the operation should produce a quite precise result. Figure 5.3 show an accurate sending

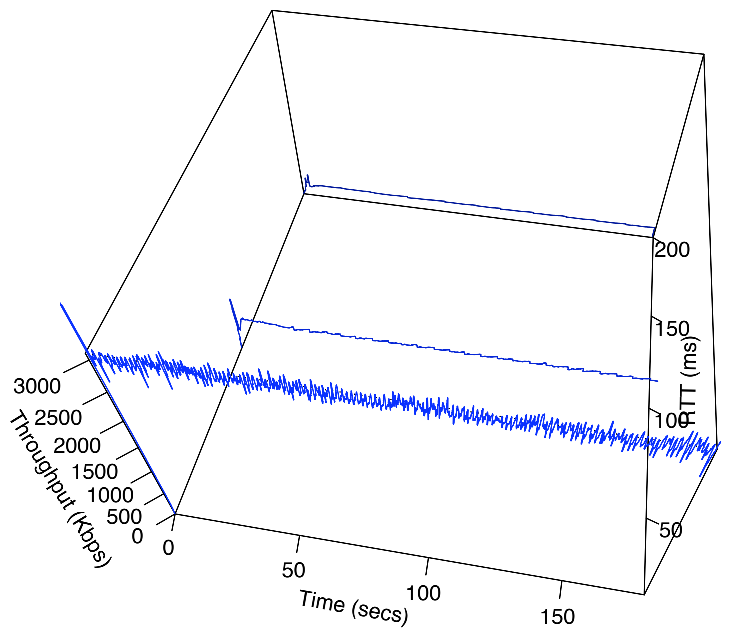


Figure 5.3: TFRC sending rate (X) in steady-state.

rate, near to the bottleneck but with some errors. The reason for these errors was seen in Section 3.2.3: this sending rate is obtained from a synthetic p, resulting in a X that can be slightly under or over the bottleneck.

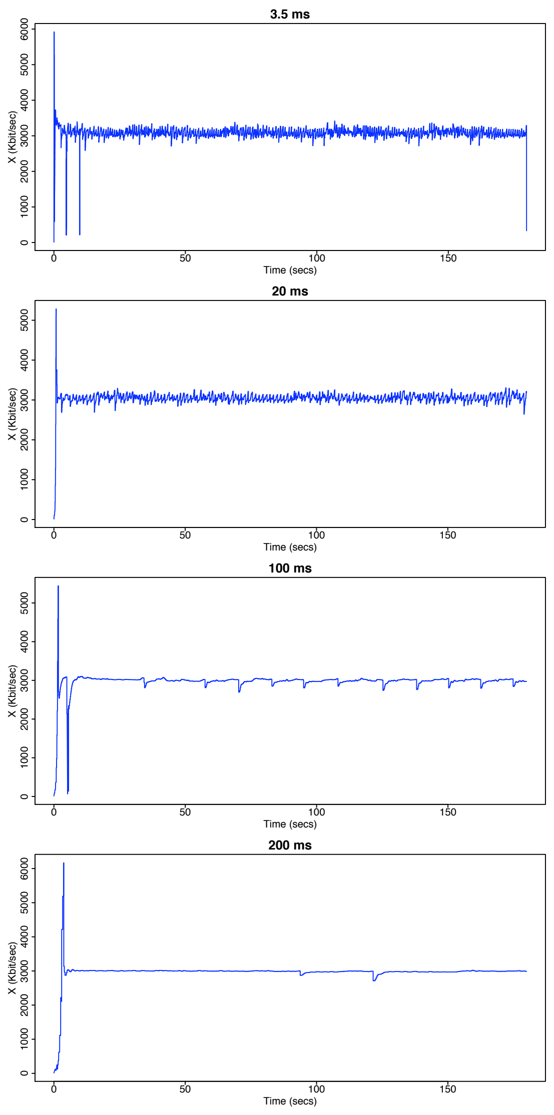
In general, we could say that a sending rate that underestimates the bottleneck would be more preferable. Then the sender could just increase X until this limit is found. Otherwise, an overestimation could result in successive losses and a higher reduction of the sending rate. However, the aggressiveness of this search for bandwidth depends on the current RTT and longer RTTs result in longer search times, as we can see in the *200ms* RTT scenario.

When the bottleneck is found, the protocol reaches a quite stable sending rate. TFRC produces an almost flat throughput in steady-state, though it seems that the best results are obtained with longer RTTs. With short RTTs like *20ms*, the algorithm produces an unstable sending rate that can get even worse for shorter values (see the *3.5ms* case on Figure A.1(a) on page 144).

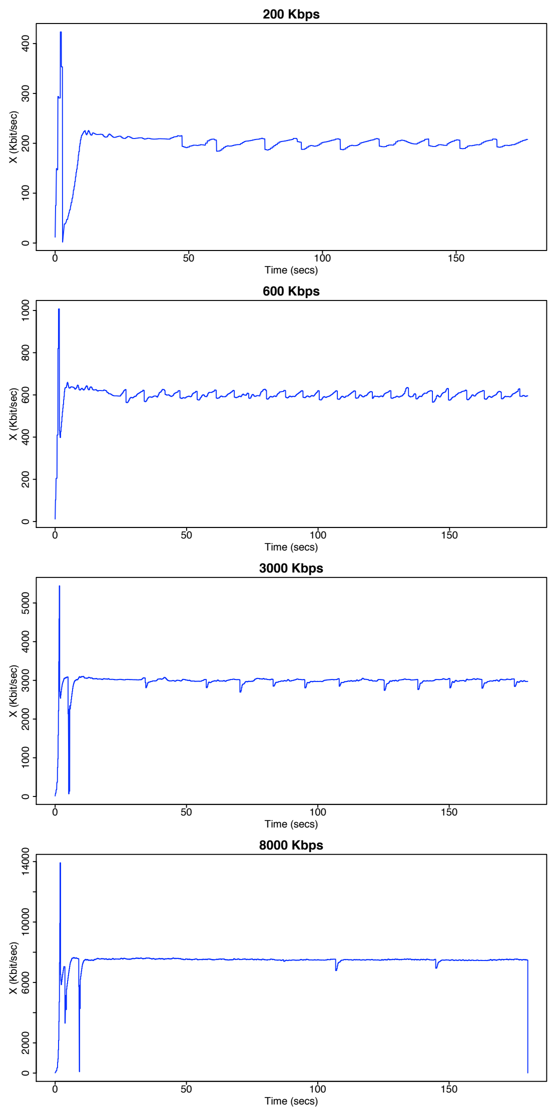
This effect can be seen in Figure 5.4. It shows the behavior of the protocol when the RTT is fixed and the bandwidth is gradually changed, and the opposite case with the bandwidth constant and a varying RTT. In Figure 5.4(a), the bandwidth is fixed at *3000Kbps* and the RTT is gradually changed. We can observe that the stability of the protocol reaches its maximum value for *200ms*, and shorter values seem to increase the sending rate fluctuations. The RTT is the parameter fixed in Figure 5.4(b), while the bandwidth is progressively changed. TFRC exhibits again a high stability, reaching the most stable situation when the bandwidth is increased up to *8000Kbps*.

However, we must notice that the protocol has made some errors on these scenarios. First, there is a bandwidth overestimation after the slow-start in the $3.5ms$ RTT case in Figure 5.4(a) (see more details in Figure A.2(a) in Appendix A). This kind of error is uncommon and, if they do not result in a congestion situation and a loss, they are quickly corrected in the following RTTs. Second, the sending rate seems to suffer some unexpected reductions, as we can observe in Figure A.2. We will see more on these errors in Section 5.3.

Despite these small inaccuracies, the overall behavior of the TFRC implementation seems quite satisfactory. Looking at these graphs, we can conclude that the algorithm shows the best behavior with combinations of high bandwidth and long RTTs, obtaining almost flat sending rates for the two longest pairs: *200ms/3000Kbps* and *100ms/8000Kbps*.



(a) Bandwidth = 3000Kbps



(b) RTT = 100ms

Figure 5.4: Sending rate for static bandwidth and RTT.

Throughput Variation

The stability of the implementation can be further studied with an analysis of the throughput variation. This provides an overview of how the throughput changes at different time scales. For this analysis, we need to calculate the throughput of a connection at different time scales, and apply Equation 5.4 over consecutive time slices.

Figure 5.5 shows the results obtained in the last three scenarios: $20ms$, $100ms$ and $200ms$. It shows the distribution of the throughput variation calculated with the nominal rate calculated by TFRC as well as with the throughput measured in the network. The range of time scales has been adapted from [25], and it is focused in the short term variation: 0.15, 0.25, 1 and 10 seconds. Figure A.3 shows the cumulative distribution with a time scale proportional to the RTT.

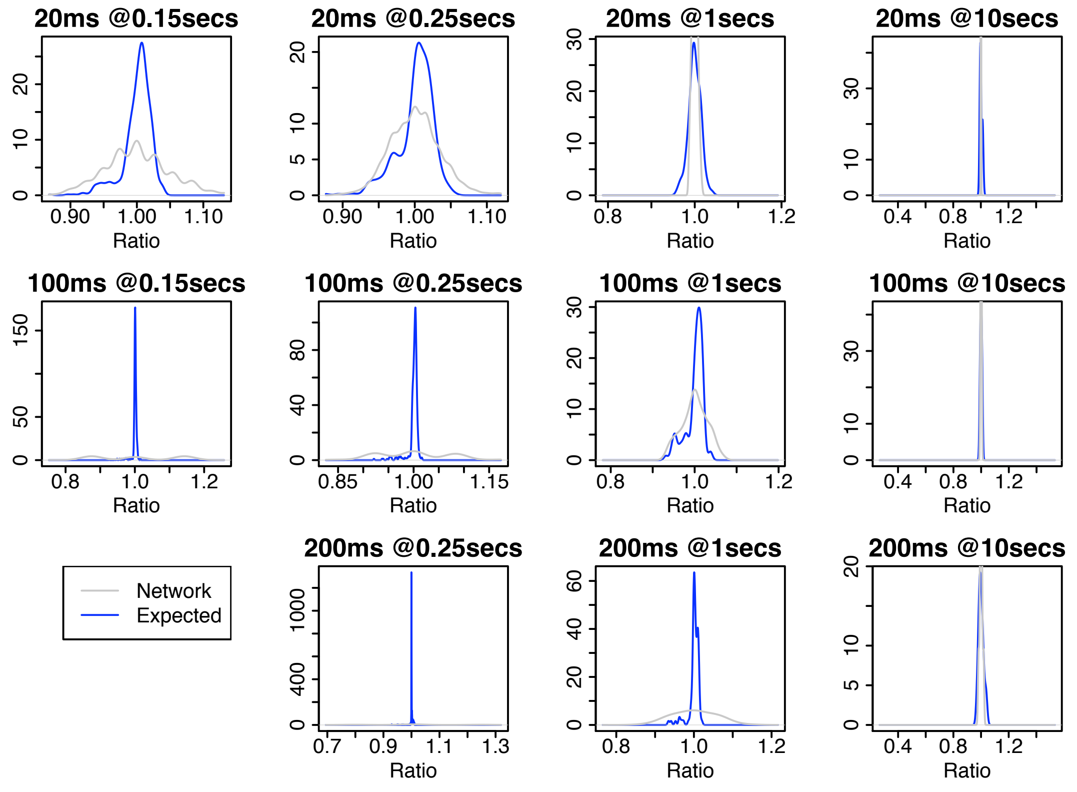


Figure 5.5: Throughput variation for the sending rate and throughput.

It appears from the results shown above that the throughput variation is centered at 1. In other words, the throughput has a tendency to stay constant for all the scenarios. This matches the graphs found in [25].

As it might be expected from a steady-state analysis, the long-term change is almost null. With a $10secs$ time scale, all tests show a throughput ratio highly centered at 1. However, the throughput variation depends on the RTT with shorter time scales. The $20ms$ scenario presents the highest differences in throughput variation. Short time scales manifest a higher variability, a direct result of the instability previously seen.

We must also observe the mismatch between the values calculated from the network or application packets and the TFRC expected equivalent. The changes of network throughput are more widely spread, especially at short time scales. This is an evidence of the difficulty of sending at the TFRC rate, something that we will discuss further in the following sections.

For longer RTTs, the throughput variation tends to be 1 even for short time scales. With $100ms$ or $200ms$ RTTs, most of the throughput changes seem to be at the $1sec$ time scale. The last case exhibits the highest stability, with values centered at 1 for every time scale.

Inter-Packet Interval study

For a better understanding of some TFRC errors, we must focus on some implementation aspects now. In Chapter 4, we discussed several sending strategies and their benefits and drawbacks. In the following paragraphs, we will take a look at the packet level and see how well our solution performs in practice.

Figure 5.6 shows the distribution of the *inter-packet interval*, *IPI*, for the connections in Table 5.2. Three different plots are displayed for each *RTT-bandwidth* combination: the *IPI* calculated by TFRC (*"Expected IPI"*), the *IPI* of the traffic generator (*"Application IPI"*) and the *IPI* calculated after the analysis of the *tcpdump* trace (*"Network IPI"*). The *Application IPI* is calculated using a trace created by the traffic generator, where an entry is added after sending a packet. Plots are obtained over the last 100 seconds of each experiment in order to avoid the effects of the slow-start phase.

Looking at these graphs, the first remarkable thing is the small variation of the expected *IPI*. Knowing that the *IPI* is directly calculated from the sending rate, an *IPI* that tends to stay constant reflects a sending rate that does not change frequently. In these graphs, the *IPI* distribution is highly centered, with a short variation range that depends on the sending rate: for the $3.5ms$ RTT it is $300\mu s$ wide, but with $200ms$ is $3ms$.

We can also notice how difficult is to reach the expected *IPI*, in particular in scenarios with small values. This results in a constant difference between the real *IPI* and the *IPI* established by TFRC. Although both *Application* and *Network IPI*s are quite similar (a detailed look would reveal that the first is a narrower version of the last, reflecting some network spacing), these values have a considerable difference from the *Expected IPI*.

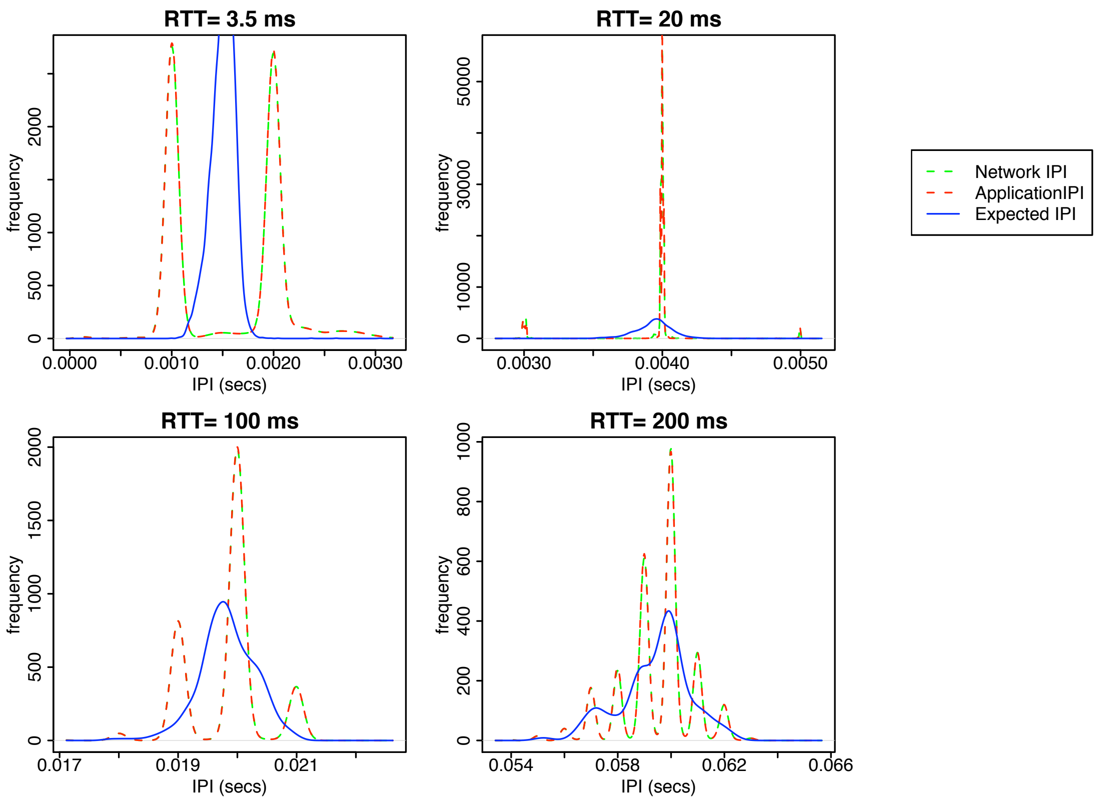


Figure 5.6: *IPI* distribution in steady-state.

In fact, if we look at the *Application* or *Network IPI*s: the sending process seems to have some kind of granularity limit. It seems that packets have a tendency to be sent at $1ms$ boundaries. If we look at the values, we can see how samples are concentrated at the ms points. This effect seems to be set by the application or the operating system, as it can be seen in the *Application IPI* graph.

Obviously, this effect is insignificant for cases like the $100ms$ or $200ms$ RTTs, where the real *IPI* (either the *Network* or *Application*) seems to be an irregular version of the *Expected* shape, producing a sending rate that, on average, will be the same. However, this can be different with shorter *IPI*s. In the $20ms$ scenario, for example, the *Network IPI* is a sharper version of the *Expected* one. The real *IPI* is spread along a wider range of time than the expected *IPI*, and both *IPI*s share the same center and their mean throughput should be equivalent.

However, the $3.5ms$ scenario is different: the sender never sends at the right *IPI*. The

$1ms$ granularity produces in this case a periodic error where the sender spaces packets at roughly 1 or 2 milliseconds. This produces a dual sending rate that, on average, is equivalent to the TFRC rate. Once again, this is not concerning for RTTs above the $1ms$ limit but, if we think about other scenarios, the situation can be different. For instance, a local area network is characterized by shorter RTTs and higher bandwidths. In this environment, with an IPI in the microseconds range, the difference shown in the $3.5ms$ case could be even bigger, oscillating between two different limits: $zero$ or 1 milliseconds. In the first case, the sender would be forced to send packets *back-to-back*, in bursts.

5.2.2 Fairness with TCP flows

The aim of these experiments is to verify the fairness of the TFRC implementation with crossover traffic, in particular with TCP flows. Two different groups of scenarios can be created for this purpose. In the first group we will generate one TFRC and one TCP connection, and we will examine their behavior under different network conditions. In the second group, we will fix the network conditions and we will study the behavior of one TFRC connection with an increasing number of competing TCP flows.

For the first group of tests, we will use the same scenarios seen in Table 5.2. In fact, as these experiments will study the behavior of TFRC when it shares the network with one TCP connection, they can be seen as a different version of Section 5.2.1 where the bandwidth available is determined by other traffic sharing the link. The generation of TCP traffic will be done with the help of *iperf*, creating traffic between hosts B and D while a TFRC flow will be generated between hosts A and C.

The TFRC sending rate adopts very different forms in this group of experiments. In Figure 5.7 (and Figure A.4), we can see not only the throughput obtained with TFRC, but a dashed line that represents the *fair share* level: the bandwidth divided by the total number of connections. We can see big throughput variations and different levels of fairness depending on the RTT and bandwidth used.

For example, TFRC changes the sending rate very slowly when the RTT is set at $200ms$. It takes 25 seconds to reach the fairness level with the other TCP flow. However, the most remarkable detail is that TFRC sends well over this fair share level during the next 100 seconds, and it will not reach this fairness until $t = 160$. In contrast, the $100ms$ RTT and *600Kbit/s*bandwidth scenario presents a different scene. TFRC seems to achieve the best stability and fairness levels at the same time. There are no fluctuations in

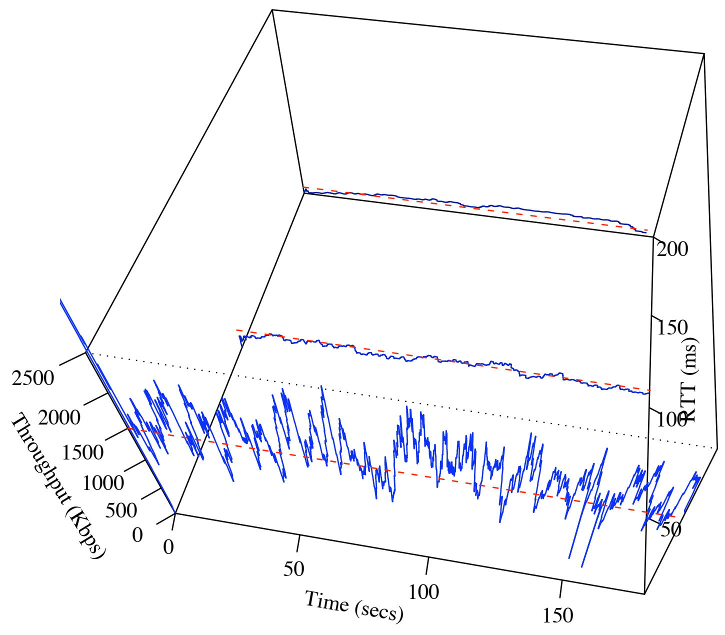


Figure 5.7: TFRC sending rate with one TCP connection.

this case, and the overall fairness is more than sufficient. It takes nearly 15 seconds to arrive at the fairness point, but after this moment the throughput remains stable (see Figure A.4).

For the $20ms/3000Kbit/s$ case (as well as the $3.5ms/8000Kbit/s$ shown in Figure A.4), TFRC gives an acceptable bandwidth share and maintains this fairness during the lifetime of the connection. However, it could give the impression that TFRC does not achieve a short term stability in this case. The sending rate seems to have an irregular shape, giving the impression of being oscillatory and unstable when we look at time scales of 1 or 2 seconds.

Despite this appearance, an analysis of the throughput variation would reveal that TFRC provides a good stability level. As we did in Section 5.2.1, we can apply Equation 5.4 at different time scales in order to analyze the throughput variation. Figure 5.8 shows the distribution of the throughput variation for these scenarios, using a time scale equal to the RTT. It must be noticed that most of the values fall in the same area, near 1, indicating a small throughput change between two successive RTTs.

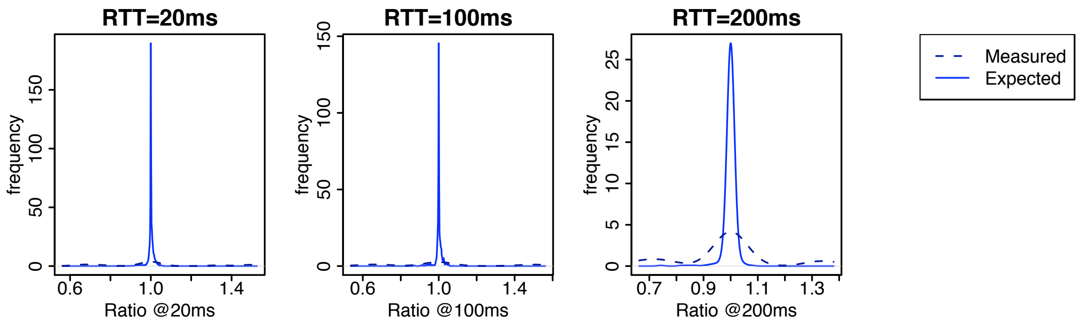


Figure 5.8: Throughput variation for the TFRC sending rate and throughput.

Applying the same analysis to the TCP throughput, the result is quite different. Figure 5.9 shows the throughput variation obtained for the TCP flows created in the previous scenarios. Using the RTT as the time scale for the study, TCP shows higher variability than TFRC. The range of values show that TCP can double or drastically reduce the throughput between two consecutive RTTs.

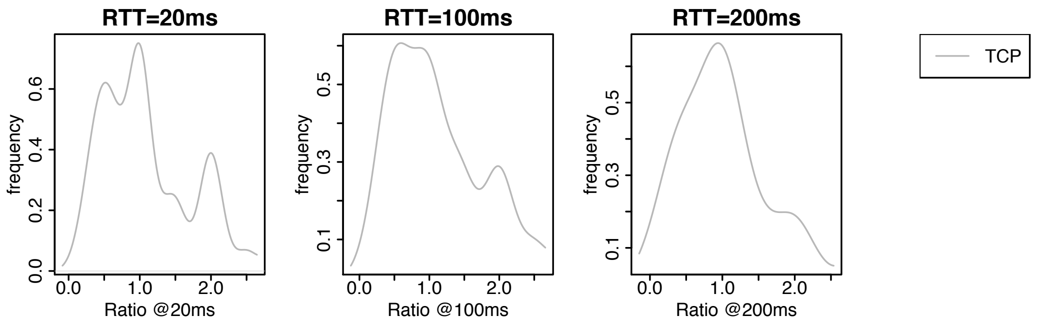


Figure 5.9: Throughput variation for the TCP throughput.

Several TCP connections

The logical evolution of the previous experiment is to test TFRC with several TCP connections. Table 5.3 summarizes the scenarios used. In this case, we will keep the network configuration while we increase the number of TCP flows. The RTT has been fixed at $200ms$, with $1500Kbit/s$ of bandwidth and $37,5Kbytes$ of buffering in the router.

Table 5.3: Throughput of a TFRC flow competing with TCP connections

Scenario	I	II	III	IV
TCP flows	1	2	4	8

TCP traffic has been generated, once again, with *iperf*[1].

We can see the results of these experiments in Figure 5.10 (and in Figure A.5). It represents the throughput obtained for TFRC when it is competing with an increasing number of TCP connections.

When TFRC shares the link with few connections, the sending rate of TFRC after slow-start is quite unpredictable. In our *dumbbell* testbed, the first loss event rate highly depends on the state of the buffer in the router. With just a couple of connections

[1] using the '–parallel #' option

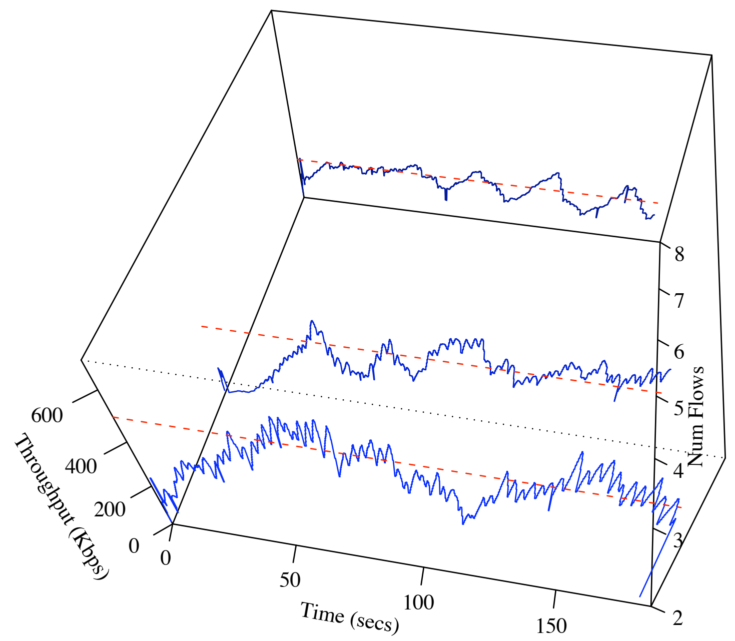


Figure 5.10: Sending rate (X) of one TFRC flow competing with several TCP connections.

filling this buffer, its length will oscillate proportionally to the throughput oscillation of the sources. In contrast, this fluctuation will tend to stabilize when the number of connections is increased.

This effect is illustrated in Figure 5.10. With 2 or 4 TCP flows, TFRC starts with very low sending rates. The state of the buffer has been unfavorable for the connections, and the first loss event rate is quite high in both cases: $p = 0.082893$ with 2 TCP flows, but $p = 0.424502$ with 4 connections.

The best scenario is seen when TFRC shares the link with 8 TCP connections. In this case, the first sending rate calculated after slow-start is not as cautious as the 2 and 4 flows scenarios, and it reaches the fair share level faster. In addition, once the sending rate is stable, TFRC shows a smooth throughput change during the rest of the experiment.

In general, we could say that the long-term behavior of TFRC is satisfactory in any of these situations, although TFRC seems to produce the best results in environment with a high degree of statistical multiplexing. The throughput is not only stable and smooth, but it also reaches a good level of fairness with the TCP connections, and the small errors in the sending rate can be forgiven if we consider the difficulties of some scenarios.

5.2.3 Aggressiveness when available bandwidth increases

The objective of this experiment is to study the behavior of the TFRC implementation when there is an increment in the bandwidth available. In this case, the congestion control algorithm should increase the sending rate until the new limit is found, although we can expect the characteristic slow responsiveness of TFRC.

For this test, a TFRC connection is generated between hosts A and C. The bandwidth in the bottleneck is initially set at *1500Kbit/s*, the RTT is $200ms$, with no additional losses in the network. The bandwidth will be changed at time *t=80secs*, increasing it up to $3000Kbit/s$.

Figure 5.11 shows the sending rate and the loss event rate obtained in this experiment. During the first 80 seconds, TFRC exhibits the behavior shown in previous sections. In contrast, TFRC seems to underestimate the bandwidth available when slow-start finishes, resulting in a long recovery time. Once the bottleneck is found, it shows a high stability until the bandwidth change.

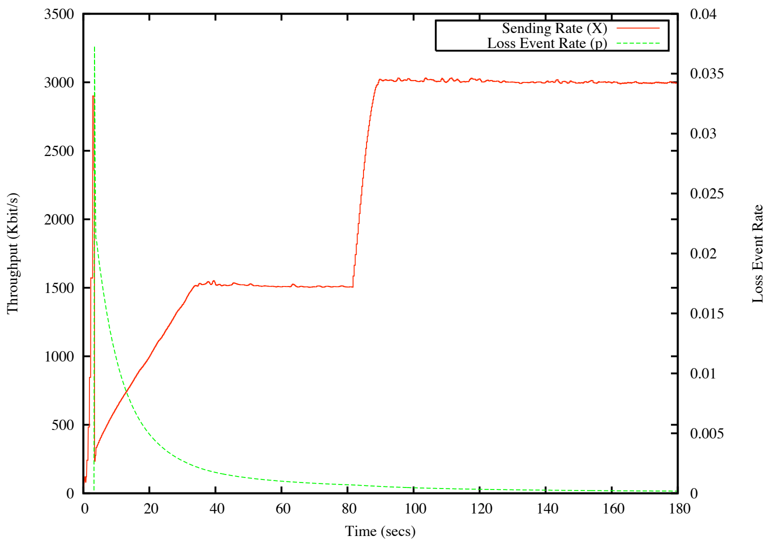


Figure 5.11: TFRC sending rate for bandwidth increment in the router.

After *t=80secs*, TFRC adapts very quickly to the new situation. The sending rate is rapidly increased, looking for any new bandwidth available. When TFRC finds the upper limit, its behavior is quite stable again. We must notice the speed difference between both rate increments: after slow-start and after *t=80secs*. These increments are strongly determined by two parameters: the loss event rate (p) and the RTT.

When the slow-start phase finishes, the first computation of p will be part of the new sending rate calculation. In the following RTTs, and provided that there are no new losses and the RTT stays constant, the increment in the sending rate will also be constant.

However, after *t=80*, the increment in the sending rate is not determined by a change in p. The RTT is the parameter responsible for this abrupt update. In fact, when the bandwidth is changed in the router, the output growth quickly reduces the buffer length. This buffer will empty very fast and this will lead to a rapid reduction of the RTT. The change will be immediately detected by the sender, resulting in a new calculation of the sending rate at the new speed.

Another scenario for studying the increment in bandwidth can be created using another connection. In this case, the bandwidth is initially set to *3000Kbit/s*, but a TCP flow is generated between hosts B and D, sharing the link with the TFRC connection since the beginning. In the same way we did before, the bandwidth will be increased at

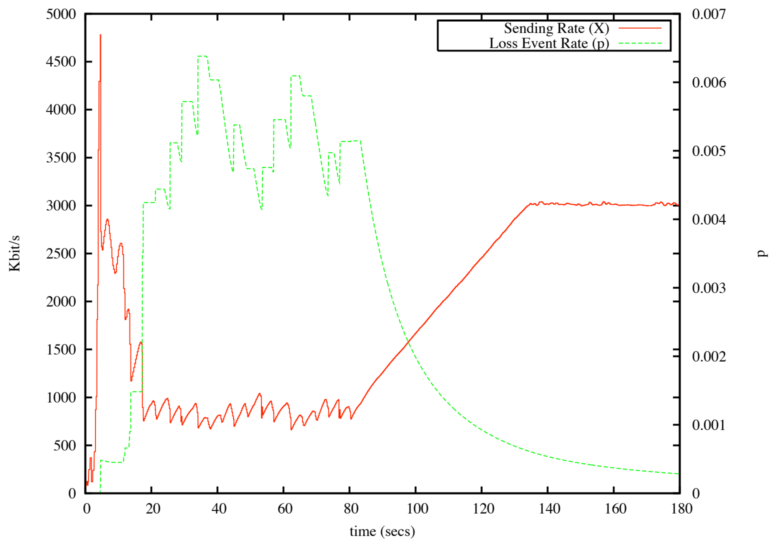


Figure 5.12: Sending rate and loss event rate (with a TCP flow until t=80secs).

t=80secs but, this time, it will be thanks to the interruption of the TCP connection. After *t=80secs* all the bandwidth will be available for the TFRC flow.

Figure 5.12 represents the TFRC sending rate obtained in this scenario. After the first loss is detected, TFRC calculates a higher sending rate than the fair share point. As we saw in Section 5.2.2, this is a typical error in environments with a low degree of statistical multiplexing, where the TFRC connection can reach a too high sending rate during slow-start. New losses and increments in p will reduce the sending rate in the next RTTs. When the TCP connection finishes at *t=80*, the aggressiveness of the algorithm is different to the previous scenario. In this case, the search for new bandwidth is determined by the loss history.

Until *t=80*, the sender has seen a scenario quite similar to steady-state. Periodic losses have resulted in a loss history with virtually uniform loss intervals. With no *history discounting*, the latest interval is weighted in the same way as the previous ones, and the increment applied when looking for bandwidth will be the same than in the most recent history.

In this case, the use of history discounting could have been helpful, and maybe the sender would have accelerated the sending rate after some time. However, the lack of this mechanism cannot be considered a handicap: it only makes the protocol a bit more cautious.

5.2.4 Responsiveness to a new TCP connection

The following experiments have the objective of testing the responsiveness of TFRC when a new TCP connection shares the bottleneck. In this case, the TFRC flow should reduce its sending rate in order to fairly share the bandwidth. However, and as it happened in the previous section, we can expect a slow response in the TFRC reduction.

The testing scenarios are similar to the ones used in Section 5.2.1. These experiments use the same pairs of *RTT-bandwidth* shown in Table 5.2, and they are run for 3 minutes. The only difference will be the addition of a new TCP flow after the TFRC stabilization. At $t = 80secs$, a new TCP connection is created in the network, reducing the bandwidth available for the TFRC flow.

We can observe in Figure 5.13 the TFRC sending rate in this experiment, as well as the fair share point after the change. The case with $3.5ms$ RTT has been skipped in this graph, but it can be found in Section A.3.

With $100ms$ and $200ms$ RTTs, it is easy to see that TFRC not only produces the adequate sending rate, but also achieves a good stability level after the new connection starts. However, these network conditions also show a lower responsiveness level for the protocol. We can appreciate in more detail this relation between RTT and responsiveness in Figure A.6.

With $200ms$ RTT, the connection appears to be slow in its adaptation. It takes more than 40 seconds until TFRC reaches the fair share level and, as we saw in the study of fairness with one TCP connection (Section 5.2.2), TFRC is not fair for the next seconds either. In contrast, a $100ms$ RTT exhibits better responsiveness to new traffic and the throughput is fair and smooth during the rest of the experiment.

Furthermore, the situation gets worse in scenarios with shorter RTTs. There is a short scale oscillation when the RTT is *20ms*. In this case, the throughput exhibits some fluctuations when a new connection is added. Figure 5.14 illustrates this effect in the loss event rate.

During the first 80 seconds, there is a periodic and smooth change in p as a result of the steady-state situation. After $t = 80$, the loss event rate adopts a more variable response when the competing traffic is introduced. As we saw in Section 5.2.2, network buffering plays an important role in this effect, and this is something that we can expect when TFRC shares the link with just one connection in an environment like this.

We can also observe some packet scheduling details if we examine the *IPI* distribution. In Figure 5.15, we can recognize the same effects seen in Figure 5.6, with a tendency to 1*ms* granularity. However, the distribution covers a different range of *IPI*s and with different intensities, as the sending rate has also been more varied.

It is interesting to see a small detail in the 3.5*ms RTT* case: there is the first sign of a *back-to-back* delivery of packets. The distribution function shows a small increment in the 0 that reveals a continuous sending of packets.

5.2.5 Responsiveness to reduced bandwidth

The aim of this experiment is to describe the responsiveness of TFRC when the bandwidth available is decreased by using new connections that share the link. This scenario is an extension of what we saw in Section 5.2.4, but studying the TFRC behavior when the crossover traffic increases. Competing traffic will be created with a different number

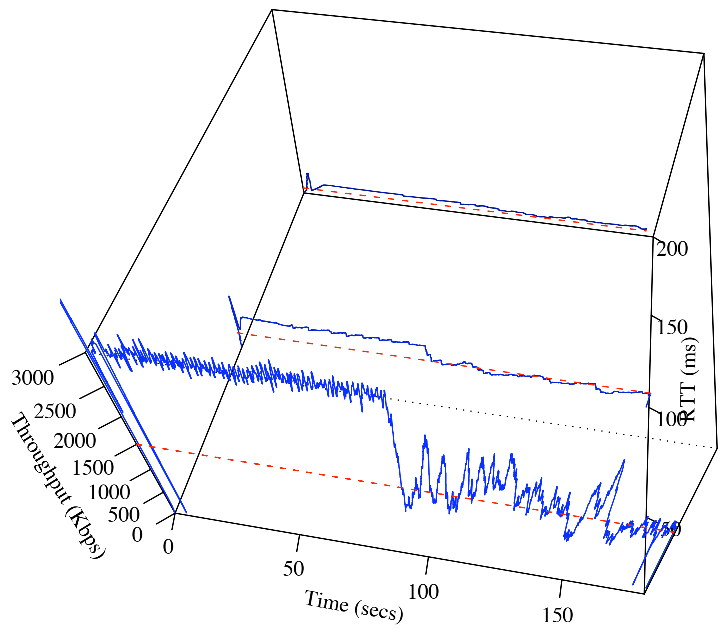


Figure 5.13: TFRC sending rate with a new TCP connection.

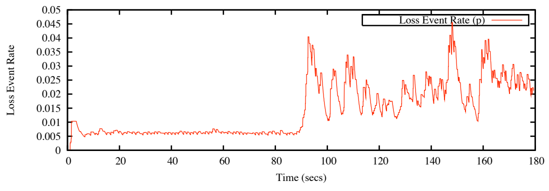


Figure 5.14: Loss event rate for $20ms\ RTT$ and $3000Kbit/s$ bandwidth.

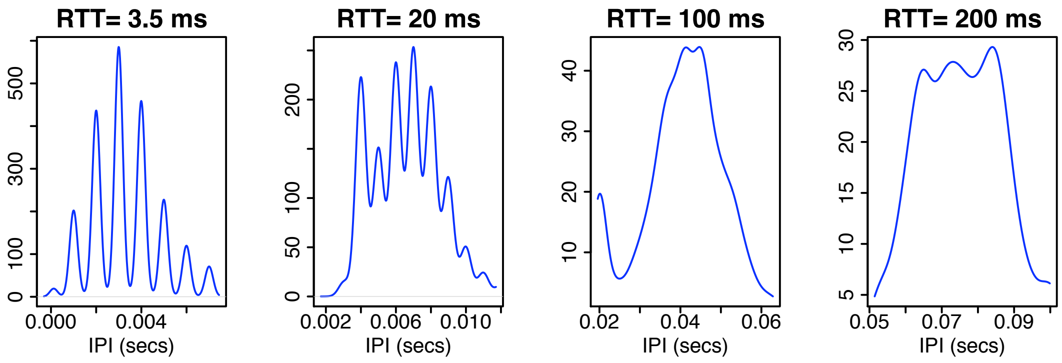


Figure 5.15: Network *IPI* with a new TCP connection.

of TCP connections. As we already know, TCP is a representative and significant example of the traffic found in a real environment, so these tests should show what could be expected of TFRC when it competes with real traffic.

Table 5.4: Responsiveness to new TCP connections scenarios

Scenario	I	II	III	IV
TCP flows	2	4	8	16

The set of scenarios is shown in Table 5.4. All the network parameters stay constant in the four scenarios. The RTT is $200ms$ and the bandwidth is kept at *1500Kbit/s*, with no additional packet loss introduced in the network. The only change will be the number of TCP connection established, although the case with one connection will be skipped as it has been previously covered.

Figure 5.16 shows the results obtained for the first three scenarios (the complete set of results can be seen in Figure A.7). Every experiment starts with a TFRC connection that will remain during the test. At time $t = 80sec$, several TCP connections are started

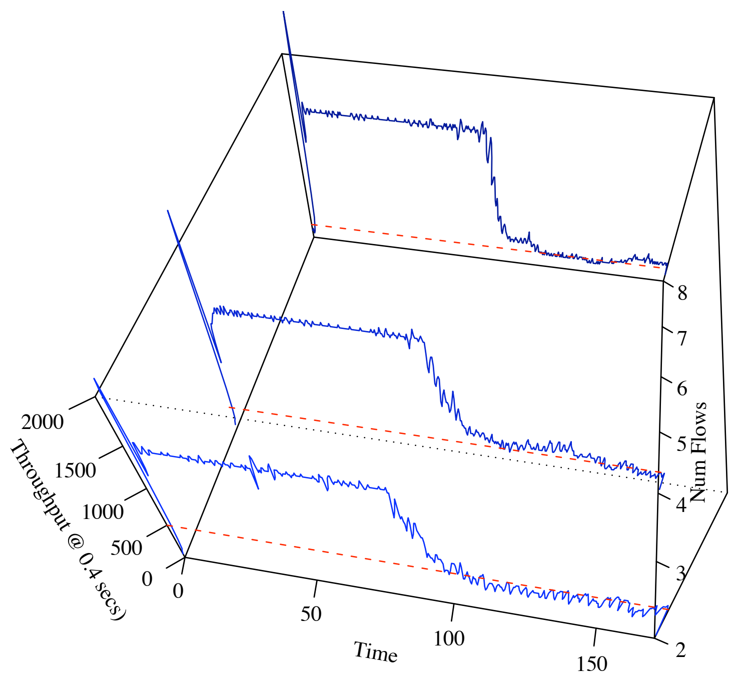


Figure 5.16: TFRC throughput with several new TCP connections.

between hosts C and D. Until then, the throughput of TFRC is once more quite stable, with a good adaptation to the bottleneck after slow-start and an almost flat sending rate in steady-state. The dashed line in the graph represents the expected sending rate after the change, where TFRC would send at the fair share level.

After the TCP traffic starts, TFRC quickly adapts its sending rate. It responds quite fast to the new situation, showing good responsiveness and stability after the change. However, this responsiveness depends on the number of TCP connections: a higher number leads to a more aggressive reduction in the bandwidth available. This comes as a result of a higher congestion level at the router, and more TCP connections increase the congestion very quickly, leading to a rapid increment of the loss event rate.

The number of TCP connection also affects the TFRC fluctuations. As we already saw in Section 5.2.2, TFRC seems to have a more stable throughput in environments with a high degree of statistical multiplexing. In *Scenario III*, eight TCP connections provide a flatter throughput than the equivalent with just two, and it achieves the best stability in the scenario with sixteen connections (in Figure A.7, page 149).

5.2.6 Throughput Variation and Stability

The aim of these experiments is to verify that the stability of the TFRC implementation matches the expected values. The smoothness study should be focused in the short time scale, as it has been demonstrated in [25] that the long term smoothness of traffic tends to follow the same distribution at large time scales (of more than 100 RTTs) regardless of which congestion control system is used. This is the reason why our study will be limited to a 10 seconds time scale.

The main metric used for this study of stability will be the *Coefficient of Variation*, CoV, calculated with Equation 5.3. The scenarios used in these experiments are the same *RTT-bandwidth* pairs used previously, listed in Table 5.5. In these context, we have generated one TFRC connection between hosts A and C and, at a different time, the same pair of hosts have been used for generating a TCP connection, with the help of *iperf*. Both TFRC and TCP flows have been run for 3 minutes. This will enable us to compare the CoV obtained with both congestion control systems.

Table 5.5: Scenarios for Throughput Variation and Stability

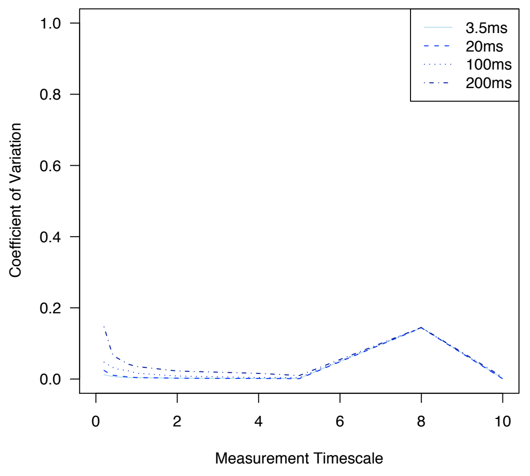
Scenario	I	II	III	IV
RTT (ms)	3.5	20	100	200
Bandwidth (Kb/s)	8000	3000	600	200

We can see the results for a typical TFRC connection in Figure 5.17(a). It displays the CoV, calculated at different time scales[2] for the scenarios of Table 5.5. Figure 5.17(b) shows the same metric but for a TCP connection in the same scenarios.

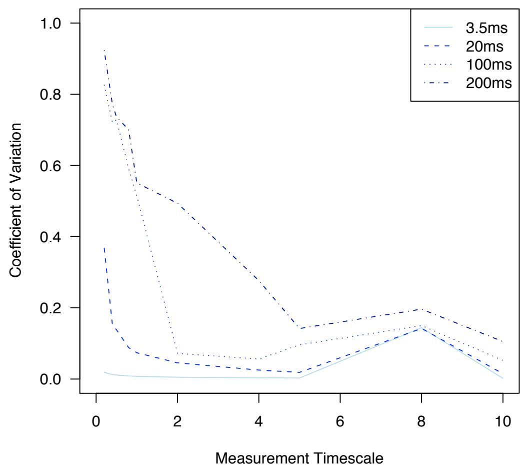
In the TFRC results, we must notice the almost negligible CoV for time scales less than 5 seconds. This represents a high smoothness level for the throughput variation in the short-term. The CoV increases for higher time scales, as the protocol changes the throughput on the long term. In this case, it seems that the TFRC throughput varies when it is observed at a 8$secs$ time scale. A comparison with TCP will show that TFRC has a smoother throughput change.

Figure 5.17(b) shows the results for TCP connections. Depending on the RTT, TCP

[2]The time scales used for the calculation of the CoV have been: 0.2, 0.4, 0.5, 0.8, 1, 2, 4, 5, 8 and 10 seconds



(a) TFRC



(b) TCP

Figure 5.17: Covariance of throughput for TFRC and TCP.

shows quite different behaviors. With a 3.5ms RTT, the CoV is similar to the TFRC results, something reasonable for a study where the minimum time scale is well over the RTT. Longer RTTs result in higher values for the CoV calculated. The best example is found in the 200ms case, where the CoV is higher for short time scales, showing a high short-term variability.

5.2.7 *Stability under loss*

In the next test suite, I will focus on the study of the impact of packet losses. These tests should show how well TFRC works in a real environment like the Internet, where losses have an unpredictable behavior.

Firstly, we will study the change in stability when the loss rate increases, simulating different drop rates by using *dummynet* at router *R2*. In addition, the development of these tests will be done with a constant bandwidth (*1500Kbit/s*) and round-trip time (200ms), and setting the buffer sizes to the bandwidth-delay product (37500$bytes$, 25 *packets*).

Table 5.6 summarizes the scenarios used for these tests. The range of loss percentages starts at 0.1% and grows up to 10%, covering the most typical drop rates that a con-

Table 5.6: Stability under loss scenarios

Scenario	I	II	III	IV
Packet Drop	0.1%	1%	2%	10%

nection experiences in the Internet [72]. Nevertheless, we must remember that the main loss source is not transmission failures but congestion situations at routers, so the total rate seen by a flow will be higher than these values.

Figure 5.18 shows the throughput obtained for a 0.1%, 1% and 2% packet drop rates. Due to the low throughput obtained for the 10% rate, this plot (as well as more detailed results) can be seen in Section A.5 in the Appendix A.

A 0.1% loss rate has a negligible effect in the throughput obtained with TFRC. The only change is some sporadic reductions in the sending rate, but the protocol exhibits a high stability degree, and the overall behavior remains almost unaffected at this level.

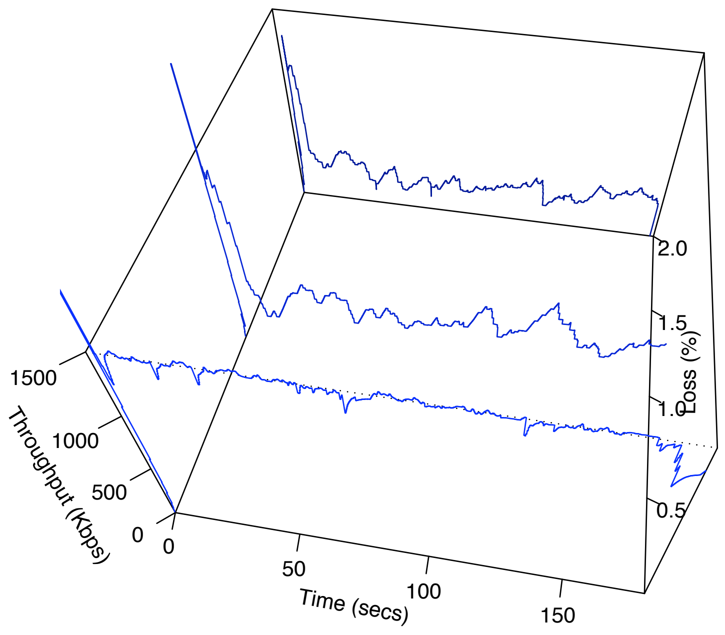


Figure 5.18: TFRC sending rate under loss.

In contrast, TFRC seems to be much more conservative with 1% and 2% loss rates, where the sending rate falls and stays well under the bottleneck capacity.

Coefficient of Variation

Figure 5.19 shows the coefficient of variation corresponding to the connections of Table 5.6, calculated using Equation 5.3 with varying packet losses. Compared with the *CoV* that we saw in Figure 5.17(a), it can be seen that the 0.1% scenario is almost identical, showing an nearly null variation. Although it is slightly higher than its counterpart in Section 5.2.6, the *CoV* shows again a good stability level for TFRC. However, there is a significant increment for the other packet drop rates.

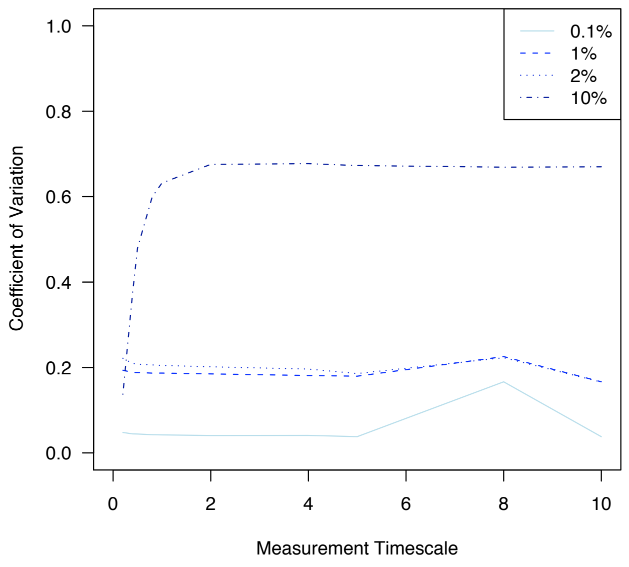


Figure 5.19: Covariance of throughput under loss.

With 1% and 2% drop rates, TFRC shows a higher *CoV* than the previous scenario. This matches the higher variability seen in Figure 5.18, where the sending rate changes more frequently with these packet drop rates. However, both values show an almost identical shape for every time scale and the difference between then is almost negligible.

The 10% rate scenario presents a substantial increase, indicating more frequent throughput variations. This was expected at this drop rate, where TFRC changes the sending rate on the long term. In contrast, short time scales exhibit a low coefficient of variation, indicating that the protocol still keeps the throughput smoothness on the short-term.

Comparison of TFRC and TCP throughput

We can also study the behavior of one TFRC connection when it shares the link with other TCP flow, in order to compare the behavior of both protocols under the same drop rate. We must realize that this is not an environment with high level of statistical multiplexing and, as we saw in Section 5.2.2, we can expect some variability and throughput oscillation that would probably not appear with more complex background traffic.

Figure 5.20 shows the throughput of a TFRC and a TCP flow sharing the same link. Bandwidth and RTT have not changed, using the same values used in this Section, and with the drop rates shown in Table 5.6. Both connection are started at the same time and run during the whole experiment. More details can be obtained from Figure A.9 in the Appendix A.

Looking at the TCP throughput, it exhibits the typical rapid changes of TCP. The TFRC sending rate remains stable on the long term, with some fluctuations on short time scales. In fact, TFRC reductions are preceded by TCP increments, represented by graph peaks. This reveals once again the dependency between flows in environments with a low degree of statistical multiplexing like this.

Furthermore, TFRC seems to achieve higher stability when the drop rate increases. If we compare the throughput obtained with TCP in the 2% case, we can see a smooth sending rate for TFRC while TCP produces a lower output. For a 10% rate (see Figure A.9) the difference between both protocols is even wider, with TFRC producing some throughput while TCP stays almost inactive.

Bursty traffic

A different scenario can be created when we include some bursty background traffic. The objective of this scenario is to test TFRC in a very *stressing* environment, with aggressive and unfriendly traffic. This should create a completely unpredictable loss model, far from steady-state losses or the artificial drop rate seen earlier. In addition, this kind of traffic could emulate to some extent the characteristics of competing web

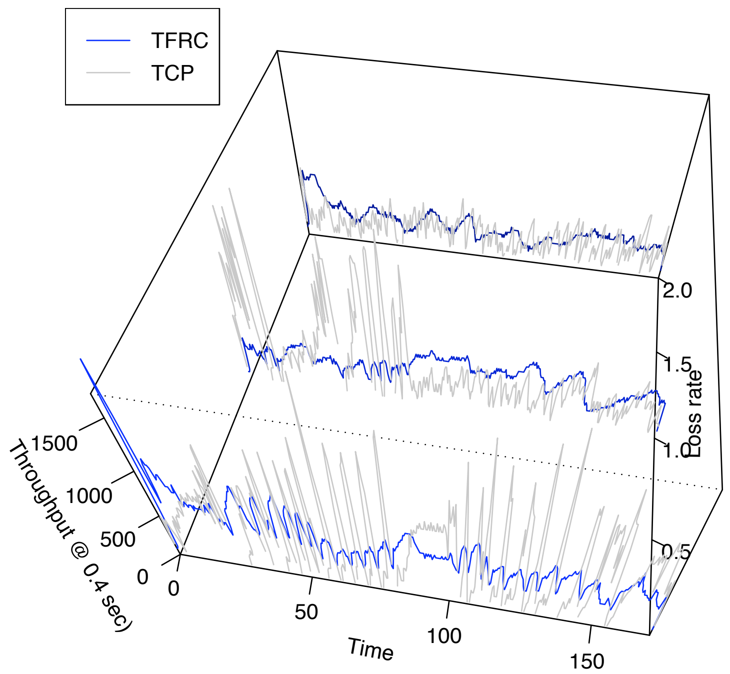


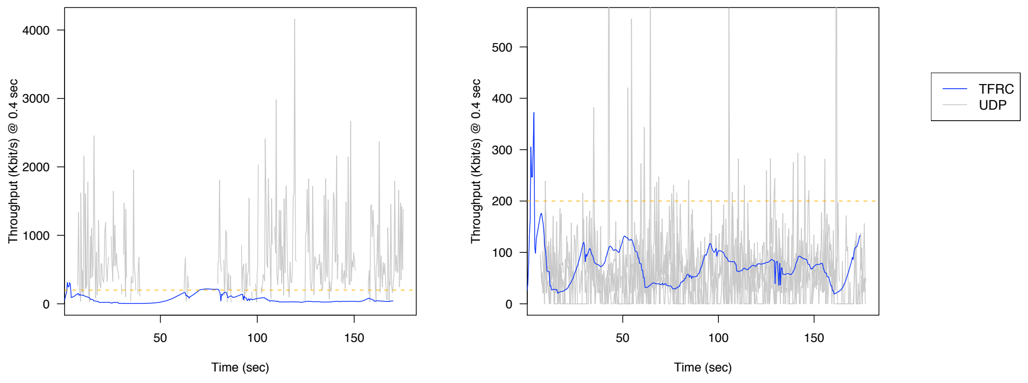
Figure 5.20: TFRC and TCP throughput under loss.

traffic. Web-like traffic can be created by using several ON/OFF UDP sources whose times are obtained from a *Pareto* distribution [19].

In this experiment, the network context will be $200ms$ RTT and $200Kbit/s$ bandwidth for a link that will be shared between a TFRC connection and some UDP flows. The UDP traffic will be generated with a traffic generation utility, *D-ITG* [20].

Figure 5.21(a) shows a TFRC connection sharing the link with a highly aggressive and unfriendly UDP flow, where the dashed line represents the bottleneck bandwidth. This flow has been created using a *Pareto* distribution for the packet inter-departure times (shape $\alpha = 1.3$ [3] and scale $20ms$) and packet sizes (shape $\alpha = 1$ and scale $1000bytes$). This connection should consume, on average, $232.4Kbit/s$ of bandwidth (at $28.3pcks/s$), well over the bottleneck of the link, but the variability of this traffic will result in strong data bursts along with some inactivity periods.

[3] A shape parameter $\alpha \leqslant 2$ means that the distribution has infinite variance



(a) 1 TFRC and 1 UDP (b) 1 TFRC and 4 UDP

Figure 5.21: TFRC sending rate (X) and UDP throughput with bursty traffic.

In this scenario, the UDP traffic is characterized by sudden increments in the throughput, producing far more data than the bandwidth available. This results in abrupt changes of the loss event rate (this can be observed in Figure A.10), but this does not lead to strong sending rate oscillation. Despite the fact that the UDP traffic can reach near $4000 Kbit/s$ at some moments, TFRC seems to smoothly reduce the sending rate in these cases and keeps a stable throughput during the rest of the experiment. However, this aggressiveness of the competing traffic forces TFRC to reduce the sending rate to near *zero* at several points in time.

A different scenario is shown in Figure 5.21(b), where the TFRC flow shares the link with 4 UDP flows. These connections are generated in the same way as in the previous scenario, using a *Pareto* distribution for inter-departure times (shape 1.5 and scale $100ms$) and packet sizes (shape 2 and scale $1000bytes$). In contrast, the UDP average throughput should not exceed the bandwidth available, with $49.3Kbit/s$ (at $6pcks/s$) per connection.

Background traffic does not seem so aggressive in this case. Although UDP flows are still unfriendly and use more bandwidth than the fair share, the higher variability of this traffic results in a more relaxed environment. TFRC suffers more regular losses but, as the loss event rate is lower (see Figure A.10(b)), they seem less intense than the 1 UDP flow case. In contrast with the previous scenario, sending rate reductions never lead to very low throughput. Even though the protocol experiences some fluctuation, it has a long period of more than $20secs$.

5.3 Internet Experiments

The following experiments present TFRC in the real world. They have been run using the environment described in Section 5.1.3. Table 5.1 lists the names and main characteristics of the machines used in these tests.

These experiments try to study the behavior of the protocol from two different points of view. First, TFRC is a congestion control protocol designed to be used in real networks, so we need something more than the *dummynet* environments: it must be tested in real situations with real traffic. Testing the protocol in a real environment like the Internet can show its value, good points and flaws. Second, these experiments have been performed using a wide range of machines, with different operating systems, kernel versions and speeds. This reveals some of the dependencies of TFRC with the host environment, bringing to light the details that make TFRC a difficult to implement congestion control mechanism.

5.3.1 Slow-start problems

Figure 5.22 presents a connection between *alderon* (as the sender) and *curtis* (the receiver). These are two fast machines located, respectively, in Glasgow and Washington D.C. The RTT measured in this experiment was $140ms$.

From Figure 5.22(a), we can observe an interesting effect: we could think that the sending rate after the first loss in not satisfactory. TFRC does not seem to reduce the sending rate to the right level, and this could be the cause of the subsequent chain of losses and reductions. However, a detailed view of the slow-start phase would reveal some interesting points. Figure 5.22(b) shows the three main elements in this stage: the sending rate (X), the sending rate at the receiver (X_{recv}) and the loss event rate (p).

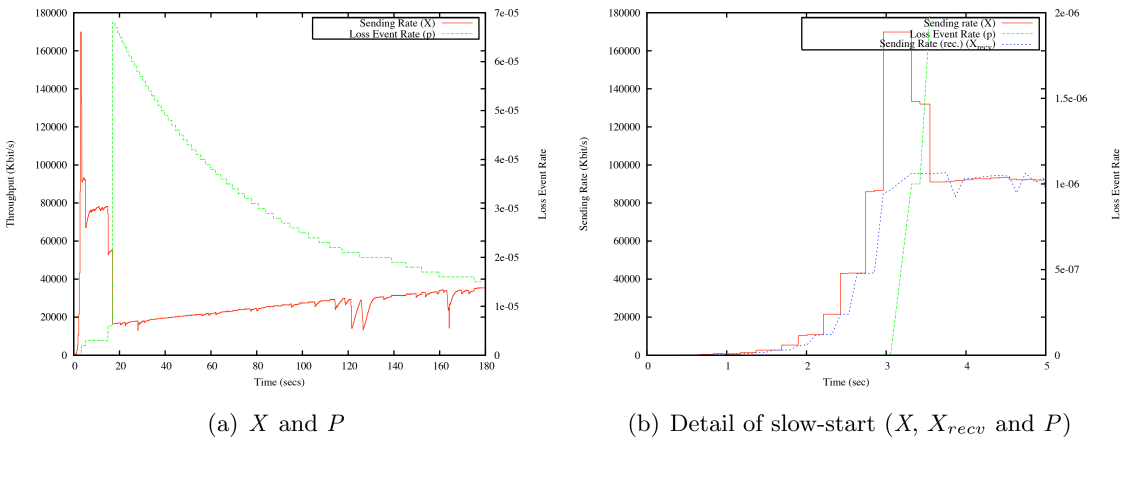
The slow-start stage commences with a very slow increase of the sending rate due to the long RTT. During this time, the rate observed at the receiver,X_{recv}, corresponds to a *delayed* version of X. The sudden increment in the value of p indicates the end of slow-start and the moment where the new sending rate must be calculated. However, if we look at the new X obtained, it is surprising to see that it matches quite well the last X_{recv} observed by the receiver: the new X calculated is right.

Then, the reason for the *cascade effect* in the sending rate reductions must be found in a different place. From our experiments, we think that this is the result of the network buffering and congestion variability. TFRC seems to behave as it should, reacting to network congestion in the right way, but the network buffering results in different loss rates at different times. After a short time producing more than $90000 Kbit/s$, the network will drop a couple of packets and the slow-start phase will finish[4]. A reduction to $80000 Kbit/s$ will not be enough after some time, and our connection will experience a more severe network congestion and the network will drop longer number of packets.

We have observed this effect in numerous occasions: the sender seems to obtain a good bandwidth for some time but it is quickly followed by strong losses that force a drastic reduction. It is remarkable that the network can transmit at $80000 Kbit/s$ for almost $10 secs$, but this leaves TFRC with the wrong *impression* of the bandwidth available and leads to following reductions.

However, we must not forget that the protocol exhibits very good responsiveness when the congestion level increases. TFRC quickly reduces the sending rate, and the new throughput exhibits the same characteristics than we have seen in the previous sections: smoothness and stability. We can observe this effect in the in Figure 5.23. It presents another experiment between *curtis* and *alderon*, in this case with *curtis* acting as the sender.

[4]The bottleneck link on the path is $100 Mbps$ on the LAN.



(a) X and P

(b) Detail of slow-start (X, X_{recv} and P)

Figure 5.22: TFRC connection between *alderon* and *curtis*: sending rate at the sender (X) and receiver (X_{recv}) and loss event rate (p)

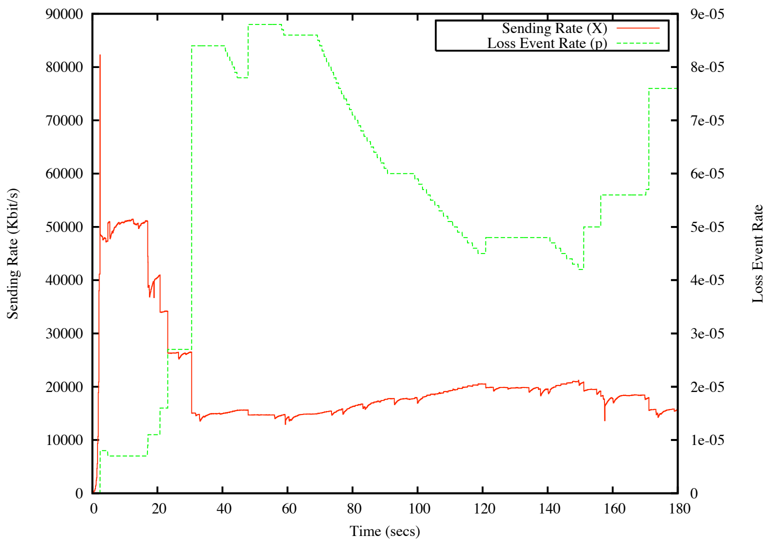


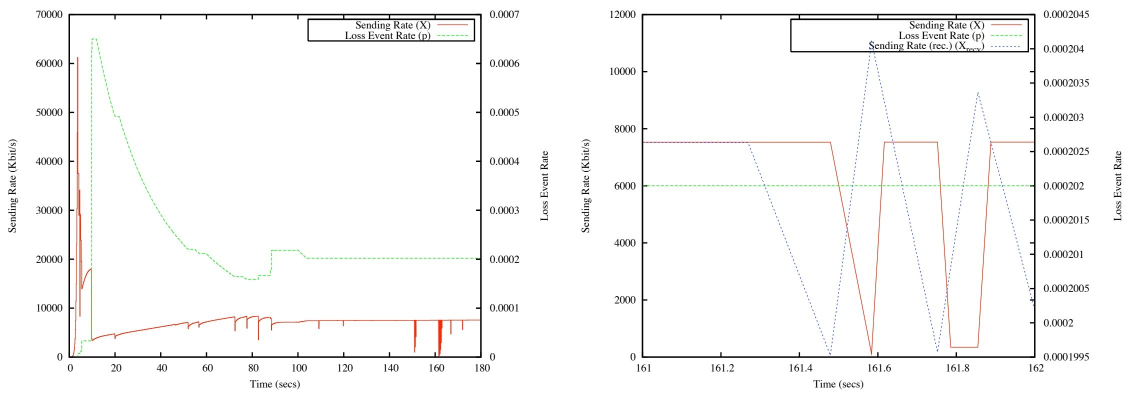
Figure 5.23: Sending rate (X) and loss event rate (p) between *curtis* and *alderon*.

This graph shows the same situation described previously, with the *cascade effect* after slow-start. However, the sending rate reaches an stabilization point at $20000 Kbit/s$ and stays at this level for a long time. TFRC exhibits a remarkable stability in this scenario, with some variations long-term variations.

5.3.2 OS and hardware dependencies

In the previous experiments, the speed of both computers has been a plus for the accuracy of the sending process, in the sender as well as in the receiver. We will now introduce a slower machine, *mediapolku*, in order to test the dependencies that could exist with the hardware. In Figure 5.24(a), it can be seen a TFRC connection between *alderon* and *mediapolku*, acting as sender and receiver respectively. The RTT between both machines is near $120ms$.

We can observe the same effects seen between *alderon* and *curtis*, with an abrupt sending rate reduction after slow-start. However, there is a difference in the magnitude, with no more than $10000 Kbit/s$ of bandwidth available when the system stabilizes. The sending rate is quite smooth in this scenario and, although TFRC looks for more bandwidth after time $t = 10$, some losses slow down this search after $t = 80$. During the rest of the experiment, the sending rate is quite constant, with a modest loss event rate.



(a) X and P (b) Detail for $t = (161, 162)$

Figure 5.24: TFRC connection between *alderon* and *mediapolku*: sending rate (X) and loss event rate (P).

However, the sending rate seems to suffer some unexpected falls. These sudden changes are the result of the *OS* and hardware dependencies. The following sequence of events can occur in a TFRC receiver:

1. The receiver generates a feedback report at t_1, and it starts a `select()` operation while it waits for the next data packet.

2. In the meantime, the OS can preempt the program. The TFRC receiver losses the CPU for a long time. In our case, this is near to $200ms$.

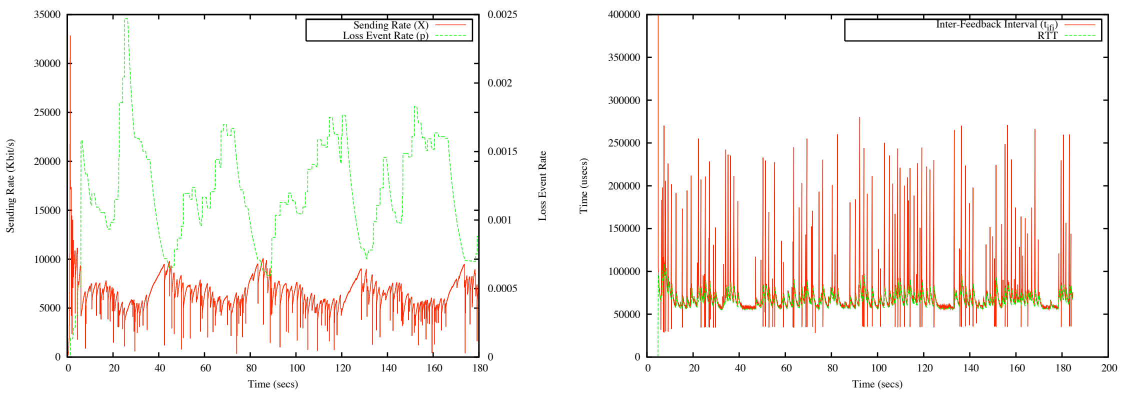
3. When the receiver is resumed at t_2, it computes the time difference, t_{ifi}, as $t_2 - t_1$.

4. The receiver realizes that t_{ifi} is far bigger than the current RTT, and it has been silent for more than $200ms$: it must quickly send a new feedback report. As no new packets are processed after the restart[5], it calculates the new X_{recv} using the number of packets received since t_1.

5. The next feedback report will include the packets that have not been computed in step 4. Since the only effect of X_{recv} is to limit the sending rate, it will not modify the new X.

In this sequence, the receiver can process some packets before losing the CPU and starting step 2. In general, more packets processed before losing the CPU will mean a

[5]This would require another `select()` operation in order to process the packets in the *OS* queue, something that could delay even more the generation of the feedback packet.



(a) X and P (b) RTT and Receiver delay

Figure 5.25: TFRC connection between *curtis* and *mediapolku*: sending rate (X), loss event rate (P), RTT and receiver delay (t_{ifi}).

more accurate X_{recv} calculation. Otherwise, the new X_{recv} will be lower than expected (or null) and the sender will be forced to drastically reduce the new sending rate.

For our connection, we can see one of these errors in Figure 5.24(b). It magnifies the TFRC connection for the $t = (161, 162)$ interval, presenting the sending rate at both endpoints as well as the loss event rate. For a constant value of p, there is an abrupt reduction in the value of X_{recv}. After generating a report, the TFRC receiver processes one packet before a $200ms$ sleep operation (the sequence of packets can be seen in Figure A.11 in Section A.6). The new X_{recv} reported will be equal to $7007 bytes/sec$, and the sender will have to reduce the sending rate at the double of this value.

We can further develop this scenario with a more complex case. Figure 5.25 presents a TFRC connection between *curtis* (sender) and *mediapolku* (receiver). In this case, the loss event rate is higher, but the most important change is the shorter RTT.

The main difference with Figure 5.24(a) is the frequency of the sending rate oscillation. It has the same characteristics of the previous problem: short and sharp falls in the sending rate. There are two factor that take part in this effect.

Firstly, we saw in Chapter 3 that the TFRC receiver must send a feedback packet when it detects a loss. The receiver does this before the next RTT boundary, and the X_{recv} reported will be calculated over a shorter time period. In the best case, it will be a low value but, if the period is really short, the X_{recv} calculated can be unpredictable.

However, it seems that the t_{ifi} will never be really short. In fact, there seems to be a limit in the minimum frequency for feedback reports. We can observe in Figure 5.25(b) that, even when the receiver must early deliver a feedback report, it will never be generated with an t_{ifi} less than $30ms$. We studied this effect in Chapter 4, when we saw that the latency of the `select()` operation could lead to a delay in the feedback process.

The other factor that creates the sending rate oscillation comes from the main difference between this scenario and the one shown in Figure 5.24(a): the lower *round-trip time*. The RTT between these two machines is in the $50-60ms$ range, forcing the receiver to be more accurate sending feedback reports.

Figure 5.25(b) shows the current RTT and the time elapsed between two consecutive feedback packets (t_{ifi}). In an ideal case, both values should match. With a $50ms$ RTT, *mediapolku* is forced to send reports quite more frequently, and there is a periodic error in this process. More frequent `select()`s makes the TFRC receiver the perfect candidate for preemption, and the TFRC receiver then suffers from a periodic overrun of the sleeping time: instead of sending feedback every $50ms$, it seems to sleep for up to $200ms$.

These errors are rather high for this RTT. The TFRC receiver is frequently interrupted and, quite often, it calculates a wrong value for X_{recv}. The result will be a proportional oscillation in the X calculated at the sender.

5.4 Conclusions and Future Work

We have seen in this chapter the main results obtained with our TFRC implementation. These findings match the behavior expected from a TFRC implementation as it has been presented in the literature. The protocol exhibits a smooth sending rate variation, with stability and fairness.

However, we have also seen some problems with TFRC in the real world. As we saw in Chapter 4, the protocol has some dependencies with the host operating system, and environments with short RTTs and high bandwidths can lead to some sending rate instabilities. The protocol could need some mechanisms for balancing these effects, and more support from the operating system could definitively alleviate the situation, but the way it could be implemented it is still a problem that needs further investigation.

In the next chapter I will introduce UltraGrid, the application where TFRC has been used, followed by the results obtained in Chapter 7.

Chapter 6

Congestion Control for Videoconference Applications

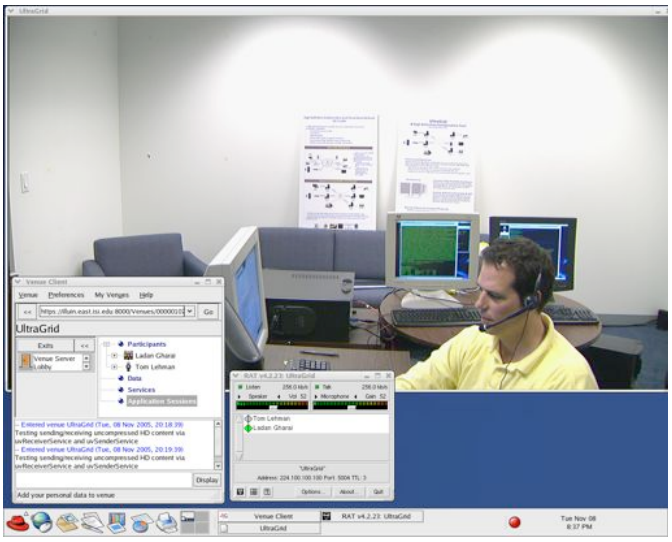
In this chapter I will present UltraGrid, the videoconferencing application where TFRC has been integrated. After a short overview of the features and architecture, I will focus on the parts affected by a congestion control system like TFRC, describing the design chosen and discussing some implementation issues.

6.1 UltraGrid Videoconference system

UltraGrid [74] is a high-definition interactive video conferencing system developed for real-time environments, providing low latencies and high data rates. It is a highly modular system where codecs, transport protocols or devices can be easily integrated.

UltraGrid supports some important codecs for video encoding and compression. In addition to uncompressed *High Definition*, *HD*, video [35], UltraGrid now supports several codecs including standard *Digital Video*, *DV*, [17] and *Motion-JPEG*. It can also be used as a general purpose HD data transmission system, encoding high-definition video into RTP/UDP/IP packets (using the RTP profile defined in [34]).

UltraGrid is focused in the video experience of a videoconference, leaving the audio transmission to an external system. For example, UltraGrid can be used in combination with AccessGrid [2], allowing a complete high definition video conferencing system. We can see in Figure 6.1 the result, where the audio tool is *RAT* [42].



-- Entered venue UltraGrid (Tue, 08 Nov 2005, 20:18:39)
Testing sending/receiving uncompressed HD content via uvReceiverService and uvSenderService
-- Entered venue UltraGrid (Tue, 08 Nov 2005, 20:19:39)
Testing sending/receiving uncompressed HD content via uvReceiverService and uvSenderService

Address: 224.100.100.100 Port: 5004 TTL: 3

Figure 6.1: Screenshot of UltraGrid running on Linux.

UltraGrid is not only an ideal platform for high performance video, but it is also the perfect framework for testing new media transmission techniques. The application can be easily modified and any component could be smoothly replaced or new elements could be included. This modular design enables the addition of a congestion control system with no problems and makes UltraGrid a good environment for experimenting with TFRC.

As we will see in the following sections, TFRC has been integrated in the UltraGrid and used for the modulation of the sending rate of the system. This application level solution contrasts with other alternatives like DCCP but, as current implementations are at the kernel level, this would reduce the portability of our application. The integration of TFRC with the existing RTP/UDP subsystem would be more appropriate, enabling an easier development and testing of an experimental congestion control algorithm.

6.2 UltraGrid Design

The main goals in the design of UltraGrid have been the performance and the modularity. This has led to an architecture where all the components are clearly separated but with strong and efficient interfaces. We will see how these elements interact and the main data flows in the system.

Figure 6.2 presents a high-level overview of the UltraGrid architecture. We can see in this diagram the main components of the the system, and how they are distributed among four different threads. This high level of concurrency reduces the subsystems coupling, reducing the total latency and increasing the interactivity.

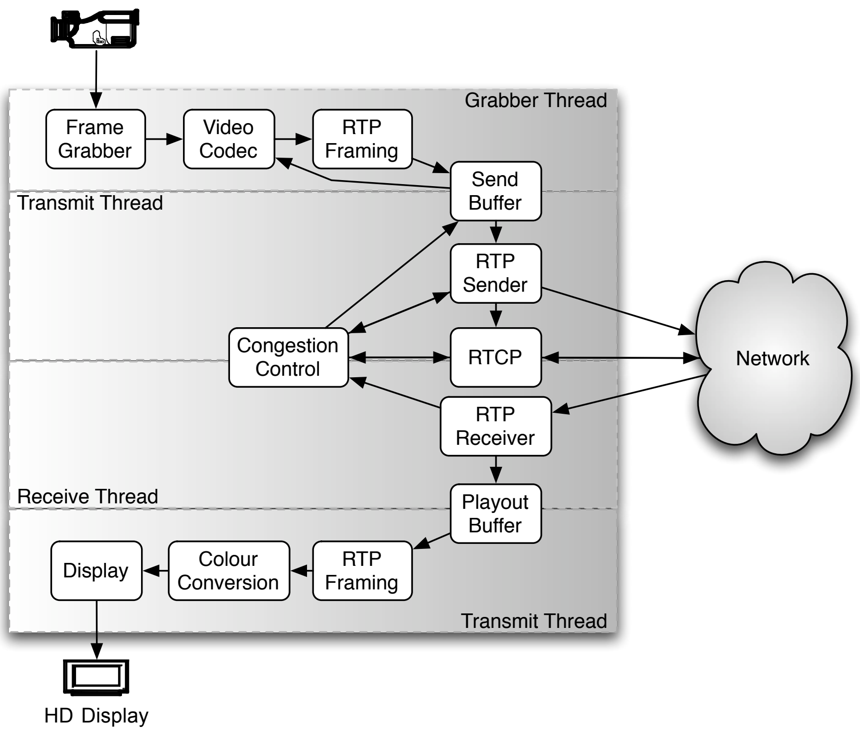


Figure 6.2: Ultragrid overview.

As it can be seen, the sender captures, encodes and transmits frames, while the receiver does the inverse operation when receives, decodes and displays the frames successfully transmitted. However, these two functions present some peculiarities when they are used in conjunction with a congestion control system.

Congestion control is a new element that has been introduced in UltraGrid. This subsystem is located in the core of the transmission system and, even when its immediate function is to establish the sending rate, the indirect effects are spread over more parts of the system.

In the following sections I will describe the most important parts of the sender and the receiver, and how the congestion control system affects their behavior.

6.2.1 Sender

A general overview of the sender functionality is shown in the Figure 6.3. In this figure, we can see two main parts of the sender: the grabbing and queuing system, and the transmission system. Both subsystems, implemented as two concurrent threads, are responsible for the timely delivery of frames to the receiver.

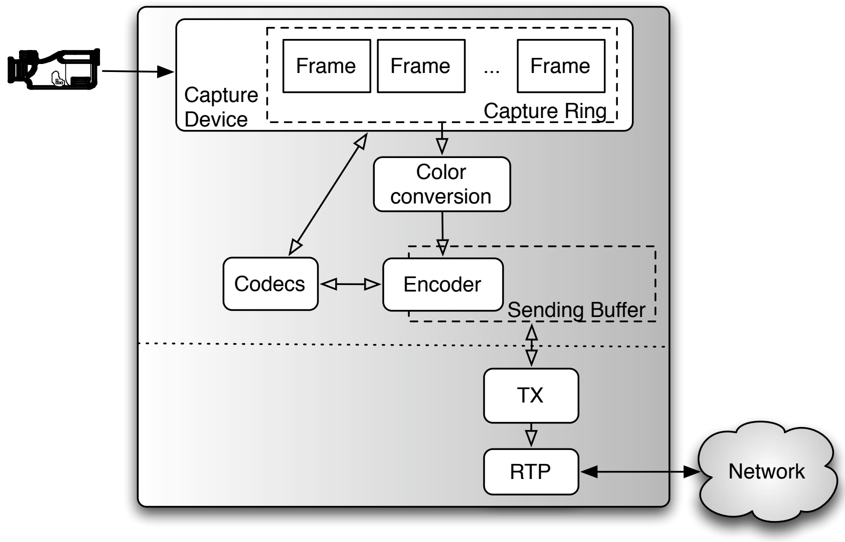


Figure 6.3: Ultragrid sender layout.

The work of the sender starts when the capture devices produce a video frame. Frames provided by these devices may need some decompression or processing, in order to match the encoder used for transmission. Sometimes they will provide frames compressed with a different codec (for example, a *DV* camera producing compressed frames) that must be decompressed. Other times, more simple transformations are needed, like color conversions or resizing.

Capture devices can store frames in an internal ring in order to increase the performance while grabbing. This ring is sometimes provided by the operating system, either as a real queue or as an artifact. For example, the DV capture uses a memory queue in *Linux*, where memory management is completely handled by the operating system, and *Mac OS X* provides a general asynchronous framework where *QuickTime* is responsible for capturing and calling back when a frame is captured. However, the *DV* capture subsystem must implement a ring in other operating systems where these facilities are not available.

Once that frames are available in the adequate format, they are inserted in the sending buffer. The sending buffer is responsible for the compression and/or encoding of frames, storing them until the transmission system retrieves its fragments and the frame is no longer needed. In that case, the frame is marked as old, and it will be a duty of the sender to trigger a periodic cleanup of these frames. Section 6.3.1 will give more details on the sending buffer internals and its relation with the congestion control system.

The last part of the sender is the transmission system. This system is responsible for the spacing and delivery of packets using the RTP/UDP library. In order to increase the accuracy of the inter-packet interval, the transmission mechanism has been implemented as a different thread, independent of the capture and queuing of frames. We will focus on the transmission system in Section 6.3.3.

6.2.2 Receiver

The design of the receiver is built around one main component: the playout buffer. Figure 6.4 shows a general view of the architectural elements of the receiver.

Reception starts at the RTP level, where packets are processed and inserted in a playout buffer. In the current implementation, this buffer has a hard-coded amount of buffering, but more sophisticated systems [55] could be implemented in the future. The current amount of buffering gives a reasonable amount of time for the reception of all the packets of the same frame, and corrects any reordering suffered in the transmission.

When the display time is reached, the playout buffer tries to decode the frame using the current codec, even if it has some lost packets. The result will vary depending on the codec used: some codecs can tolerate a corrupt frame while others will simply discard the whole frame. If the codec has produced a decoded frame, the playout buffer will send it to the display device.

After rendering a frame, it is not immediately discarded. Instead, the playout control system marks the frame as old and it is stored until a future cleanup operation. This behavior shows another important function of the playout buffer in the decoding process: it is a store of frames for the system codecs. Frames are not completely independent for some codecs and they need to access to previous frames in order to decode a new one. Those frames must be kept in the playout buffer and they will only be discarded when the current codec no longer needs them.

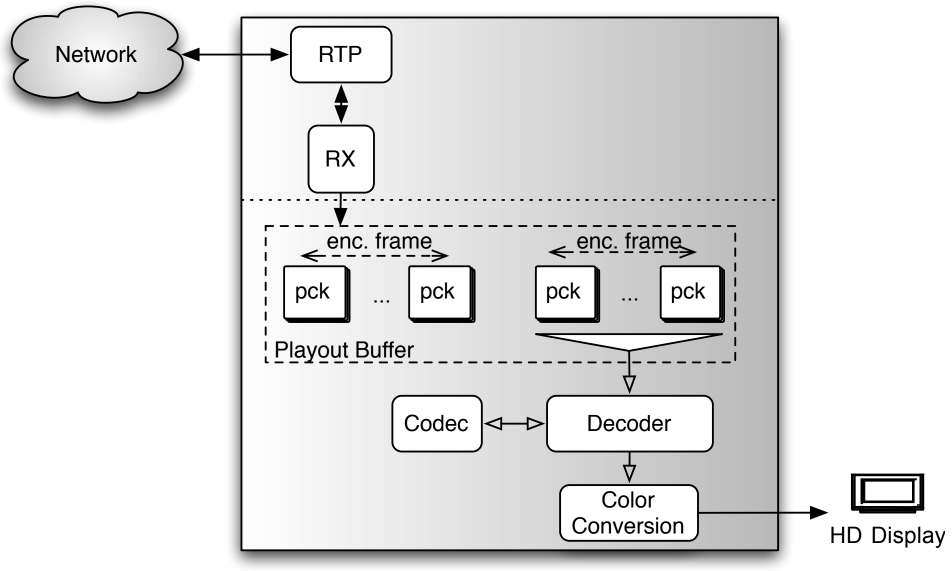


Figure 6.4: Ultragrid receiver layout.

In the current implementation, the display of frames is done in the same execution thread as the playout buffer. However, the design of some display devices decouples this synchronization. The use of double buffering in some displays increases the responsiveness of the system, allowing to write a new frame in the display buffer while the system is, at the same time, displaying the previous one. Some devices (like the *Simple DirectMedia Layer*, *SDL*, or the *QuickTime* display) implement this feature in a transparent way to the application, while for other devices (like the *X-Video*, *XV*, display) this must be implemented in UltraGrid.

6.2.3 Codecs

Codecs play an important role in any media transmission system. Both sender and receiver must use a codec for encoding and decoding frames. In the sender, the current codecs accept a captured frame and, depending on the set of parameters provided, produce a frame suitable for transmission. This packetized frame is then transmitted and, at the receiver, is reconstructed and decoded for the final rendering by the display system.

We can identify several essential tasks that are performed by codecs in this transmission. First, the codec is responsible for packetizing the media, producing transmission elements that can be encapsulated in packets and conform with the current *maximum transfer unit*, *MTU*. Second, this flow of packets is constructed following a convention or standard, enabling the correct reconstruction at the receiver, or even the interoperation with other systems that follow the same standard. Finally, codecs can provide compression for the media content, reducing the bandwidth requirements and the time needed for the transmission.

If we focus on the last characteristic, we can divide the video codecs currently present in UltraGrid in two main groups. The first group is composed of those codecs where the main function is to packetize frames, and there is no compression or the compression system is hard coded at a predefined level. Codecs of this group are the *YUV* [35] or the *DV* codec[1], where frames are segmented and encapsulated with an appropriate format in RTP packets. In the second group could be codecs where the compression applied to the source frame can be controlled using some parameters.

The first codec with compression in UltraGrid has been the Motion-JPEG (MJPEG) codec. MJPEG is a simple video codec where every single frame is compressed using the JPEG [95] compression algorithm. In this way, frames are completely independent, providing more error resistance but at the price of a higher output size.

For the implementation of this codec, a commonly used JPEG implementation has been used, *libjpeg* [1]. The *libjpeg* library can be found in every modern Unix system, and provides a modular compression system that produces standard JPEG files. Nevertheless, this library has presented several difficulties.

First, when a compression operation is performed, the *libjpeg* library does not accept the target size as a parameter. Instead, it requires a *compression factor*: a number in the $0 - 100$ range that specifies the compression level applied to the source image. This is a characteristic of the JPEG compression algorithm, that uses this *quality* factor as a key parameter in the compression mechanism. Applications should specify the factor corresponding to the desired quality, but the output size is unknown in advance and there is no predefined function that maps both values.

In our system, we must specify a target size for the output of the codec. This *"quality"* system used by the *libjpeg* can be acceptable for other interactive application where the

[1]Even when DV is a compressed format, the DV codecs lacks the ability of compressing frames at different sizes. So far, there is no compression functionality in this codec and, due to the rigid compression system of DV (where all the frames have the same size), there are no plans for implementing it.

accuracy of the output size is flexible, but it is not adequate in UltraGrid. As we will see in the following sections, the amount of data that can be generated by the codec is specified by other parts of the system, and the codec must try to act in accordance with these instructions as much as it can.

In order to overcome this limitation, a very simple solution has been implemented where a compression factor is calculated using the last three compression experiences of the codec. When a new compression factor is needed, an interpolation function is obtained using the last three *(ratio, q)* points, and the target compression ratio is used as the interpolation point.

Even though more complex solutions could be used (with a more complex function and using more points), this would probably require a longer history and this could be problematic in some situations (*i.e.*, at the system initialization). As long as frames are relatively similar, a quite good approximation can be obtained with a short history of only two or three samples. Although this could not be used in other environments with abrupt changes in the media content (*i.e.*, movies streaming), videoconferencing frames are characterized by image steadiness. This is the expected scenario for our application, where the temporal similarity of consecutive frames will result in similar compression targets and, as a consequence, similar output sizes for the same *ratio*. Besides that, any history-based solution suffers of some output *inertia*, being resistant to sudden changes, and longer histories could make this situation even worse.

We should not forget that, thanks to the modular design of UltraGrid, other codecs could be easily integrated in the system. In particular, a new MPEG [65] codec is being developed and implemented, and more codecs could use the same infrastructure without any major change.

6.3 Congestion Control in UltraGrid

The most important task performed by the congestion control system is to establish the sending rate in the sender. By setting this rate, the sender knows how fast it can inject data packets into the network and, in consequence, how fast it should be generating data.

The previous generation rate of the sender is controlled by the codec currently used. The codec is responsible for producing a data rate that matches, on the long term, the rate established by the congestion control system. If it generates more data than allowed, some of this data will have to be dropped. On the other hand, if the codec generates a insufficient flow, this will deteriorate the perceived quality of the video stream.

The pairing between the transmission and the generation of data is mostly done by modulating some codecs parameters like the compression level that they can apply to frames. Codecs are controlled by the *encoding context*. This *context* specifies a set of limits and objective that codecs should try to reach (ie, the maximum length for the output of an encoding operation), and it is dynamically adjusted in the place in the system where the data transmission and generation meet: the sending buffer.

6.3.1 The Sending Buffer

One of the most important parts of the sender functionality in UltraGrid is found in the sending buffer. This buffer is the place where the sender stores the encodes frames for transmission. In the sending buffer, packets produced by the encoder are queued and, following the sending rate dictated by the congestion control system, they are extracted at the right time, encapsulated in RTP packets, and sent using the transmission system.

Figure 6.5 shows the basic architecture of the sending buffer. The input in the sending buffer is controlled by the grabbing system, while the end of the queue is controlled by transmission mechanism. This will be the subsystem responsible for taking packets from the sending buffer using the rate calculated by TFRC: the transmission system will get one packet every *IPI* seconds, where the *IPI* corresponds to the inter-packet interval provided by TFRC.

The input of the sending buffer will consist in uncompressed frames obtained from the capture devices. Once that the frame is given to the sending buffer, it will decide if the frame is inserted and how it should be encoded. For making this decision, the sending buffer must perform some checks that try to enforce the current *sending buffer policy*. The *policy* is the set of rules and constraints that govern the behavior of the sending buffer and its relation with other parts of the system. Depending on this policy, the frame can be finally inserted or dropped:

- If there is enough free space available, the sending buffer will encode and/or packetize the frame. In this case, the sending buffer can change the encoding context in order to enforce other constraints. For instance, a possible requirement could be to keep constant the total length (in bytes) of the buffer, increasing the compression

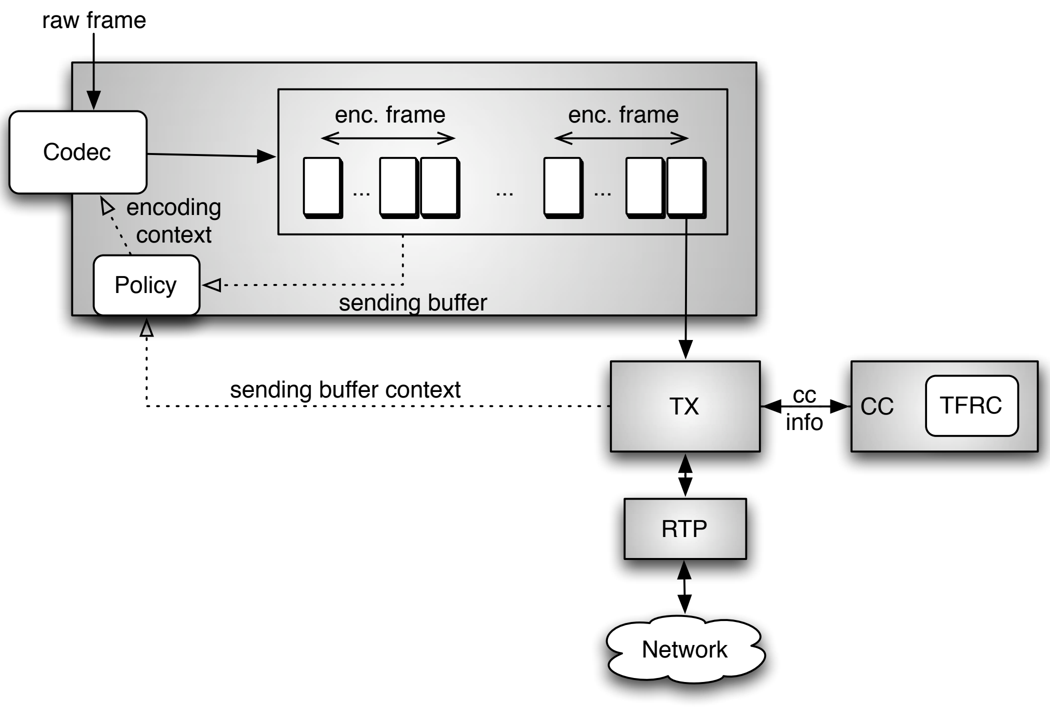


Figure 6.5: Sending buffer architecture, and relation with other parts of the system.

level when space becomes scarce or decreasing it otherwise.

- The frame can be directly dropped. For example, if the sending buffer determines that the arrival time for this frame will be too late, or if the free space available is not sufficient for storing the encoded frame.

As we have seen, the sending buffer policy is not only responsible for deciding if a frame is accepted or not, but it also establishes the right encoding context, adjusting the parameters that will result in the amount of data that the buffer stores. In conclusion, the current policy decides the future of a frame, the compression level and, eventually, the length of the buffer. In the following section we will describe the current policy that has been implemented in UltraGrid, and the benefits and defects of this strategy.

6.3.2 Sending Buffer Policy

The difference between the input and output rate in the sending buffer will result in a variable length of the queue. However, we have seen how the amount of data inserted can be altered by changing the encoding context, modulating the compression level of the current codec and producing more or less data.

The sending buffer policy establishes the global behavior of the buffer, controlling the data generation rate using the current state of the system. An important part of the sending buffer policy is the function that establishes the relation between the sending buffer length and the amount of data that is introduced from the encoder.

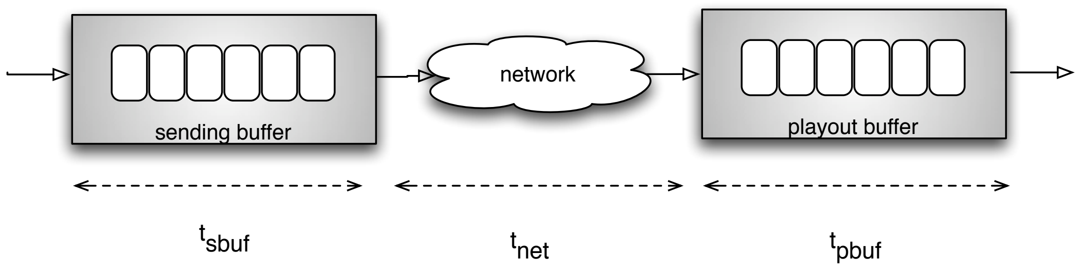


Figure 6.6: Network and buffer belays.

The policy used in the sending buffer must try to enforce a constant end-to-end delay for frames transmitted. The total delay of a frame is the time elapsed between when a frame is captured and the moment when it is finally displayed. For an interactive video system, it has been established that this delay should be between 100 and 150 milliseconds [45].

This total delay of a frame is composed by several delays, depicted in Figure 6.6. We consider the sending buffer delay, t_{sbuf}, as the time elapsed between the moment when the frame is grabbed and the time when the last packet of this frame is sent. The playout buffer delay, t_{pbuf}, would be the equivalent delay but in the receiver, calculated as the time elapsed between the reception of the last packet of a frame and the final display of this frame. The last component, the network delay, t_{net}, can be roughly estimated as half of the current RTT. However, a more accurate approximation of the one-way trip time could be obtained, as the RTT has been proved to be asymmetric [15]. However, and for the sake of simpleness, the network delay has been calculated as a half of the last known RTT.

In conclusion, the sending buffer must limit the queue length in bytes in order to keep the total delay below a time constant, γ, that we can assume to be $150ms$. If the time used of transmitting a frame of length L, at a sending rate X, is $\frac{L}{X}$, then the rule to apply would be

$$\omega + \frac{L}{X} + t_{net} + t_{pbuf} < \gamma = 150ms \tag{6.1}$$

In this formula, ω represents the time that will take the transmission of the previous data in the buffer or, in other words, the time it would take to empty the buffer. If we define θ as the length, in bytes, of the sending buffer at that moment, then ω could be approximated by $\omega = \theta/X$, assuming that all this data is transmitted at the current sending rate. This assumption will be true for short values of θ, due to the smooth change in X, but could lead to a wrong ω estimation for longer values (we will see some problems on this in next chapter).

Knowing that the value of t_{net} can be substituted by $\frac{RTT}{2}$, and using a constant value for the delay in the playout buffer, t_{pbuf}, we can represent the previous expression as:

$$\frac{\theta}{X} + \frac{L}{X} + \frac{RTT}{2} + t_{pbuf} < \gamma \tag{6.2}$$

that can be simplified as

$$L < (\gamma - \frac{RTT}{2} - t_{pbuf})X - \theta \tag{6.3}$$

Enforcing this rule, the sending buffer could transmit the last frame in less than $150ms$, assuming that the current situation in the sending buffer does not change significantly. As long as the sending buffer is not too long, this will be held true and there will be short differences in the sending rate.

Equation 6.3 forms one of the most important rules of the sending buffer *policy*. When a new frame is inserted in the buffer, the maximum frame length for an encoding operation is set in the encoding context using this equation. If the situation is reasonably constant, the encoded frame should arrive at the receiver in less than γ seconds.

Another important directive of the policy is the insertion condition. This is currently implemented in a very simple way: every frame is encoded and inserted in the sending buffer. On an insertion, frames are never dropped by the sending buffer: they are always queued in the buffer. Even though this can result in a longer buffer in some situations, the sender is expected to be provisioned with enough memory for this operation, and there are good reasons for this rule[2].

First, by inserting all the frames, we ensure that the sending buffer is going to be as full as we can. If there is no data available in the buffer it is due to an insufficient input in the system. Otherwise, and provided that the encoding context is set in the right way and the capture flow provides enough data, the sending buffer should always have some data to give.

Second, even although we could try to determine if the frame would arrive in time, it is preferable to delay this decision until the last moment. In fact, it is better to make another check right before the transmission system requests the first packet of a new frame. At this moment, when a new frame is starting, the sending buffer has more updated information, and it can use it for checking if the frame will arrive in time.

If the frame was captured at an absolute time $T_{capture}$ and the expected display time is $T_{display}$ then the frame will be accepted for sending if $T_{display} - T_{capture} < \gamma = 150ms$. Knowing at T_{now} the current sending rate, X, the RTT and the compressed frame length, L, then the frame will arrive in time if

$$(T_{now} - T_{capture}) + \frac{RTT}{2} + \frac{L}{X} + t_{pbuf} < \gamma = 150ms \qquad (6.4)$$

The *transmission condition* is then the following. When a new frame is going to be transmitted, the Equation 6.4 is evaluated and, if this condition is verified, the sending buffer will supply the first packet of this frame to the transmission system. Otherwise, the buffer will mark the frame as old, skip it and repeat the operation with the next frame in the queue.

In addition, this system enables the use of any codec with TFRC, even when the codec can not change the compression level. For example, when the DV codec is used, frames are grabbed from the capture device (ie, a DV camera) and they are simply packetized by the the codec. The DV codec always provides the same number of packets, as DV frames have always the same frame length. In this case, the congestion control system can not modify the compression level and, in consequence, the amount of data inserted in the sending buffer.

However, this is not a problem when we use the transmission condition. In this case,

[2]In practice, the sender has a hard limit for the number of frames that can be queued in the sending buffer.

all the frames will be inserted in the sending buffer and it will establish the encoding context (although ignored by the DV codec). When the transmission system retrieves the first frame, and provided that the conditions enable the transmission in less than $150ms$, the condition will be verified and it will be delivered.

If we examine how frames are extracted from the sending buffer on the long term, we can calculate the output period of frames. We can immediately deduce that if this output period is greater than the input period, then the transmission condition will fail sometimes. For example, if we capture from a DV camera at 25 frames per second, but the transmission system is extracting from the buffer and delivering only 10 frames per second, then 15 frames every second will have to be discarded.

This process is done with the transmission condition. The presence of any transmission lag, due to the mismatch between the input rate and the congestion control rate, will periodically trigger the failure of this condition. By doing this, the system induces a natural reduction in the number of frames per second, interleaving the frames discarded and adjusting the rates mismatch.

6.3.3 Transmission System

In the lowest level of the sender we can find the *transmission system*. This system interfaces with two parts of UltraGrid: at one end it uses the sending buffer as the source of packets, and at the other end it uses the RTP/UDP level for the final delivery of these packets.

The first function of the transmission system is the retrieval of packets from the sending buffer. Following the design shown in Chapter 4, the transmission will be performed in a loop where, while the congestion control systems allows it, packets will be requested from the sending buffer and sent using RTP. In this loop, the sender will also check for any change in the congestion control state and adjust the inter-packet interval by using a simple sleep operation.

There is another task performed in this loop: a flow of information exists in the opposite direction between the transmission system and the sending buffer. Since the transmission system is the only part of UltraGrid that interacts with the congestion control mechanism, it must provide some information about the current state to the sending buffer, using what we call the *sending buffer context*. In this context, the transmission system describes parameters like the current sending rate or the RTT, information that the sending buffer will use for setting the right encoding context.

The other main function of the transmission system is the delivery of packets using the RTP level. A session is created for the delivery of video content, identified by the endpoint addresses and ports. There are several elements for the RTP session created at this level. First, the session must implement a particular RTP *profile*. In our case, UltraGrid uses a RTP implementation that has been modified in order to support the TFRC profile [36].

This profile specifies the interchange of information between senders and receivers by using some additions in RTP headers and extensions in the RTCP *Receiver Reports*. As the receiver must inform to the sender at least once per *RTT*, the profile also includes some modifications in the RTCP timing intervals regarding feedback frequency and their implications.

As we saw in Chapter 2, the RTP *control protocol* allows the interchange of information between senders and receivers, like quality feedback or user information, as well as time-base management functions for time recovery. It is based on the generation of *Receiver Reports* (*RR*) and *Sender Reports* (*SR*) that are interchanged between both endpoints. In the TFRC profile, the *SR* includes information like the current *RTT* known by the sender, and the *RR* reports the last loss event rate known by the receiver, as well as the other information needed by a TFRC sender.

The last part of the transmission is performed with the RTP *data transfer protocol*. This subsystem is in charge of the delivery of the media content, according to one or more payload formats (it can be changed during the transmission).

6.4 Summary

Throughout this chapter, we have seen the basic layout of UltraGrid, the modifications performed in order to include support for TFRC and the main parts affected by this integration. We have also described the basic policy implemented in the sending buffer, studying the constraints that limit an interactive system like this, and discussing the pros and cons of this approach.

In the next chapter we will see how all these elements interact in the system, and the suitability of this solution for a video-conferencing system. Although this design should provide an adequate base for the correct behavior of our system with TFRC, we will see in the following chapter that this can not always be the case.

Chapter 7

Experiments and Evaluation

This chapter presents the results of the TFRC integration in UltraGrid. It will study the design presented in the previous chapter, focusing on the relations between the different parts involved in the dynamics of TFRC in a videoconferencing system.

After an overview of the methodology used, I will describe the dynamics of the system and the basic behavior of the application in Section 7.2. Section 7.3 will expose the dependencies found with the environment, followed by a description of other problems with the data rate and codecs in sections 7.4 and 7.5. Finally, I propose some improvements and future work for UltraGrid in order to enhance the TFRC integration.

7.1 Evaluation Objectives and Methodology

The aim of these experiments is to test the basic behavior of the system and to verify the applicability of the design presented in Chapter 6. While the TFRC testing in Chapter 5 was focused on the response of the protocol in a real environment, these tests try to see if TFRC could be used with a videoconferencing application like UltraGrid.

However, this testing will be highly determined by the design of UltraGrid. It is not important how good a congestion control algorithm is if it is not used in the right way, and our design makes some assumptions about the properties of TFRC a priori. An inadequate integration between the two main dynamics involved in this system, the congestion control mechanism and the videoconferencing control, could result in the wrong findings. So, the results obtained will need to be examined in this context, talking into account the basic design developed for the integration of TFRC.

For these experiments, the environment used is quite similar to the one used in Section 5.1.3. Local area tests have been performed with the network setup shown in Figure 7.1. Routers *R1* and *R2* are used for the network simulation, with *dummynet* running in the last one. The *RTT-bandwidth* pairs will be basically the same used in Chapter 5, but with an slightly higher *bandwidth* due to the amount of traffic generated by UltraGrid.

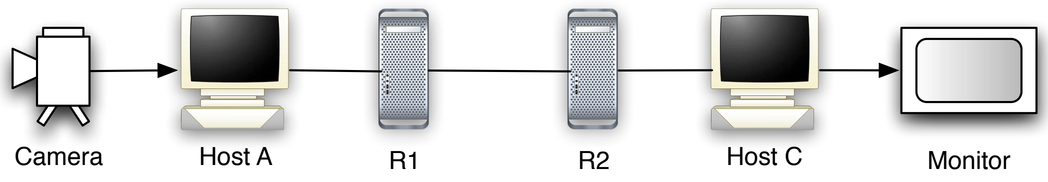


Figure 7.1: Local area testing layout.

In this configuration, Hosts *A* and *C* have been equipped with a video camera and a monitor, respectively. The input video format has always been PAL[1] and the capture frequency in *frames per second* (*fps*) will depend on the experiment. The camera has been replaced by recorded movies in some tests, using simple sequences with *talking heads* that could represent a common videoconferencing scenario.

Regarding the metrics used, the complexity of a a videoconferencing system makes it quite difficult to evaluate. The use of metrics like the *Peak Signal to Noise Ratio* (*PSNR*) only provides a measure of the final result but does not consider all the intermediate steps involved. For a better understanding of TFRC in a real-time multimedia application, we must take into account other aspects of the system, using metrics that allow a deeper study of interdependencies. In consequence, this analysis will be more focused on the dynamics of the system and, in particular, all the data flows that are conditioned by TFRC.

The mentioned *PSNR* is a commonly used measure for this kind of system that we will sometimes employ. The *PSNR* is an indicator of picture quality that is used for the evaluation of video transmission. For a degraded N_1xN_2 8-bits image f obtained from

[1]PAL frame dimensions = 720x576.

the original image F, the PSNR is computed by the formula

$$PSNR(F,f) = 20\log_{10}\frac{255}{(\frac{1}{N_1N_2}\sum_{x=0}^{N_1-1}\sum_{y=0}^{N_2-1}[F(x,y)-f(x,y)]^2)^{1/2}} \tag{7.1}$$

The *PSNR* is, however, a controversial metric. Some authors [59] have shown that, although the *PSNR* to some extent correlates to the perceived quality, this relation is no longer valid for low bit rate video or with packet losses. Nevertheless, this metric has the advantage of simplicity and it provides a good measurement system on average situations. We will see some problems with *PSNR* in Section 7.5, where we will use it for evaluating the performance of the Motion-JPEG codec.

7.2 Sending Buffer Rates and System Dynamics

In this section we will focus on the main data flows involved in UltraGrid and how they are controlled. The first flow is produced by the grabbing system and, after some transformations, it becomes the sending buffer input. On the other hand, this buffer is also the source for the system output, with the transmission system extracting data at the pace established by TFRC.

The system dynamics are then established by a triple relation: the TFRC sending rate dictates the output that must be obtained from the sending buffer, and this must match the input obtained from the codec. If this situation is kept at any time, the sending buffer will have a constant length and the system will utilize all the bandwidth available.

This match between the first and last parameters is illustrated in Figure 7.2, using a connection with $100ms\ RTT$ and $800Kbit/s$ bandwidth and $10fps$. This figure shows the sending rate specified by TFRC and the output rate from the data obtained from the sending buffer[2]. Knowing that the TFRC rate determines the input rate of the buffer (using the policy described in Section 6.3.2), the match between both parameters is the ideal case where the input is equal to the output of the system.

In this example, the scenario has the most favorable characteristics for TFRC in Ultra-Grid. A smooth sending rate variation creates a perfect context where the input rate can easily follow the output rate and, although we can see a sending error about $t = 24$, the sending buffer is fed by a steady input rate during all the experiment. The match

[2] This is an estimation of the X_{real} seen in Section 4.1.4, calculated with an EWMA of the data retrieved from the buffer.

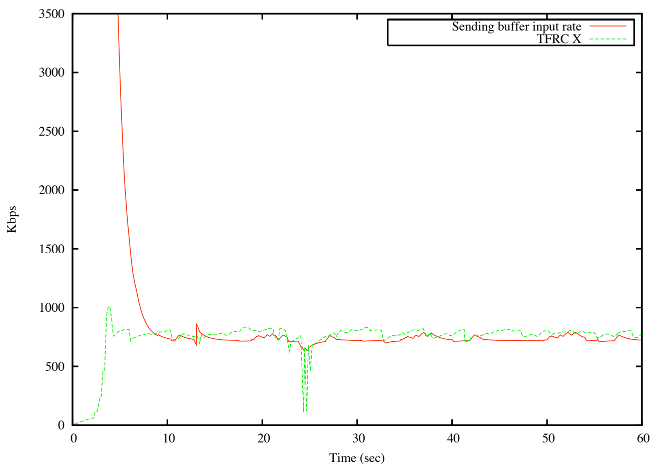


Figure 7.2: Sending buffer input rate and TFRC sending rate ($100ms$ RTT and $800Kbit/s$ bandwidth)

between compression and transmission is quite accurate in this case and the sender uses all the bandwidth specified by TFRC, as we will see in Section 7.3.

As we said, when this occurs, the length of the buffer will be kept constant, although any sending rate change must be controlled in order to maintain this property. In the next section we will see how the sending buffer length is controlled.

7.2.1 Sending Buffer Length

The input and output rates are responsible for the sending buffer length, and the match between both parameters determines the system stability. When the input data rate exceeds the TFRC rate, the sending buffer will grow and some frames will be discarded. On the other hand, a low input rate will produce situations when the sender does not have data to send (we will focus on this problem on Section 7.4).

So, the buffer length does not depend on the input frame rate as it could be thought. In fact, a higher *fps* do not lead to a longer queue, as the thread responsible for inserting frames is also responsible for removing them: it would only result in a higher cleanup frequency. We can see how this works in the following loop:

1. A frame is inserted by the *capture thread*.
2. The *sending thread* sends some packets and, when a frame is completed, it retrieves

the next frame that will arrive in time. Old frames will be marked for a later removal.

3. The *capture thread* starts a cleanup operation before a new frame is inserted. All old frames are removed from the sending buffer and it starts a capture operation, continuing the loop at Step 1.

In this sequence, step 2 is responsible for keeping constant the sending buffer length. Before a new frame is sent, the sender must check if it will arrive in time with the *transmission condition* (Equation 6.4) and, in case it will not, discard it. Using this method, the sender can keep a controlled sending buffer length and spend the bandwidth in transmitting useful frames. We must take into account that the sender never discards a frame after the first packet in the frame is sent. This avoids a possible indecision in the delivery of frames, although it can also lead to very different compression levels for consecutive frames, specially when abrupt sending rate reductions occur.

This control of the sending buffer length is illustrated in Figures 7.3, 7.4 and 7.5, where we can see the number of enqueued frame fragments with an increasing frame rate. These examples have been created in a $20ms\ RTT$ and $6000Kbit/s$ bandwidth environment. In all three cases, the sending buffer length is almost constant, even when the insertion rate in the buffer is quite different. Maybe the only change is that, as a consequence of a higher insertion frequency, the sending buffer is longer more frequently in the last two cases (compare the dashed rectangles in the graph). Nevertheless, the sending buffer shows a controlled length and smooth change in the number of fragments queued.

It is also remarkable the length of the sending buffer in the slow-start phase. Before $t = 10secs$, a high sending rate relaxes the compression level applied and produces a sending buffer with a long list of frame fragments to transmit. We can see in Figures 7.3, 7.4 and 7.5 how the number of fragments in the queue reaches the highest levels between the start and $t = 10secs$.

However, the end of the slow-start stage results in a drastic sending rate reduction. Most of the frames previously encoded are too long to be transmitted under the new circumstances, as a lower sending rate will make them arrive late at the receiver. The sender will probably discard all the frames in the queue, and this frequently leads to an empty buffer and a compression system that struggles to change the output dynamics.

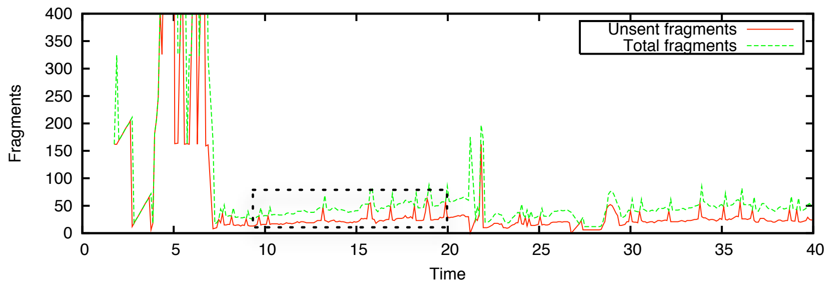


Figure 7.3: 10 *FPS*

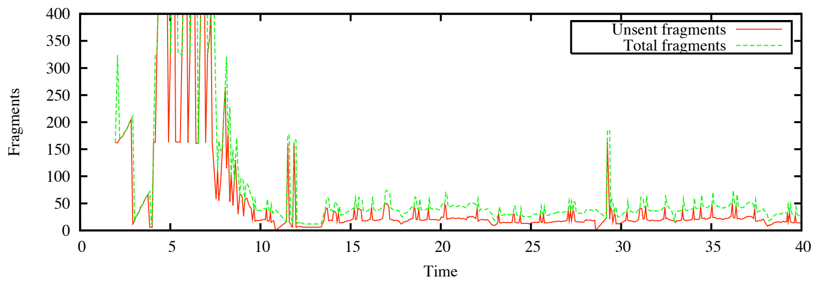


Figure 7.4: 25 *FPS*

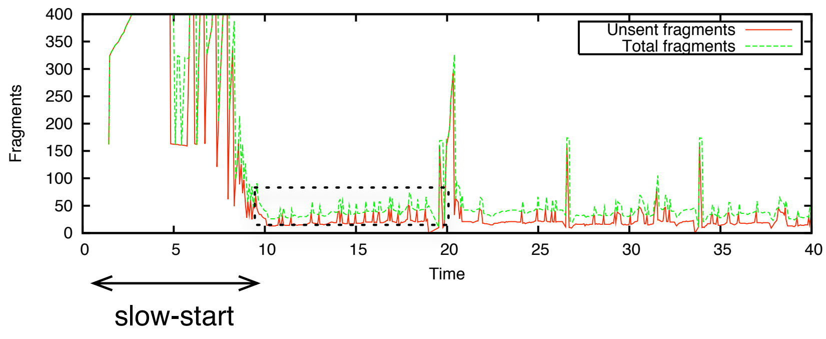


Figure 7.5: 60 *FPS*

7.2.2 Problems with Frame Discards

The mechanism for controlling the sending buffer length can present some problems when the sender discards frames too frequently. If the sender generates more data than

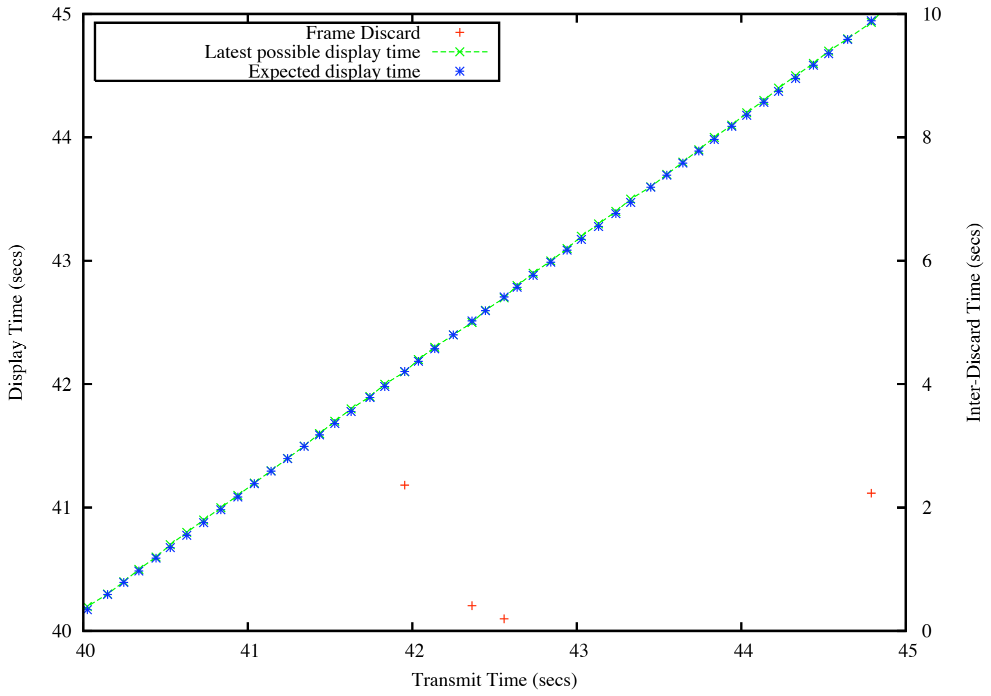


Figure 7.6: Expected and latest possible display times, and time between frame discards (100ms RTT and 800$Kbit/s$ $bandwidth$).

can be transmitted, it will have to discard frames periodically. This can be the result of an unexpected sending rate reduction or any other optimistic estimation of the sending rate when the frame was compressed.

In Figure 7.6 we can see the expected and latest possible display times at the receiver, as well as the time since the last discarded frame, with a 100ms RTT and 800$Kbit/s$ bandwidth session.

This graph illustrates the basic mechanism in the sender for obtaining frames for transmission. When a new frame is due, the sending buffer looks for the next useful frame in the queue. The sender checks the expected display time in order to verify if it will be used, starting from the first frame in the buffer. Figure 7.6 illustrates this mechanism with the parameters used for determining if the frame will be useful or not: the *expected* and the *latest possible display* times.

The *expected display* time is obtained from the current conditions (*i.e.*, sending rate, RTT) and the frame length of the frame. As a smooth change in the sending rate results in a smooth change in the frame length, the *expected display* time will be roughly identical for the first enqueued frames in the buffer. In contrast, the *latest possible display* time is a specific parameter calculated from the particular capture time of each frame, and it will be usually obtained as $T_{capture} + 150ms$. If the *expected display* time is greater than the *latest possible display* time, the frame is dropped since it will not reach the display device in time at the receiver.

As we will see in Section 7.3, the compression system is quite tight and produces the maximum length for frames that will arrive in time. In consequence, most of the frames have an *expected display* time just below the *latest possible display* time in this graph. However, any sending rate reduction will break the previous length calculation, and this tightness will lead to some sporadic discarded frames.

Figure 7.7 shows a scenario where more frequent discards are performed at the sender. When the sender needs a new frame, it will obtain almost identical *expected display* times for all the frames in the queue, as they have very similar lengths. However, we

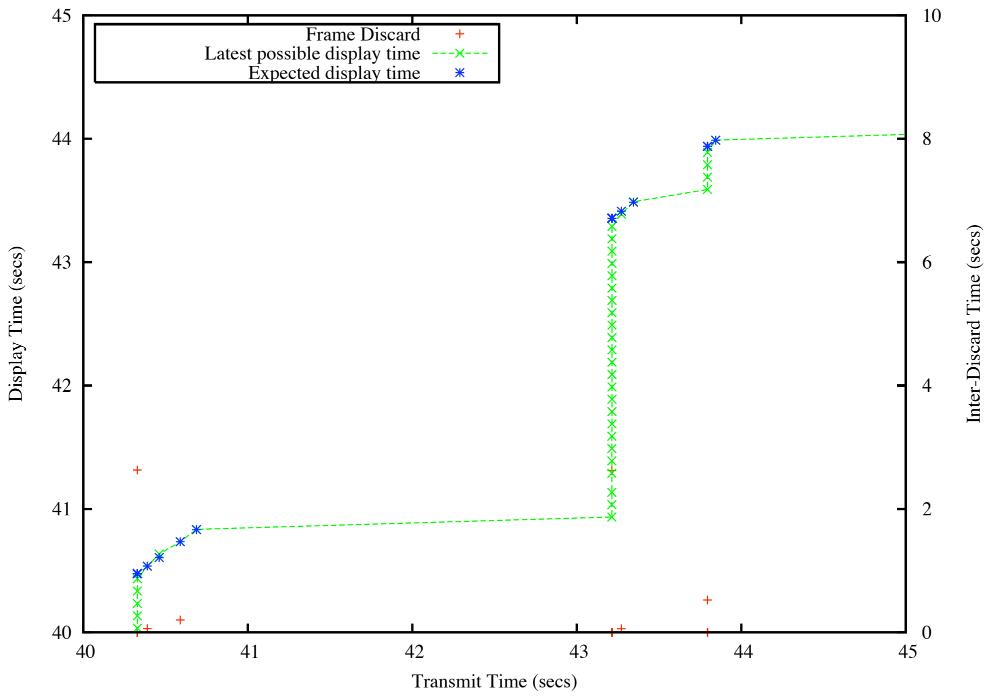


Figure 7.7: Expected and latest possible display times, and time between frame discards ($100ms$ RTT and $800Kbit/s$ $bandwidth$).

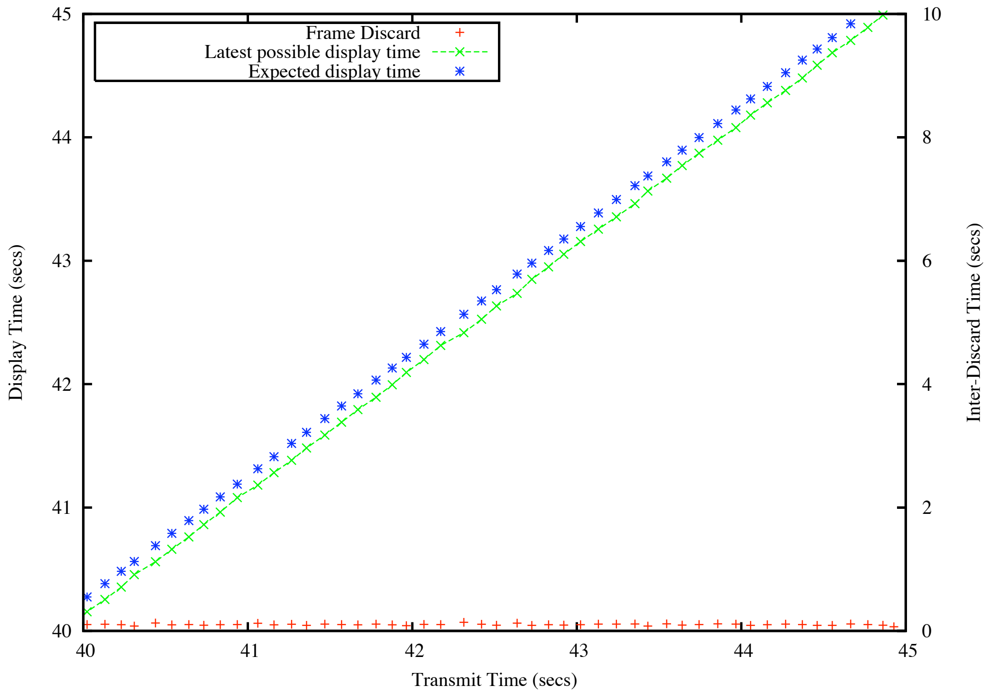


Figure 7.8: Expected and latest possible display times, and time between frame discards ($200ms$ RTT and $3000Kbit/s$ *bandwidth*).

can see how the *latest possible display* times change as it is calculated as a function of the capture time.

We can observe that the sender discards sequences of frames due to a compression level that is inappropriate for the current state. As the *transmission condition* is executed when a frame is due for transmission, we can deduce from the graph that the time elapsed between transmission of frames is up to 2 seconds, an interval far from the capture period of $10fps$. This can be the result of an optimistic sender that, as we will see in Section 7.3, has underestimated the time it will take to transmit a frame.

This mechanism can lead to an inactive sender in some situations. Figure 7.8 shows the same variables but in a scenario with $200ms$ RTT and $3000Kbit/s$ *bandwidth*. Even with this bandwidth increment, the sender will need high compression levels (which it can not achieve in this case) in order to satisfy the very strict timing requirements[3]. When

[3]The RTT specified in *dummynet* does not correspond with the real measured value, as the network buffering will increase the effective value. So the *one-trip time*, calculated as $RTT/2 = 200ms/2 = 100ms$, will be really closer to the $150ms$ limit.

a new frame is due, the expected time calculation will show that it will arrive late and it will be discarded, and this continuous discard of frames leads to a paradoxical situation where all the frames are discarded at the sender and nothing is transmitted.

A completely inactive sender is not a good solution and is probably confusing. Users would prefer to see an image, even with a perceivable delay, rather than a black screen. UltraGrid should take into account this situation and include a timer that could avoid this idleness by changing the allowable latency to force the transmission of a frame when the sender can not achieve the desired compression ratio.

7.3 Sending Buffer Flexibility

In previous sections, we have seen the main dynamics of the system and how the input and output rates determine the sending buffer length. As we have seen, the match between both flows is essential for the stability of the system, and the sending buffer policy must enforce this balance in any situation.

However, while this equilibrium is easy to maintain in scenarios with long RTTs, it presents more difficulties with shorter values. In these cases, rougher changes in the TFRC sending rate lead to a difficult scenario where our sending buffer policy does not perform so well. An example of this is displayed in Figure 7.9, where we can see a case with $20ms$ RTT and $5000Kbit/s$ bandwidth, using $10fps$. In contrast with the match seen in Figure 7.2, this scenario shows a considerable difference between the input and output rates in the buffer, with a sender that can not reach the throughput required by TFRC.

This difference between input and output is the product of the current sending buffer policy. The source of this mismatch is the result of the following sequence of events:

1. Maximum frame length calculation.

   We must take into account that the TFRC sending rate will not be immediately followed by the maximum frame size calculated. An example of this problem is represented in Figure 7.10. With an average sending buffer length of $\bar{\theta}$ bytes, the sender will set a new compression level at t_1 by using the current sending rate X_{t_1} in Equation 6.3. The resulting frame will be enqueued and transmitted between t_2 and t_3 ($t_3 > t_2 > t_1$), $\bar{\theta}$ bytes after t_1.

   For a balance between input and output in the sending buffer, the sending buffer

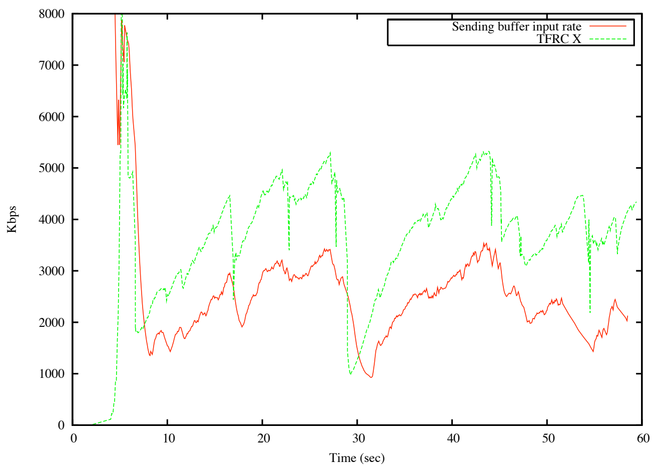


Figure 7.9: Sending buffer input rate and TFRC sending rate ($20ms$ RTT and $5000Kbit/s$ bandwidth)

input at t_1 should not only match the sending rate at t_2, X_{t_2}, but it should be equal to the mean value during the frame transmission, $\bar{X}_{[t_2,t_3]}$. Even if $t_1 = t_2$, $\bar{X}_{[t_2,t_3]}$ can not be safely replaced by X_{t_2} when the sending rate function has a considerable change between frames.

As we have studied in Chapter 5, we can expect such variation in sending rate from TFRC in local area networks and other scenarios with short RTTs. As with long RTTs, the sending buffer will hold a few useful frames in the queue, applying enough compression for sending them just in time. However, the X used for calculating the maximum frame length L will frequently change in this case, leading to frames that either do not use all the bandwidth available, or require more capacity than available.

The source of this error is the simple sending buffer policy applied. The use of X_{t_1} as a predictor of $\bar{X}_{[t_2,t_3]}$ is valid as long as the sending rate has no abrupt changes. Our design produces a disparity between the input rate and the TFRC rate proportional to the slope of the X function and $\bar{\theta}$. However, a higher level of accuracy would probably require a complex prediction of the sending rate that the sender will experience in the future.

2. *Transmission condition.*

We explained in Section 6.3.2 how the *transmission condition* is evaluated prior

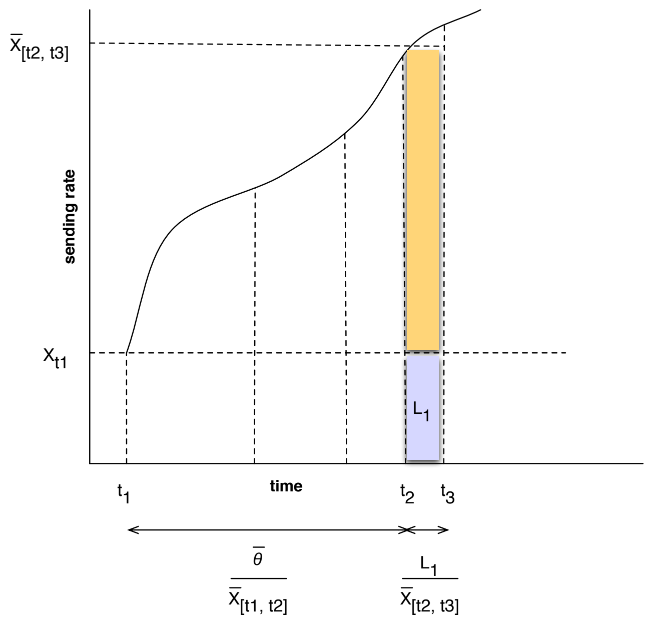


Figure 7.10: Sending buffer rates mismatch.

to transmitting the first packet of a new frame. In the example of Figure 7.10, it would be evaluated at $t = t_2$.

The output rate suffers the second wrong reduction when the sending rate at $t = t_1$, X_{t_1}, does not match the sending rate used in the *transmission condition*, X_{t_2}. In general, any significant output reduction at t', with $t_1 < t' < t_2$, will make false the *transmission condition* and the previously compressed frame will be dropped.

As the compression level is tightly adjusted in order to produce the longest frames that can be transmitted in less than $150ms$, a lower sending rate breaks the timing constraints, the frame is dropped and the difference between output and TFRC rates is increased. A more relaxed compression system could provide more flexibility to the *transmission condition* and reduce the discard frequency, but would reduce the quality of the frames transmitted[4].

[4] The sender could relax the maximum frame length calculation by using an enhanced version of Equation 6.3, where other variables could be included. An EWMA of the relative increment in the last n samples of X, $\triangle_X$, the loss event rate, p, and the standard deviation of the RTT, σ_{RTT}, could be used in a new equation, $L = (\gamma - \frac{RTT}{2} - \sigma_{RTT} - t_{pbuf})\hat{X} - \theta$, where $\hat{X}$ is an estimator of $\bar{X}_{[t_2,t_3]}$ that could be approximated by $\hat{X} = X(\alpha\triangle_X + \beta p)$ (with some unknown α and β) or some other functional combination.

3. Real transmission time.

   As a consequence of this miscalculation, the sender will probably not send the new frame in L_1/X_{t_1} but in $L_1/\bar{X}_{[t_2,t_3]}$. A frame will be transmitted faster if $\bar{X}_{[t_2,t_3]} > X_{t_1}$, and the sending buffer output, obtained from TFRC, will be higher than the input. If this situation is sustained for long periods of time, it can lead to a drained sending buffer.

   On the other hand, sending rate reductions do not counterbalance this effect. The next frame will not satisfy the *transmission condition*, and it will be dropped. Nevertheless, a discarded frame does not necessarily make lighter the pressure in the buffer, and it can produce a lack of data to transmit.

In conclusion, the right behavior of the sending buffer policy depends on the difference between X_1, X_2 and $\bar{X}_{[t_2,t_3]}$ for every frame inserted in the buffer. Flat sending rates will reduce the difference between the TFRC sending rate and the output rate obtained from the sending buffer, while frequent changes in X will increase this error.

The disparity between input and output rates is then highly influenced by network

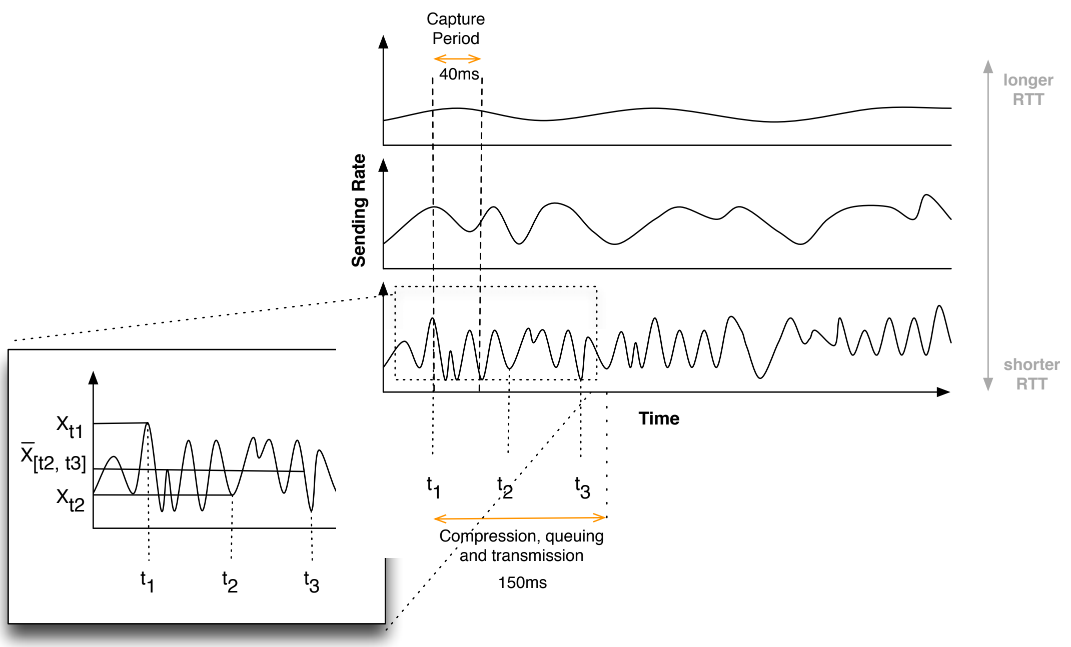


Figure 7.11: Sending buffer and sending rate oscillations.

conditions. The tightness of the maximum frame length calculation is an important factor in this error, but other variables as the the capture period and, indirectly, the RTT (as this parameter affects the sending rate variation, as it has been shown in Chapter 5) play an important part in this problem.

Figure 7.11 illustrates this problem, showing some of these variables and focusing on the effects of a RTT variation with a constant capture period. In particular, it shows the timings for the compression and queueing ($t = [t_1, t_2)$) and transmission ($t = [t_2, t_3)$) of a frame with a short RTT.

Studying this figure, we see that:

- Long RTTs result in more stable TFRC sending rates, and the short variation of X makes X_{t_1} a good approximation of $\bar{X}_{[t_2,t_3]}$. A capture period shorter than the RTT will improve the accuracy, as the sending buffer input state will be updated more often than the network state.
- With a constant capture period, shorter RTTs lead to higher variability in X, and X_{t_1} is no longer a valid estimation of $\bar{X}_{[t_2,t_3]}$. There is also a higher probability of $X_{t_2} < X_{t_1}$, and more frames will be discarded in the *transmission condition.* Longer capture periods would worsen this situation, increasing the difference between these three rates.

Overall, the system will prefer relatively long RTTs and high frame rates. With a varying sending rate or low frame rates, there are more possibilities for a mismatch between X_{t_1}, X_{t_2} and $\bar{X}_{[t_2,t_3]}$. In this particular example, as $X_{t_2} < X_{t_1}$, the sender would probably discard the frame captured at t_1[5].

In addition to these dependencies, we must include another factor that increases this error between input and output. We can see in Figure 7.9 that the difference between X and the input rate is lower for low values and higher for high values. This behavior could remind us of a *Type 1* sending rate error, as seen in Section 4.1.3, where the TFRC sender wrongly thinks that it is reaching the sending rate.

In the UltraGrid case, the sending buffer policy uses the current sending rate, X_{t_1}, for calculating the maximum frame length. A wrong value will result in an optimistic sender that compresses frames at a lower level and will have to discard them more frequently. Higher differences between X_{t_1} and $X_{t_1 real}$ will break the balance between input and

[5] The times represented in Figure 7.11 are spread over a long period of time for a better understanding of the problem, with long queuing and transmission times.

output, and this will feedback the error and worsen the miscalculation.

In conclusion, the sending buffer input depends on a wider group of factors than initially thought. As variations at short-time scales lead to wrong compression levels, the policy should take into account the long term state of the sender without reducing the responsiveness to network changes. The correct sending buffer policy must consider a complex set of variables, and react in the right way under very different circumstances. The development of a new, more elaborated policy is an unfinished issue that will be accomplished in future versions of UltraGrid.

7.4 Output Rate Problems

So far, we have seen scenarios where UltraGrid could potentially produce the output rate specified by TFRC. In an ideal case, we can assume that this will be true: the sending buffer is not empty and the sender always has some data to transmit. However, we can imagine some cases where the input rate can not satisfy the need for data at the sender. Not only high bandwidths can drain the sending buffer, but also low capture frequencies, short frame sizes or any other situation that does not produce abundant data could lead to this situation. Nevertheless, the tight relation between the input and the output of the buffer is a more common source of this problem.

We can see the sending buffer length as a function of the current network state. Although there will be more frames stored in the buffer, the real amount of frames that will be sent depends on the current RTT and bandwidth. In situations with long RTTs or low bandwidths, this leads to very short buffer lengths of just a few milliseconds.

For example, with a 100*ms* RTT and 1000*Kbit/s*, we can expect a very short queue, with some moments where the sending buffer will be empty. Sequences of frames will be discarded due to late arrivals, and the effective number of useful frames will be quite short. In fact, when a new frame is needed, only the last frame in the sending buffer will probably be used, and an even longer value could lead to a waiting time for a more recent frame. In general, we can expect some empty buffer situations when the network delay plus the capture period is greater than the maximum transmission time or, using the notation seen in Section 6.3.2, when $t_{net} + 1/fps > \gamma$.

This situation is illustrated in Figure 7.12, where we can see the problem as a function of t_{net} in a scenario with frames captured at 25*fps*. For shorter values of t_{net}, it is more probable to find a frame in the queue that will arrive in time. In fact, if $t_{net} + 1/fps < \gamma$

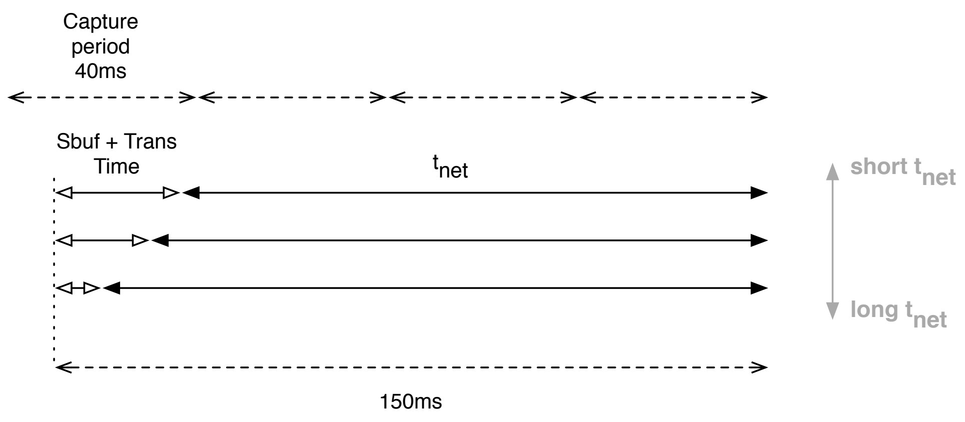


Figure 7.12: Relation between transmission delay, capture frequency and lack of data in the sending buffer.

there must be at least one frame that satisfies this condition. For longer values of t_{net}, present frames could be discarded due to late arrival and the next useful frame maybe is not still available. In this example, the sending buffer can be empty for a relatively long time, waiting for a frame for up to $40ms$.

This data shortage can present a a problem for a videoconferencing system that uses TFRC. As we saw in Chapter 3, the protocol must perform a slow-start when it has been inactive. In fact, this strategy is enforced when there is an *"idleness problem"* in *DCCP* [75], but it can present some problems for a videoconferencing system.

Imagine a frame where the first packet is transmitted at the commence of slow-start, at $t = 0$. With a RTT of $100ms$, the second and third frame fragments would be transmitted after $t = 100ms$, and the next four fragments would be transmitted after $t = 200ms$ and so on. The complete frame would be received well after the $150ms$ recommended limit. So, we must try to avoid any slow-start phase during normal operation as, otherwise, we will be transmitting useless frames[6].

We can see this effect in Figure 7.13, with a network RTT of $100ms$ and $1000Kbit/s$ of bandwidth in the bottleneck. A low capture frequency of $10fps$ will make data available every $100ms$, resulting in a probable waiting time for new frames. Even although the

[6] The alternative, where the sender transmits frames using slow-start, would also produce the wrong result: it would lead to the transmission of partial frames that would result useless at the receiver (or would complicate the decoding process).

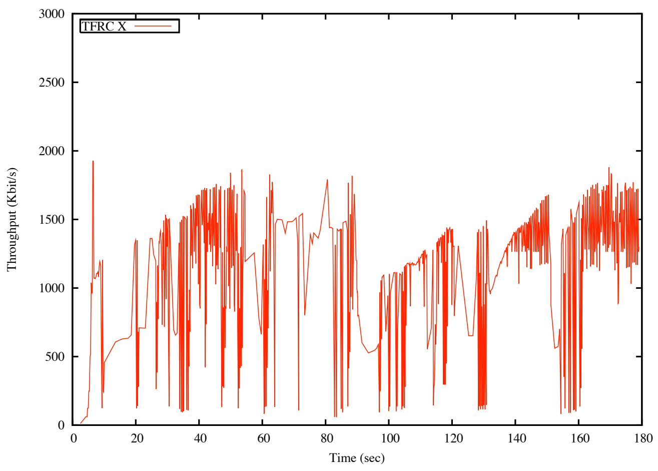


Figure 7.13: TFRC sending rate with oscillatory throughput.

need for data in the sender is something temporary, a low X_{recv} in a feedback report will result in a complete slow-start phase.

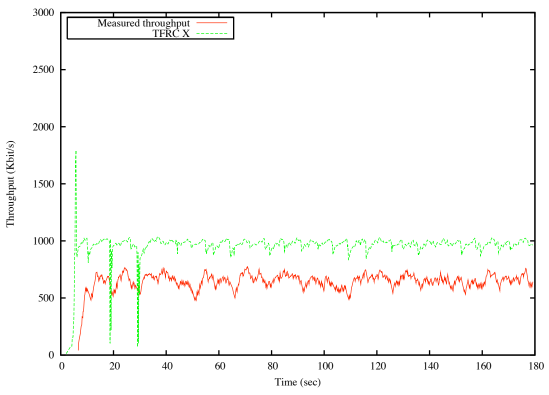
As a consequence, the slow-start algorithm produces an oscillatory sending rate in the sender, with full cycles of inactivity and transmission. This leads to unstable network conditions[7] an a TFRC X that doesn't match the real network capacity: the congestion control algorithm destabilizes the transmission system with a wrong sending rate calculation. We can see in Figure 7.13 how the sending rate is frequently over the bottleneck rate of $1000 Kbit/s$.

This problem can easily be solved if the sender has some data to transmit, so we have implemented a sending buffer that *always* produces data when it is needed. This could be packets with redundant data, some kind of error correction information or simple *"dummy"* packets. The simplicity of the last solution has made it the preferred alternative, and the UltraGrid sender will use packets containing uninitialized data when there is no data available in the sending buffer[8].

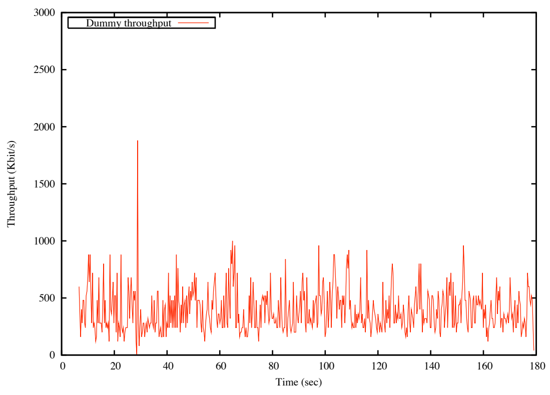
Figure 7.14 represents the main input and output rates obtained for a session with UltraGrid, with the same scenario seen in Figure 7.13 but using *"dummy"* packets. In Figure 7.14(a), the *"TFRC X"* rate represents the sending rate specified by TFRC, while

[7] In this case, a sender that switches between full transmission and complete inactivity will result in an oscillatory RTT, as network routers will suffer drastic changes in their buffers in an environment like this, with a low degree of statistical multiplexing.

[8] A robust implementation would send error correction data.



(a) TFRC sending rate and output throughput



(b) Dummy packets throughput

Figure 7.14: Sending rates, with dummy packets throughput.

the *"Measured Throughput"* is obtained from the output of the sending buffer.

As we saw in Section 7.2, the TFRC sending rate determines the sending buffer output rate, so both measures should match. However, there is a difference between the real output and the TFRC sending rate resulting from the lack of data at the right time. This difference is filled with the *"dummy"* packets. The *"Dummy Throughput"* in Figure 7.14(b) represents the throughput calculated from the *dummy* packets generated in the sending buffer. They have an abrupt nature, as they are generated for short periods of time between frames. Once they have been transmitted, they will be computed as regular data and immediately discarded at the receiver.

The difference between the expected and the real output rates increases with a lower probability of finding a frame with $t_{net} + 1/fps < \gamma$. A sender that is constantly late and discards most of the frames will send large amounts of *dummy* packets. In fact, the scenario shown in Figure 7.8 will result in a sender uses *dummy* data for all the bandwidth available. Other situations where most of the bandwidth will be used by *dummy* data include high bandwidth environments or any other scarce data input that will quickly drain the sending buffer.

In conclusion, although it is not an graceful solution to the problem, by using dummy packets the sender can avoid oscillations and keep the sending rate stable. More investigation is needed in order to find more useful alternatives that could fit in real-time media systems.

7.5 Codec Issues

Codecs are an essential part of a videoconferencing system. They are an important step in the transformation of the input rate in a suitable output rate, and they must obey the parameters that establish this relation. It has, however, been difficult to achieve the required level of accuracy and quality with the Motion-JPEG codec. We can distinguish two main problems with the codec: some weakness when packets are lost and errors in the *quality* calculation.

The M-JPEG codec improves the interactivity and loss resilience by reducing the dependency between frames. However, there is still a dependency between packets used for transmitting a frame: the correct decoding depends on the reception of all these packets. Resilience to packet loss can be improved by isolating data blocks in the frame with *restart markers*. These are positions in the frame that can be used as restart points when some data is not available, allowing the processing of a frame with lost packets. The M-JPEG codec currently uses the maximum value currently supported by the *libjpeg* library: one restart marker per frame row. The result can be seen in Figure 7.15, where we can see how missing fragments are replaced by previous data.

Due to missing packets and limited use of restart markers, the overall behavior of the M-JPEG when packet losses occur does not seem satisfactory. Losses have a high impact in JPEG frames, reducing the perceived quality of the video and the PSNR calculated. Figure 7.16 presents the PSNR calculated with Equation 7.1 for a sequence of the scenario seen in Figure 7.14, reaching an average value of only 18.8831 when a good quality

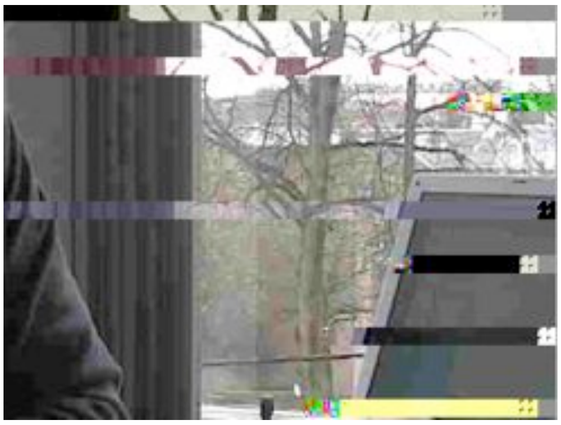


Figure 7.15: Missing lines in JPEG frame.

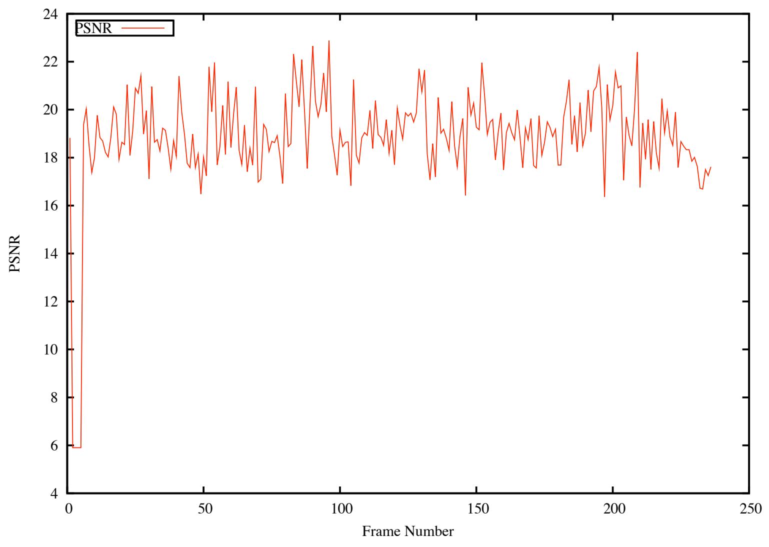
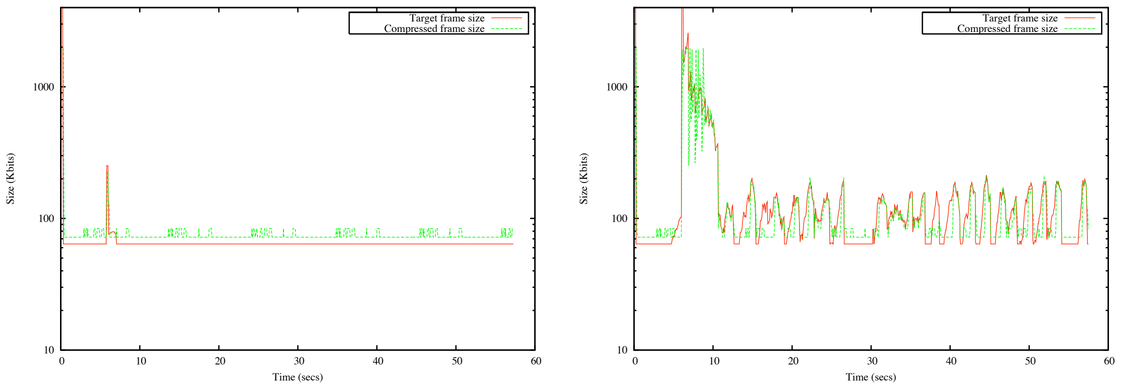


Figure 7.16: PSNR.

transmission with an average loss should be over 25 [22]. Not only the low value obtained but also the frequent variation of the PSNR indicate that the perceived quality will be poor.

In addition, the CPU performance requirements have forced a low-quality configuration of the JPEG system, with a reduction in the *pre* and *post* processing applied to frames. The parameters that configure the *DCT* algorithm, color dithering or image smoothing must be established at reasonably low values in order to reach some potential sending rates. Otherwise, the amount of data generated or the CPU consumption could lead to unsustainable loads, and dynamic adjustments would require a complex framework that would take into account previous info or use a trial and error strategy. Even so, the performance of the *libjpeg* seems very poor: 20*fps* in a 3*GHz* machine.

In conclusion, we must reduce the number of dynamic parameters in order to simplify the compression system, and the look for a target frame length has to be done with just one JPEG parameter. However, even this simplification seems to present the highest difficulties. In fact, the main problem of the codec seems to be the calculation of the *quality* (q) parameter. As we described in Section 6.2.3, JPEG uses a q parameter for setting the frame compression level. The M-JPEG interface uses an interpolation function obtained from the recent compression history for the calculation of a proportional q for a target ratio.



(a) 100*ms* *RTT* and 800*Kbit/s* *bandwidth* (b) 10*ms* *RTT* and 8000*Kbit/s* *bandwidth*

Figure 7.17: JPEG real and expected frame sizes.

This interpolation mechanism can fail in some situations. As the interpolation function is obtained from the last *(ratio, q)* samples, there can be some situations where the n points of the history form a straight line. In cases like this, the q calculated with this function will be the same for any target ratio sought. Our system must break this situation by using an alternative q calculation, generally not so accurate, but that will also break the uniformity when it is added to the history.

We can see some examples in Figure 7.17, showing the output produced by the M-JPEG codec for two different *RTT-bandwidth* combinations: 100*ms*-800*Kbit/s* and 10*ms*-8000*Kbit/s*. The sender captures frames at 15*fps* in both cases.

These graphs show the match between the target and the resulting frame sizes. We can notice a frame length slightly over the target in Figure 7.17(a) due to the flatness previously mentioned. The sender uses a simpler q calculation that prefers overestimations in this case. In addition, the granularity of q adds some error to the output, in the form of sporadic spikes in the length.

When the target length is more irregular, the codec can use the more accurate calculation. However, too many changes in the compression lead to other errors. The interpolation function induces some *inertia* of the output, with a tendency to produce the same result even if conditions are different.

Figure 7.18 shows this effect for a detail of the scenario shown in Figure 7.17(b). This case shares some characteristic with the scenario shown earlier, like an output over the target when this is flat. When there are changes, the q calculation mechanism is mod-

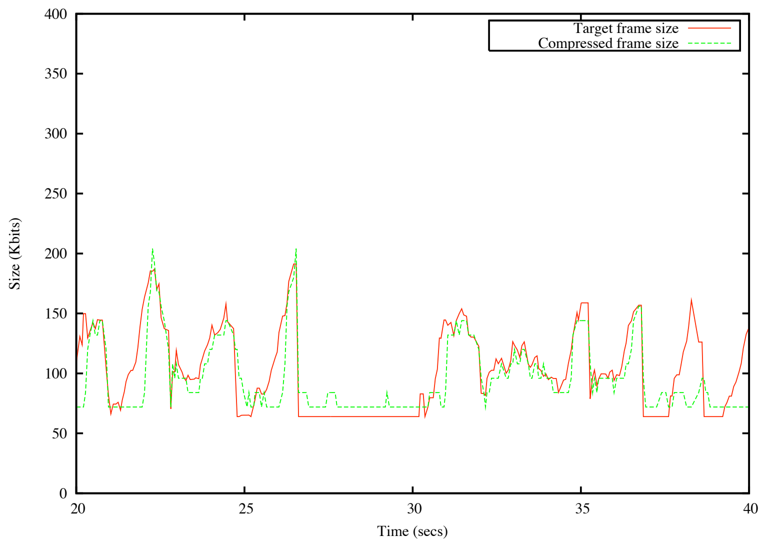


Figure 7.18: Detail of frame sizes for Figure 7.17(b).

erately inflexible, with a preference for previous values and a more continuous output length. This property is not concerning for constant target sizes, but abrupt changes could lead to an unpredictable output.

7.6 Summary and Future Work

In this chapter, we have studied the main parts involved in the behavior of UltraGrid after the introduction of TFRC. The new congestion control algorithm was more complex to integrate than expected, and our design is not as flexible as we would like.

With smooth conditions, TFRC is a suitable congestion control system for UltraGrid. The system has a satisfactory functioning with a $100ms$ RTT and a moderate capture frequency, and it can be used for videoconferencing in real scenarios like this.

However, we have identified several environmental aspects that reduce the operability of UltraGrid. As we saw in Section 7.2, the codec can produce an insufficient input rate when there are strong changes of the sending rate. This effect can be observed in environments with short RTTs and high bandwidths. On the other hand, long RTTs increase the probability of late frames, and the sender will reduce the effective output by dropping an increased number of frames and producing more *dummy* data.

Broadly speaking, UltraGrid needs a more complex control system that could bind all the data flows involved. The adaptation of these flows to varying conditions needs more sophisticated mechanisms and a better adaptation to the environment. For example, the timing requirements could be relaxed with a dynamically adjusted playout buffer, reducing the pressure on the sending buffer and increasing the global system interactivity. Or we could develop a more flexible *transmission condition* that considers the long term sending rate.

Furthermore, the M-JPEG codec has not been a big help in the final result. The biggest advantages of the M-JPEG codec, quick adaptation to the throughput and independence between frames, have turn to be the biggest drawbacks. The M-JPEG codec has proved to be excessively fragile when losses occur, the *quality* calculation needs to be corrected in order to avoid problems, and performance does not allow high capture rates. Codecs are one of the most important parts of UltraGrid, and this is an aspect that must be improved in future versions in order to get the most of the system.

We have also seen in Section 7.4 that the *"idleness problem"* is not the only case where the sender needs to produce *dummy* data. This can happen with relatively low *RTT*s, as the capture and the sending buffer retrieval frequencies do not always match. TFRC needs a mechanism for *explicit bandwidth usage*: it can not see the difference between dummy and real data but it needs the packets length for updating the state at the receiver. Although use of *dummy* packets can be acceptable when it is an sporadic event, it is a terrible solution when it consumes most of the bandwidth, especially if the application is completely inactive. If the sender could specify arbitrary lengths in its packets, different from the real value, the receiver could do the right computation and still use the right amount of bandwidth.

In conclusion, UltraGrid needs some improvements in order to reach a complete integration of a congestion control system. In addition to the problems seen in Chapter 5, TFRC should include new mechanisms for a better utilization of resources. The protocol seems to forget some aspects of an interactive video-conferencing system, impeding the right interaction between the transmission and upper levels.

Chapter 8

Conclusion and Future Work

This thesis started with an overview of the congestion control problem for videoconferencing applications in Chapter 2. The TFRC protocol was described in Chapter 3, explaining its advantages and summarizing the most important characteristics of the congestion control algorithm. The next two chapters were focused on TFRC on the real world, and comprises the first major part of the thesis.

Chapter 4 studies the problems surrounding a TFRC implementation. In Section 4.1, I explained that TFRC requires accurate timing support from the operating system, in particular in some environments, otherwise the sender and receiver will make errors that affect the final throughput. The sender can also deliver packets in long sequences, increasing the need for buffering in the host and network. In consequence, the design of an application that uses TFRC is constrained by all these problems, and I presented some suggestions for the implementation in Section 4.2.

Chapter 5 provided the results of the TFRC experiments. I demonstrated the correctness of the implementation by showing the behavior of TFRC in a broad variety of scenarios, comparing it with the results described in the literature. With the help of *dummynet*, I demonstrated in Section 5.2 that TFRC has good stability and smooth sending rate variation, in general much better than TCP, although the protocol reaches the best results with long RTTs and environments with a high degree of statistical multiplexing. We obtained equally satisfactory results in Section 5.3, where the protocol was finally tested with wide-area experiments, although the dependencies with the operating system and the associated problems where confirmed by these findings.

The second part of this thesis has been focused on *UltraGrid*, our videoconferencing application. I introduced the design of UltraGrid in Chapter 6, where I showed the design details, reasoned the suitability for a TFRC integration and discussed a basic design for this integration. After an introduction to the codecs used by the application, I described the changes needed in the sending buffer, the relation with TFRC and how the data flows are controlled in order to satisfy the timing constraints.

Finally, the integration of TFRC in UltraGrid was evaluated in Chapter 7. It presented an overview of the dynamics of a videoconferencing system, the balance between input and output in the system and the dependencies with the environment. We saw how these dependencies can break the fragile equilibrium in Section 7.2, especially with too short or too long RTTs. I also presented the *"idleness problem"*, and why the sender must use *"dummy"* packets in order to stability the system.

8.1 Future Work

TFRC shows some dependencies with the hardware and operating system that must be solved. A congestion control system is not an isolated mechanism, and it must be designed for interacting with all the levels involved in the communication. If the algorithm forgets one of these elements, the whole mechanism can fail. The protocol needs the inclusion of some advanced system that could take into account these errors and adjust the global behavior. These modifications could counterbalance the errors produced by external factors and reduce the dependencies with the operating system.

In consequence, TFRC is not usable in all possible environments. It needs some improvements in order to be a general purpose protocol for multimedia traffic. The instabilities found with short RTTs in Section 7.3 make it inadequate for local or maybe national connections. Even though the operating system could help in this problem, the protocol needs an internal solution that could be used in any host system.

Regarding the integration of TFRC in UltraGrid, this objective has not been completely successful. Further investigation is required into determining the right control system for one of the most important parts of the application: the sending buffer. The timing requirements are a disadvantage in this case, and the system dynamics cause some difficulties for controlling the input and output data flows. We could get a more relaxed environment with a dynamically adjusted playout buffer, giving more time for transmitting a frame and responding to changes in the network.

It would be interesting to expand the range of codecs supported by UltraGrid, too. This is particularly relevant for UltraGrid, as the existence of a robust variable rate codec determines a stable and continuous data flow. Maybe the use of more advanced codecs could increase the loss resilience, but it would also decrease the interactivity by adding some processing delay. This is an issue that must be studied in future versions of the application.

8.2 Conclusion

During this work I have demonstrated that TFRC can be used for interactive applications in some scenarios. However, I have identified a list of problems in TFRC and its integration in a videoconferencing tool, delimiting the cases where the protocol reaches the expected results. I have also proposed solutions and improvements for these problems, outlining the main points for a complete solution.

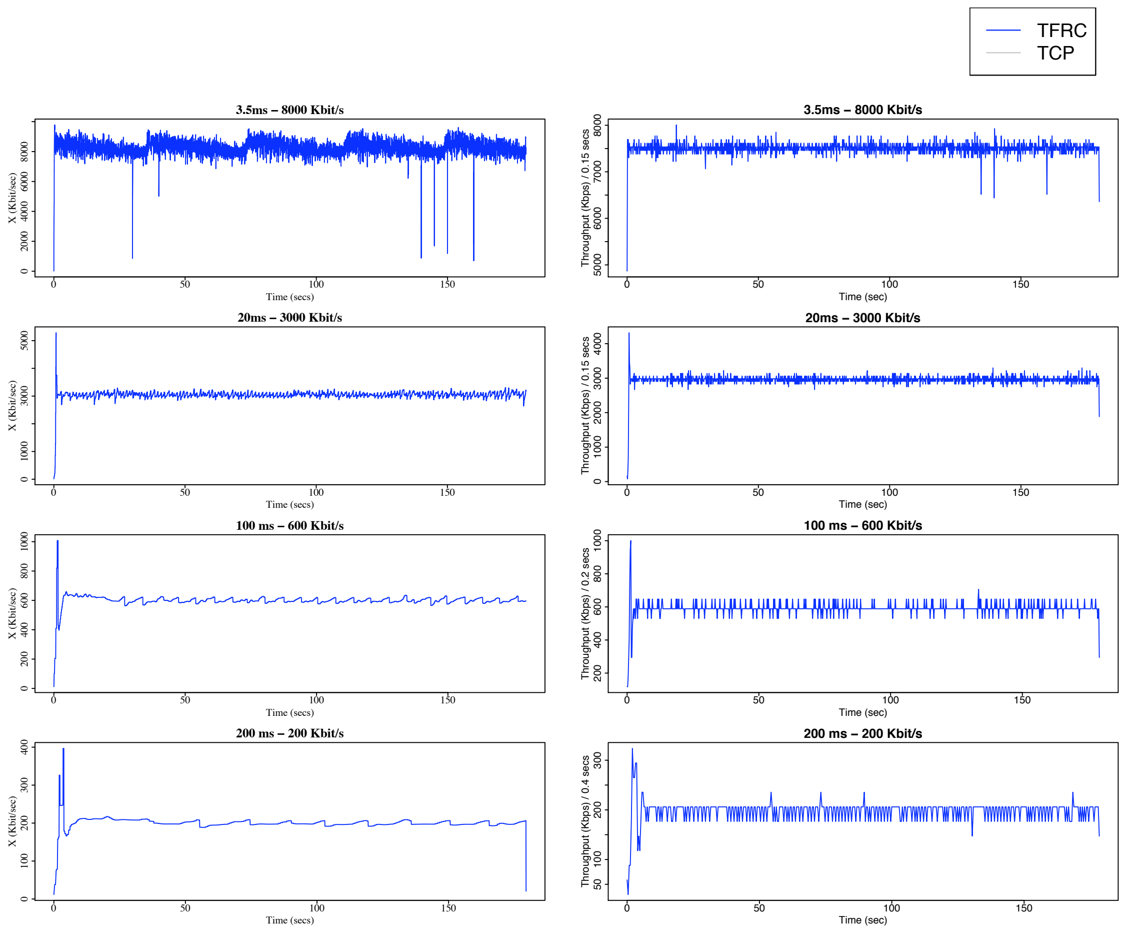
At the beginning of this dissertation, I raised the question *"is TFRC suitable for interactive videoconferencing applications?"*. Although it still needs further improvements in order to be a general purpose congestion control protocol, TFRC could be a suitable solution for this problem in favorable conditions and with more elaborate application and codec support.

Appendix A

TFRC Experiments: Details

This appendix provides details for the discussion of Chapter 5.

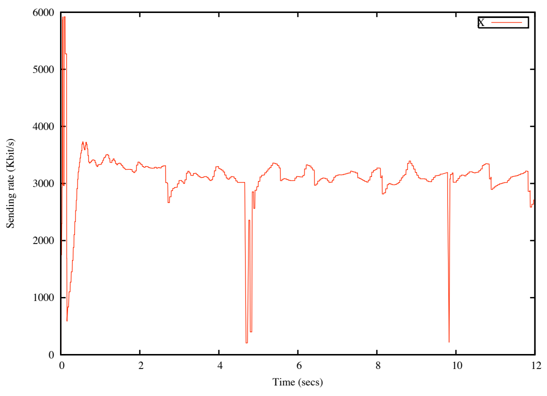
A.1 Aggressiveness when starting up and steady-state behavior



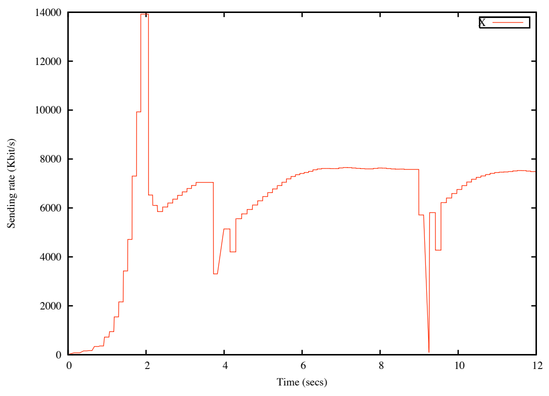
(a) TFRC X

(b) TFRC Throughput

Figure A.1: Sending rate (X) and throughput on the sender for steady-state. *This corresponds to the scenarios shown in Section 5.2.1. In particular, it shows the same connections seen in Figure 5.3*



(a) $RTT = 3.5ms$, Bandwidth $= 3000Kbit/s$



(b) $RTT = 100ms$, Bandwidth $= 8000Kbit/s$

Figure A.2: Sending rate errors. *Details corresponding to scenarios shown in Section 5.2.1:* $3.5ms/3000Kbit/s$ *and* $100ms/8000Kbit/s$ RTT/bandwidth *scenarios in Figure 5.4.*

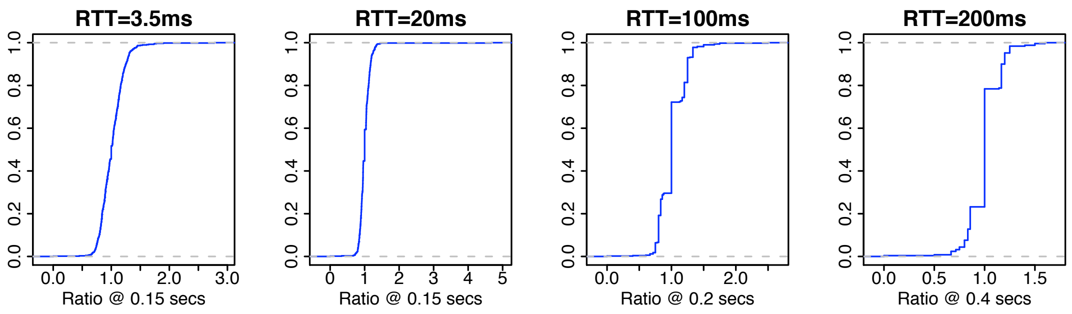
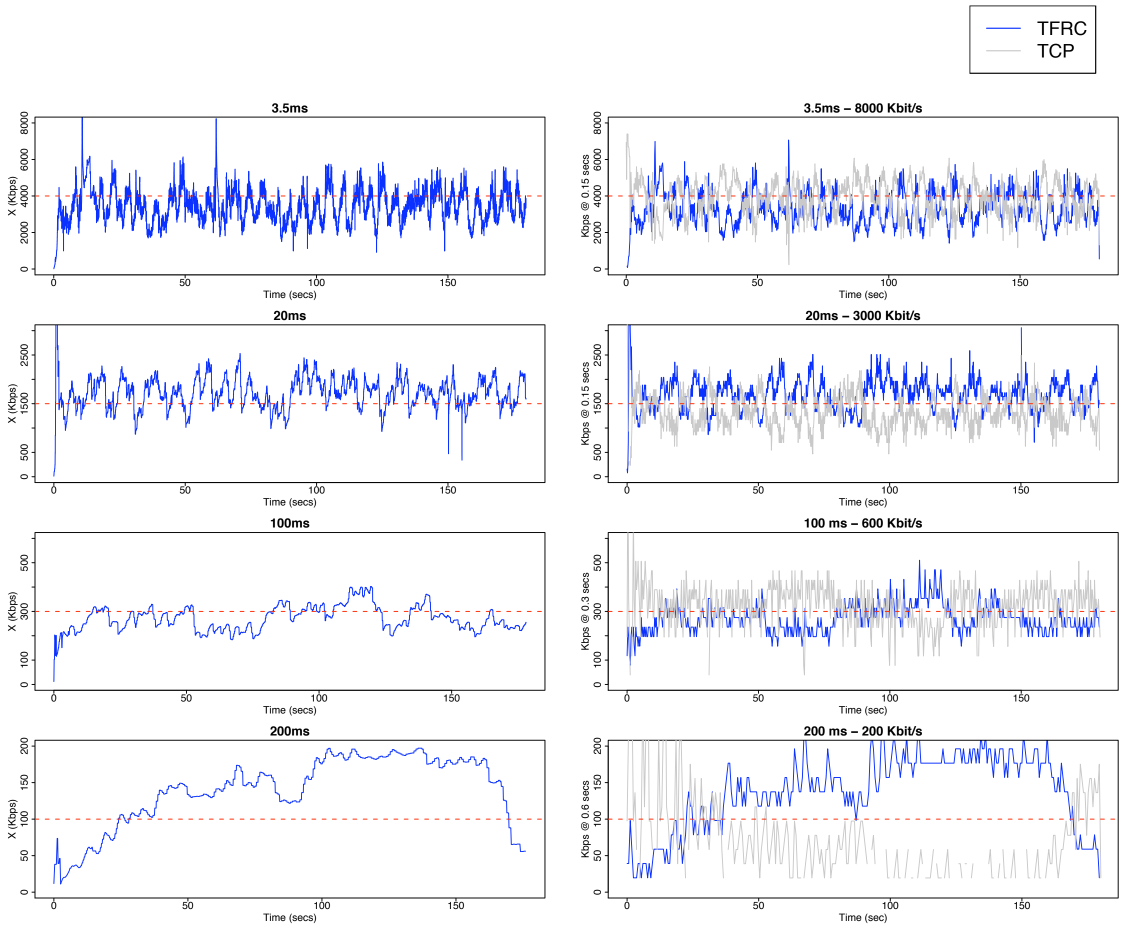


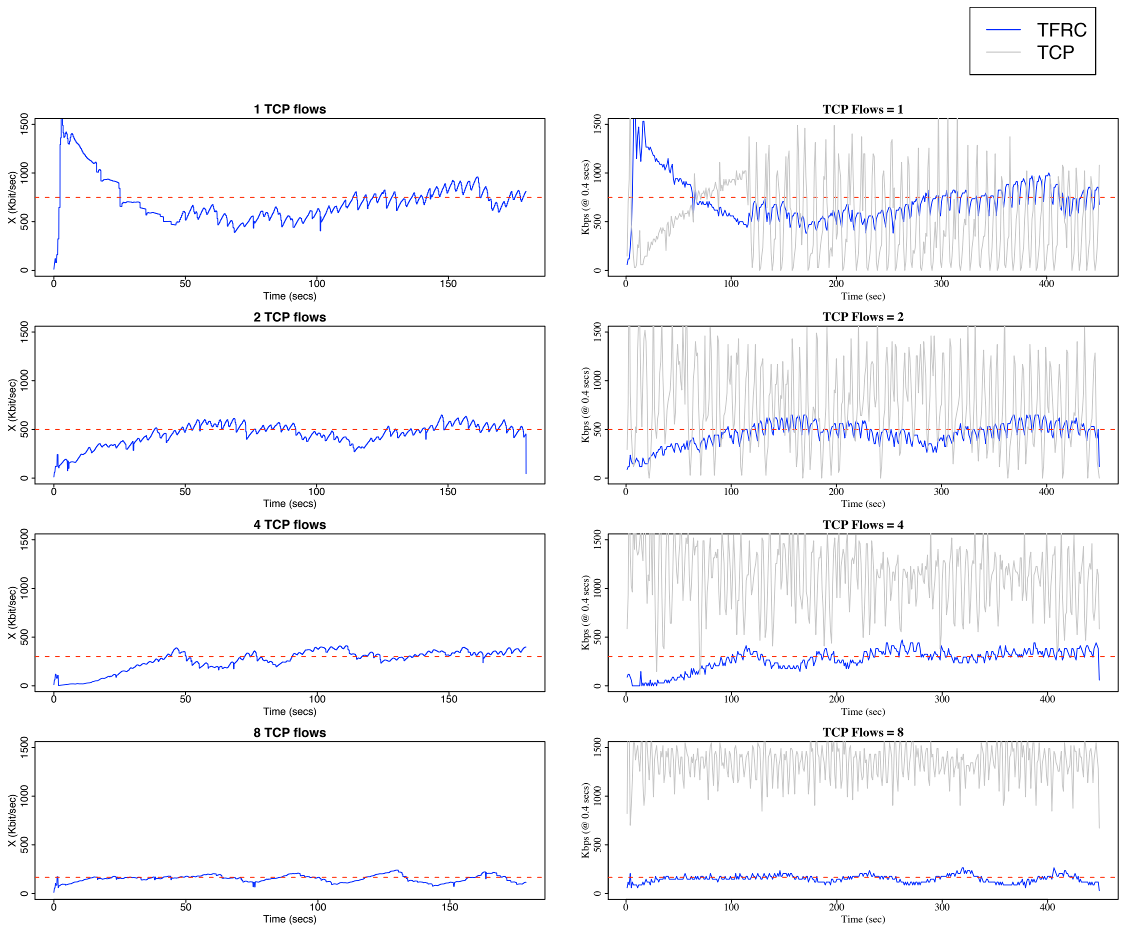
Figure A.3: Cumulative distribution of throughput variation on steady-state. *This corresponds to the scenarios shown in Section 5.2.1, and in particular Figure 5.3. It has been calculated using a time scale equal to the double of the RTT, with a minimum value of 150ms.*

A.2 Fairness with TCP flows



(a) TFRC X (b) TFRC and TCP Throughput

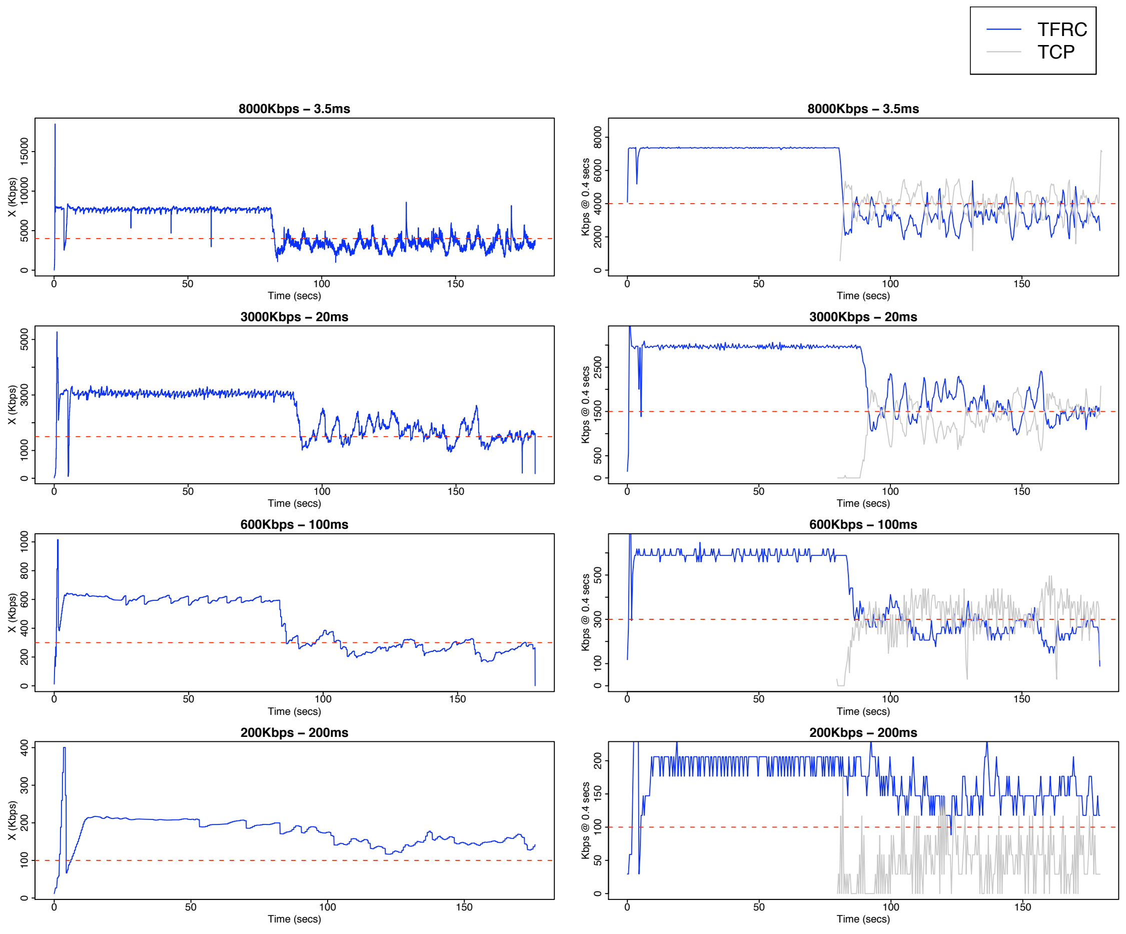
Figure A.4: Sending rate (X) and throughput of a TFRC flow competing with one TCP connection. *This corresponds to the scenarios shown in Section 5.2.2, and displays the connections seen in Figure 5.7*



(a) TFRC X (b) TFRC and TCP Throughputs (Cumulative)

Figure A.5: Sending rate (X) and throughput of a TFRC flow competing with several TCP connections. *This corresponds to the scenarios shown in Section 5.2.2, and displays the connections seen in Figure 5.10*

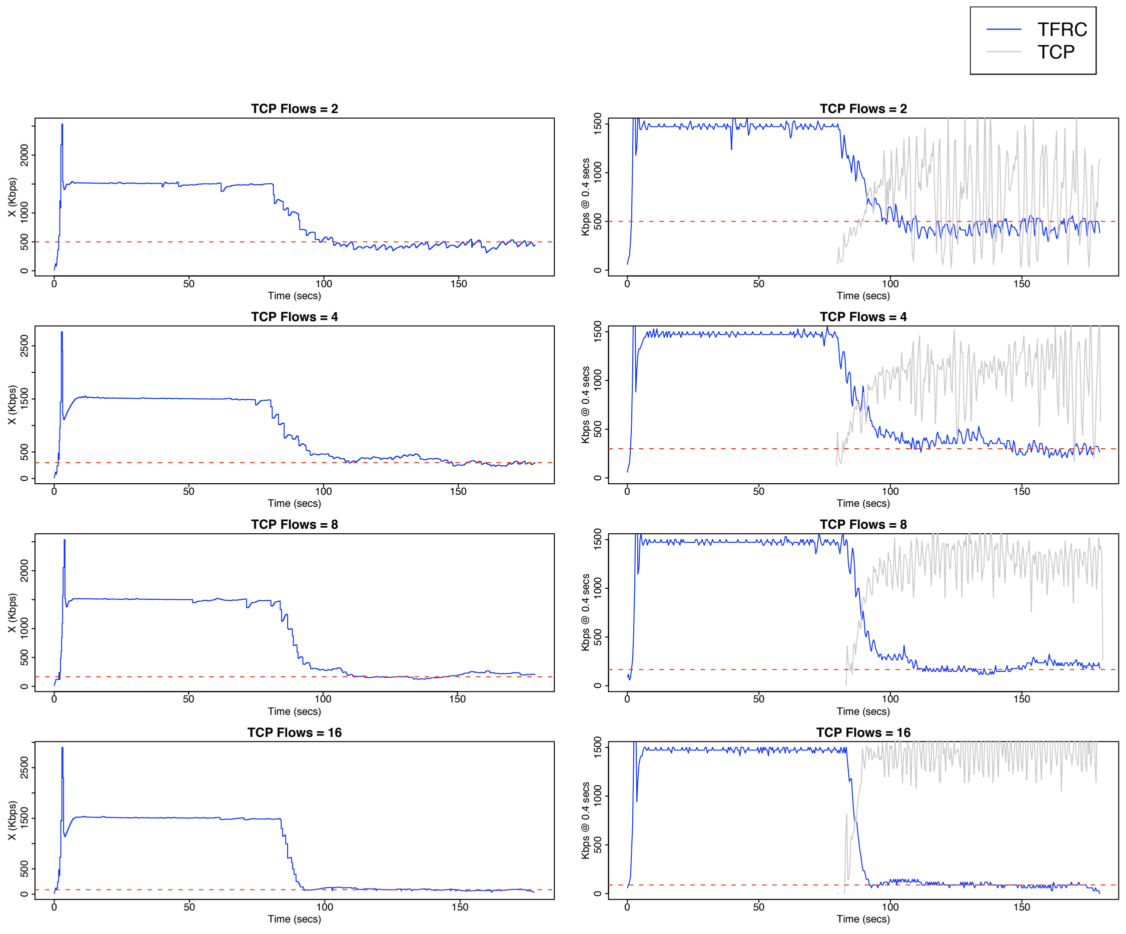
A.3 Responsiveness to a new TCP connection



(a) TFRC X (b) TFRC and TCP Throughput

Figure A.6: TFRC sending rate (X) and throughput with a new TCP connection. *This corresponds to the scenarios shown in Section 5.2.4, and displays the connections seen in Figure 5.13*

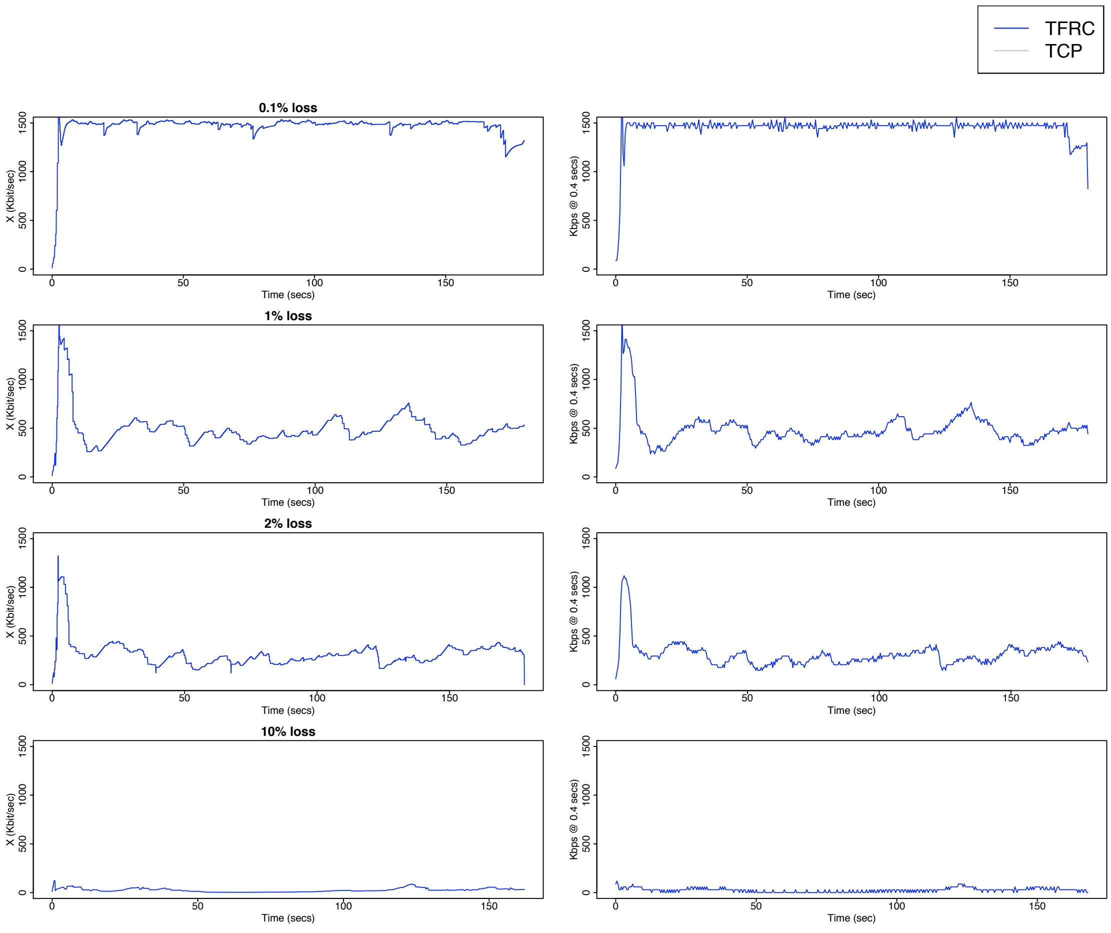
A.4 Responsiveness to reduced bandwidth



(a) TFRC X (b) TFRC and TCP Throughput (Cumulative)

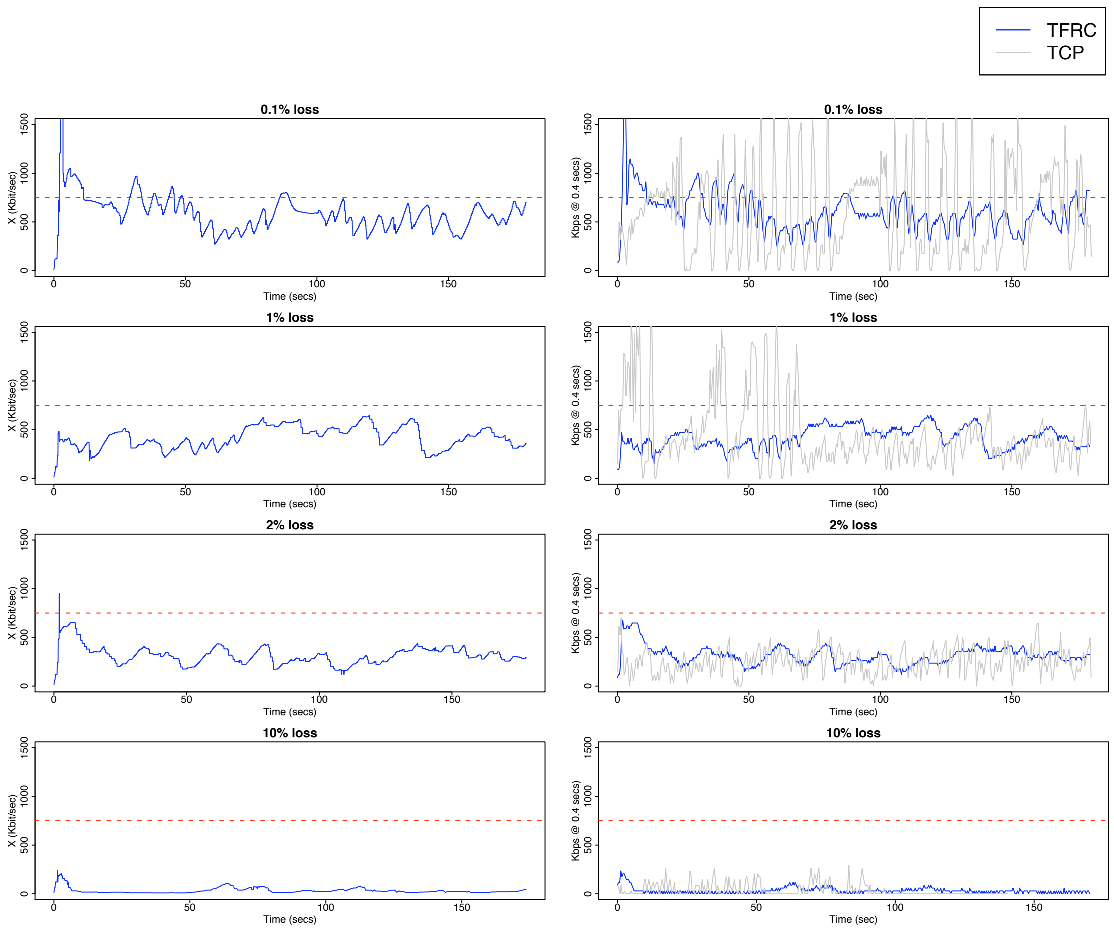
Figure A.7: TFRC sending rate (X) and throughput with new TCP connections. *This corresponds to the scenarios shown in Section 5.2.5. These connections are the same as seen in Figure 5.16*

A.5 Stability under loss



(a) TFRC X (b) TFRC Throughput

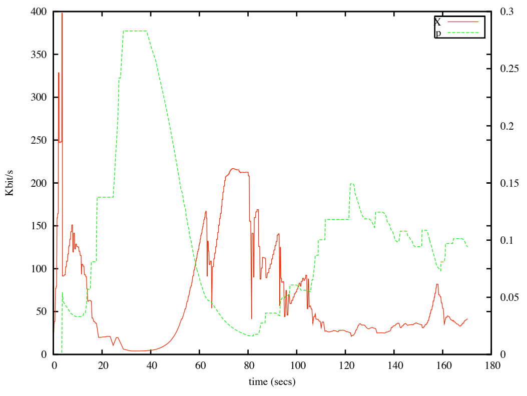
Figure A.8: TFRC sending rate (X) and throughput under loss. *This corresponds to the scenarios shown in Section 5.2.7. These connections are the same as seen in Figure 5.18*



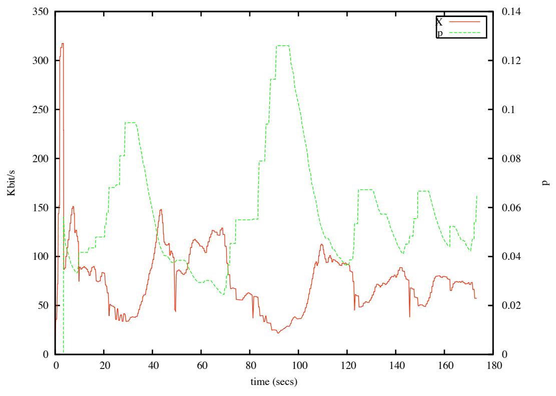
(a) TFRC X

(b) TFRC and TCP Throughput

Figure A.9: TFRC sending rate (X) and throughput with one TCP connection under loss. *This corresponds to the scenarios shown in Section 5.2.7. These connections are the same as seen in Figure 5.20*



(a) X and throughput for 1 TFRC and 1 UDP



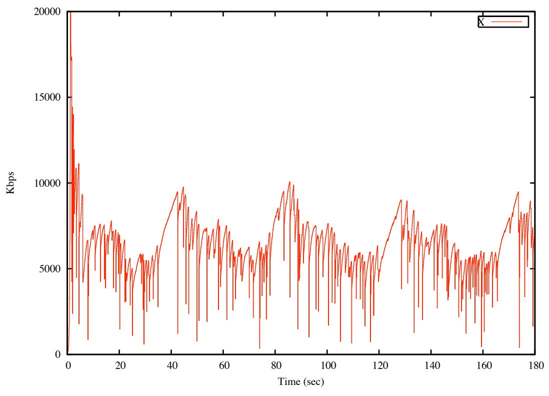
(b) X and P for 1 TFRC and 4 UDP

Figure A.10: TFRC sending rate (X) and loss event rate (p) with UDP bursty traffic. *This corresponds to the scenarios shown in Section 5.2.7. These connections are the same as seen in Figure 5.21*

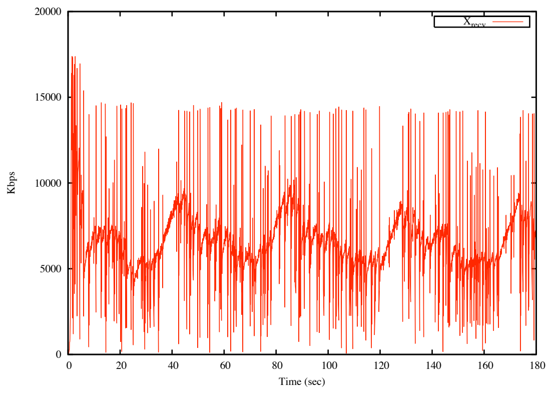
A.6 Internet Experiments

```
...
165.998590  P-   SEQ=36218       TS=161.130104  ...
166.000590  P-   SEQ=36219       TS=161.131104  ...
166.001854  P-   SEQ=36220       TS=161.133103  ...
166.003867  A+   X_RECV=939841   P=0.000202     ...
166.003971  P-   SEQ=36221       TS=161.134103  ...
166.214055  A+   X_RECV=7007     P=0.000202     ...
166.214301  P-   SEQ=36222       TS=161.136102  ...
166.214614  P-   SEQ=36223       TS=161.137102  ...
...
```

Figure A.11: TFRC connection between *alderon* and *mediapolku*: Sequence of packets received (*P*-) and feedback packets generated (*A+*) in *mediapolku. This sequence corresponds to Figure 5.24(b) seen in Section 5.3.*

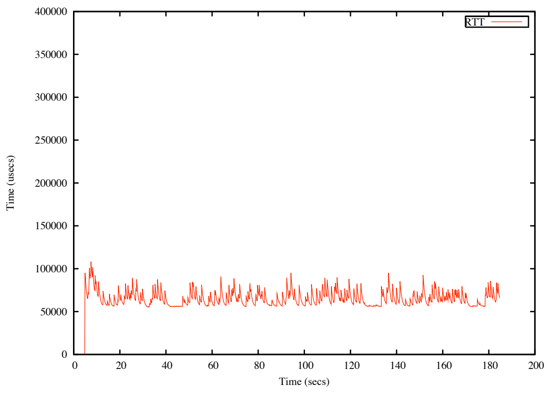


(a) X

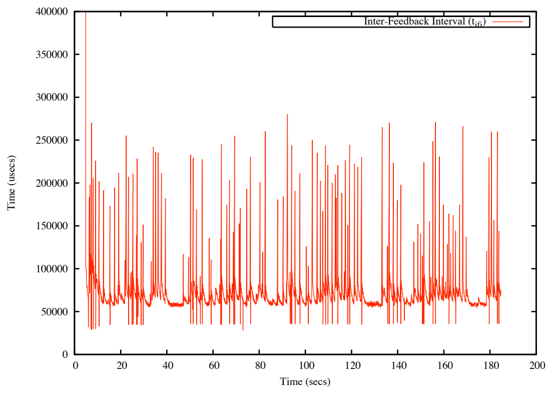


(b) X_{recv}

Figure A.12: Details of a TFRC connection between *curtis* and *mediapolku*: X and X_{recv}



(a) RTT



(b) t_{ifi}

Figure A.13: Details of a TFRC connection between *curtis* and *mediapolku*: delay and RTT.

Appendix B

TFRC Testing Strategies

This chapter describes the test suite used for the TFRC implementation. These tests verify the basic requirements of the protocol, and they could be used as a base for any testing of TFRC.

B.1 Testing Overview

In the following sections, I will describe the objectives and basic testing layout used.

B.1.1 Components Layout

The architecture for testing TFRC requires three major components: two end systems, performing as communicating elements, and an interface system, acting as communication medium and testing instrument.

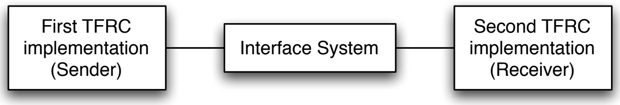

First TFRC implementation (Sender)
Interface System
Second TFRC implementation (Receiver)

Figure B.1: Testing layout

The interface system must be capable of sending arbitrarily packets to both TFRC implementations, and of receiving packets sent by the sender implementation, parsing them, computing metrics based on those packets, and transmitting them to the receiver. This system should also be able to discard or reorder selected packets, or to introduce some delay in the transmission process.

The testing framework is independant of the internal details of the TFRC implementation. Most of the tests have a zero knowledge of the system used, considering it as a black box and observing only the interchanged information.

However, some other tests have been proposed as optional, just in case there is enough information of the inner organization of the system.

B.1.2 Parts Tested

For the testing of TFRC we must stress the following main parts of the protocol:

- The basic interchage of information and interoperatibility (Section 4).
- The *RTT* estimation (Section 5.1).
- The TFRC sending rate calculation and inter-packet spacing (Section 5.3 and Section 5.4).
- The feedback mechanism (Section 5.2 and Section 6.1).
- The loss detection, loss history and associated loss rate estimation (Section 6.2).

The following sections describe a series of tests for these parts of the TFRC protocol.

B.2 Data Transport

The intention of these tests is to show that basic communication can be performed between the two TFRC implementations. In order to verify this, the system initialization must be tested as well as the basic operation of the system.

B.2.1 System Initialization

In these tests, tested the correct initialization of both end points will be tested, in particular the initial values for the fundamental parameters.

The sender initialization is tested first. There few specific tests for the sender, as the estimated *RTT*, R_i, and the RTO, t_{RTO}, have unspecified initial state. The sender initialization tests are limited to verifying that, for the first data packet sent, the sequence number is 0 (or an initial value chosen).

The test continues with the receiver initialization. The receiver is initialized when the first packet from the sender arrives. When the receiver receives an initial data packet at t_1, with sequence number 0, $i = 0$, and associated timestamp, $ts_0 = 0$, and *RTT*, $R_0 = 0$, some checks must be performed:

1. Verify that the receiver sends a feedback report at t_2, $t_2 = t_1 + t_{delay}$.
2. Verify that the value reported for t_{delay} matches the time elapsed between t_1 and t_2.
3. Verify that the value reported for X_{recv} is equal to the packet size.
4. Verify that the value reported for $t_{recvdata}$ is 0.
5. Verify that the value reported for p is 0.
6. Verify that the sender receives this feedback report.

These tests demonstrate that the receiver correctly processes the first data packet, and is initialized with appropriate state.

B.2.2 Basic Behavior

The purpose of these tests is to verify the basic correctness of the implementation of the TFRC transmission rules. These requirements can be verified at any point in a session.

The sender application must be able to handle the following cases

- Verify that the sequence number is incremented by one for each data packet sent.
- Verify that the timestamp is incremented for each data packet sent.
- Verify correct operation during sequence number wrap-around.
- Verify correct operation during timestamp wrap-around.

The receiver should also be verified to correctly handle the following edge cases:

- Verify correct operation during sequence number wrap-around.

- Verify correct operation during timestamp wrap-around.

B.3 Sender Behavior

In this section, in addition to the basic communicacion requirements of Section B.2, other features of the sender behavior must be verified. In particular, the RTT measurement, the feedback mechanism and the sending rate.

B.3.1 RTT Measurement

It should be verified that the initial conditions regarding the RTT are correctly initialized in both systems. This process is described in Section 4.3 of [39]. As the RTT known by the receiver is provided by the sender, a correct measure by the sender is decisive.

For the first test, we will tests how the receiver measures the RTT using the RTT sample provided in every feedback report. In order to do this, we will use a t_{delay} of $0ms$ in the receiver, and the interface system will initially simulate a fixed $10ms$ delay between both systems that will not change depending on the queue length.

The sender will initialize the system sending an initial data packet at t_0.

1. Verify that the first RTT reported by the sender is 0 (or null).
2. Verify that the first t_{delay} reported by the receiver is 0.
3. Verify that the first not-null RTT reported by the sender is equal to the RTT sample, 0.020.

If this test succeeded, the process should be repeated for some time, keeping the network delay and send rate constant.

1. Verify that the RTT reported by the sender, R_i, is constant, $R_i = 0.020$, provided that the network delay is constant.

If this test succeeds, the interface system will change the simulated delay to $20ms$ after the reception of a feedback report, at t_2. Next feedback report, received at t_3, $t_3 = t_2 + 0.020$, will provide a new R_{sample}.

Next data packet, sent at t_4, $t_4 > t_3$, and with sequence number j, will include a different RTT measure:

1. Verify that the value reported for RTT in the data packet with sequence number j is $22ms$.

For subsequent data packet sent after a feedback report is received, the RTT measure included must follow a known sequence:

1. Verify that the RTT included in the data packets evolves following the sequence $23.8ms$, $25.42ms$, $26.87ms$, $28.19ms$, $29.37ms$, $30.43ms$, $31.39ms$, $32.25ms$, $33.02ms$...
2. Verify that, for every successive data packet sent after a feedback report, the RTT included, R_i, is equal to the new RTT estimation, $R = 0,9 * R + 0.1 * (t_{now} - t_{recvdata})$.

B.3.2 Errors with Feedback Reports

The sender behavior should be verified for the absence of feedback reports. In order to verify this, the sender will be initialized, but the receiver will not send any feedback report. The sender has no knowledge of the RTT, so it must wait up to 2 seconds for a feedback report.

- Verify that the sender sends one packet per second for two seconds.
- Verify that the sender halves its sending rate every two seconds.
- Verify that the sender reaches a minimum sending rate of one packet every 64 seconds.

```
TIME: 0.000000 SEQ: 0
TIME: 1.000000 SEQ: 1
TIME: 2.000000 SEQ: 2
TIME: 4.000000 SEQ: 3
TIME: 6.000000 SEQ: 4
TIME: 10.000000 SEQ: 5
TIME: 14.000000 SEQ: 6
TIME: 22.000000 SEQ: 7
TIME: 30.000000 SEQ: 8
TIME: 46.000000 SEQ: 9
TIME: 62.000000 SEQ: 10
TIME: 94.000000 SEQ: 11
TIME: 126.000000 SEQ: 12
```

```
TIME: 190.000000 SEQ: 13
TIME: 254.000000 SEQ: 14
```

The behavior of the sender must change if no feedback information is received for some time, given by the current RTO.

In order to verify this, the sender must begin sending packets and the sender must respond with ACK packets, continuing with the normal operation until $t = 500ms$, when all feedback reports must be suspended again.

At this moment, we must take into account the last RTT, r, and sending rate, x, known by the sender when the last feedback report was received at the moment t.

- Verify that, if the sender does not receive a feedback report in four RTT, at $t + 4r$, it halves its sending rate, $X = x/2$.

This situation must be kept for some time, discarding all feedback reports, verifying that the sender spaces packets accordingly.

- Verify that the sender halves the sender rate every $4 * r$ seconds .
- Verify that the inter-packet interval is decreased until it reaches a minimum value of one packet every 64 seconds (as specified in Section 4.3 of [39])

B.3.3 TFRC Sending Rate

A TFRC implementation should be conformant to the throughput Equation 2.1. For $t_{RTO} = 4 * R$ and b = 1, the throughput equation is defined as:

$$X = \frac{s}{RTT * f(p)} \quad ; \quad f(p) = \sqrt{\frac{2 * p}{3}} + (12 * \sqrt{\frac{3 * p}{8}} * p * (1 + 32 * p^2)) \tag{B.1}$$

This formula depends on three parameters: the packet size (s), the loss event rate (p) and the *round-trip time* (RTT). Setting all three parameters to known values for a period of time should produce a known sending rate for the same period.

Alternatively, if we set two parameters with known values but we change the last one, we should be able to observe a conforming variation in the sending rate of the sending system.

To ensure this, some tests must be performed:

- Fixing the packet size to S, the loss event rate to P and the RTT to R for a time T, verify that the calculated sending rate X corresponds to these values.

```
Varying p:
s=1500 p=0.006 R=0.010   -> X=2.25006*10^6
s=1500 p=0.026 R=0.010   -> X=919512
s=1500 p=0.100 R=0.010   -> X=265515

Varying S:
s=1500 p=0.010 R=0.010   -> X=1.68498*10^6
s=4800 p=0.006 R=0.010   -> X=7.20021*10^6
s=9000 p=0.006 R=0.010   -> X=1.35004*10^7

Varying RTT:
s=1500 p=0.006 R=0.001   -> X=2.25006*10^7
s=1500 p=0.006 R=0.200   -> X=112503
s=1500 p=0.006 R=0.400   -> X=56251.6
```

- Fixing the packet size to S and the loss event rate to P for a time T, verify that a change of R in the RTT produces a change of X in the sending rate.
- Fixing the packet size to S and the RTT to R for a time T, verify that a change of P in the loss event rate produces a change of X in the sending rate.

We don't take into account a variation in the packet size, as it should be constant.

It must also be verified that the real sending rate matches the amount of data sent in a RTT, R. This can be tested in the following manner. The interface system must measure the amount of data interchanged between two feedback reports, received at times t_1 and t_2, provided that the receiver is sending one report per RTT, $t_2 - t_1 = R$.

Taking into account the first data packet sent after t_1, at t_d, $t_1 < t_d < t_2$, and the sending rate, X_d, and RTT, R_d, included:

- Verify that the amount of data send between t_d and t_2 matches the sending rate for that period, $X_d * R_d$.

In addition, some tests should be performed to verify that the sender conforms to the data reported by the receiver. Some checks could be:

- Verify that the sending rate is always less than twice the X_{recv} reported by the receiver.

B.3.4 Inter-Packet Interval Calculation

It must be verified that the inter-packet interval matches the current sending rate reported by the sender (see Section 4.6 of [39]).

In order to tests this, the sender must send packets to the receiver, and the interface system must log the times when these packets are sent. The packet size, S, is known, as it is the sending rate, X.

- Verify that the average space between packets in one RTT is S/X.

B.3.5 Slow-Start Algorithm

To perform this test, the system must be intialized. The sender will send packets and the reciever will answer with the corresponding feedback reports. The interface system must measure the amount of data sent between consecutive reports.

- Verify that the sender doubles the sending rate once per RTT (see Section 4.3 of [39]).
- Verify that the receiver reports a null value for the loss event rate, $p = 0$.

This situation can be sustained for some time, until t_1, where the last sending rate of the sender is X_1. After that, the receiver must report a loss event rate greater than 0, $p_1 > 0$, in order to test that the sender finishes the slow start phase.

- Verify that the first data packet sent after t_1, at t_2, includes a sending rate, X_2, with $X_2 < X_1$.

B.3.6 Oscillation Prevention

This is an OPTIONAL feature (specified in Section 4.5 of [39]).

To prevent oscillations in the sending rate, the sender keeps an estimate of the long-term RTT and sets its sending rate depending on how much the last RTT differs from this mean value.

For this testing scenario, the sender must be initialized and the receiver must send feedback reports on a regular basis. The RTT must be kept fixed at $20ms$ for at least two RTT (producing a internal value of 0.141421 for R_{sqmean}).

This must must be kept for some time until a feedback reports arrives at t_1. After this, the RTT must be halved, $RTT = 10ms$, and a new feedback report will arrive at t_2, providing an updated RTT sample.

If the sending rate at t_1 was X_1, and the sending rate reported in the first data packet sent after t_2 is X_2:

1. If the internal value of R_{sqmean} is known, verify that R_{sqmean} is updated and its value is 0.137279.
2. Verify that the sending rate X_2 is $X_1 * R_{sqmean}/\sqrt{R_{sample}} = X_1 * 1.37279$.

This test must be performed again, but this time the RTT must be doubled after t_1, setting $RTT = 40ms$.

1. If the internal value of R_{sqmean} is know, verify that R_{sqmean} is updated and its value is 0.147279.
2. Verify that the sending rate X_2 is $X_1 * R_{sqmean}/\sqrt{R_{sample}} = X_1 * 0.736396$.

These tests demonstrate that the receiver correctly calculates R_{sqmean} based on an expentially weighted moving average of the observed RTT values.

B.4 Receiver Behavior

In this section, some advanced characteristics of the receiver will be verified. In particular, the feedback mechanism and some aspects of the loss event rate calculation.

B.4.1 Feedback Mechanism

The receiver is expected to send a feedback report once per RTT. The following tests verify the accuracy of the information provided in a feedback report.

The sender begins transmission, with the delay through the interface system kept constant for at least two RTTs. It must be known the RTT, R_1, when last feedback report arrived at the sender, at t_1. The interface system must then record all packets, since t_1 to t_2, $t_2 - t_1 = R_1$.

1. Verify that a feedback report is sent at t_2.

2. Verify that the reported timestamp of last packet received, $t_{recvdata}$, matches the timestamp, t_{last}, of the last packet received, $t_{last} < t_2$.

The sending rate reported in the absence of losses must match the sending rate of the sender. Calculating the amount of data, d, sent in the interval between t_1 and t_2:

- Verify that the reported sending rate as seen by the receiver, X_{recv}, matches the sending rate of the sender, X, in the previous RTT.
- Verify that the sending rate as seen by the receiver, X_{recv}, matches the amount of data received since t_1, $X_{recv} = d/RTT$.

Feedback reports must be sent only when some data has been received since the last one was sent. In order to test this, all data packets must be discarded after a feedback report arrives at the sender, at t_1. If the last RTT known by the receiver was R_1, then a new feedback report would be expected at $t_2 = t_1 + R_1$.

- Verify that there is no feedback report sent at t_2.
- Verify that there is no feedback report sent while there are no data packets.

B.4.2 Loss Event Rate Estimation

In the next test it will be tested some aspect of Section 5.1 of [39].

The receiver is required to keep a history of packets that have been successfully transmited in order to detect a lost packet. Using this facility, the loss detection system must register in a loss history any lost packet, and any change in this history will modify the current loss event rate reported.

In these tests, the first implementation is made to transmit data packets, which are then received by the second implementation. The test instrument must discard some packets sent by the sender in order to simulate losses. An increment in the reported loss event rate will indicate that the loss has been detected. In the first test, a packet with sequence number i is discarded:

- Verify that, after the receiver system receives three packets with sequence numbers $i + 1$, $i + 2$ and $i + 3$, this is detected as a loss.
- Verify that the receiver sends a feedback report immediately after it detects the loss.
- Verify that the p reported has been increased.

If this tests succeded, the process should be repeated but with some packet reordering. In this case, no packet will be discarded, but some packets must be delayed by the test instrument, altering their natural sequence:

- Verify that, after the receiver system receives three packets with sequence numbers $i + 2$, i and $i + 1$, this is not detected as a loss.
- Verify that, after the receiver system receives four packets with sequence numbers $i + 3$, $i + 2$, $i + 1$ and i, the last packet is detected as a lost packet.
- Verify that the receiver sends a feedback report immediately after it detects the loss.
- Verify that the p reported has been increased.

In the next test it will be tested some aspect of loss computation (Section 5.2 of [39]).

Like in the previous test, the testing instrument should discard some packets and verify the behavior of the receiver under such circumstances. However, it must be taken into account the state of the sender before the loss is produced. In particular, the RTT and p must be known.

In this test, the interface system discards two packets in the same RTT, with sequence numbers i and $i + 1$.

- Verify that the receiver sends a feedback report after the packet with sequence number $i + 4$ has been received.
- Verify that there is an increment in the value of p reported by the receiver.
- Verify that, discarding two packets in a RTT, it has the same effect as a single loss on the loss measurement.

The test must be repeated with the same initial conditions. However, only one packet will be discarded this time:

- Verify that a feedback report is sent by the receiver when the loss is detected.
- Verify that the increment in the value of p is the same as in the previous measurement.

These tests could be extended using longer sequences of packets, dropping a packet when the sender has been transmitting for some time.

Bibliography

[1] Libjpeg, http://www.ijg.org, 2006.

[2] AccessGrid. http://www.accessgrid.org, 2006.

[3] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390, Internet Engineering Task Force, October 2002.

[4] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. In *Proceeding of ACM SIGCOMM'04 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (Portland, Oregon, USA, Aug. 30 – Sept. 3 2004)*, volume 34, pages 137–144, New York, NY, USA, September 2004. ACM Press.

[5] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for internet hosts. *ACM SIGCOMM Computer Communication Review*, 29(4):175–187, October 1999.

[6] Anindo Banerjea, Domenico Ferrari, Bruce A. Mah, Mark Moran, Dinesh Verma, and Hui Zhang. The tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Transactions on Networking (TON)*, 4(1):1–10, February 1996.

[7] Deepak Bansal and Hari Balakrishnan. Binomial congestion control algorithms. In *Proceedings IEEE INFOCOM 2001 Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (Anchorage, AK, USA, April 22-26 2001)*, volume 2, pages 631–640, 2001.

[8] Salman A. Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internel telephony protocol. Technical Report cs.NI/0412017, Columbia University, New York, December 2004.

[9] L. Berc, W. Fenner, R. Frederick, S. McCanne, and P. Stewart. RTP Payload Format for JPEG-compressed Video. RFC 2435, Internet Engineering Task Force, October 1998.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W . Weiss. An architecture for differentiated service. RFC 2475, Internet Engineering Task Force, December 1998.

[11] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309, Internet Engineering Task Force, April 1998.

[12] R. Braden, D. Clark, and S. Shenker. Integrated services in the Internet architecture: an overview. RFC 1633, Internet Engineering Task Force, 1994.

[13] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specifi cation. RFC 2205, Internet Engineering Task Force, September 1997.

[14] Kai Chen and Klara Nahrstedt. Limitations of equation-based congestion control in mobile ad hoc networks. In *Proceedings of 24th International Conference on Distributed Computing Systems (ICDCS'04) (Hachioji, Tokyo, Japan, March 24-26 2004)*, pages 756–761, March 2004.

[15] Kimberly C. Claffy, George C. Polyzos, and Hans-Wener Braun. Measurement considerations for assessing unidirectional latencies. *Journal of Internetworking*, 4(3), September 1993. UCSD Report CS92-252, SDSC Report GA-A21018.

[16] Mark Claypool and Jonathan Tanner. The effects of jitter on the perceptual quality of video. In *Proceedings of the 7th ACM MULTIMEDIA'99 International Conference on Multimedia (Part 2) (Orlando, Florida, USA, October 30 - November 05 1999)*, pages 115–118, New York, NY, USA, 1999. ACM Press.

[17] International Electrotechnical Commission. IEC 61834, Helical-scan digital video cassette recording system using 6,35 mm magnetic tape for consumer use (525-60, 625-50, 1125-60 and 1250-50 systems).

[18] G. Cote, B. Erol, M. Gallant, and F. Kossentini. H.263+: video coding at low bit rates. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(7):849–866, November 1998.

[19] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking (TON)*, 5(6):835–846, December 1997.

[20] D-ITG. http://www.grid.unina.it/software/ITG, 2006.

[21] Sally Floyd Deepak Bansal, Hari Balakrishnan and Scott Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. In *Proceedings of the ACM SIGCOMM'01 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (San Diego, CA, USA, August 27-31 2001)*, volume 31, pages 263–274, New York, NY, USA, August 2001. ACM Press.

[22] N. Feamster and H. Balakrishnan. Packet loss recovery for streaming video. In *Proceedings of the 12th International Packet Video Workshop (Pittsburgh, PA, USA, April 24-26 2002)*, April 2002.

[23] Nicholas G. Feamster. Adaptive delivery of real-time streaming video. Master's thesis, Massachusetts Institute of Technology, 2001.

[24] Stenio Fernandes, Djamel Hadj Sadok, and Ahmed Karmouch. Explicit feedback notification for transporting multimedia streaming flows over the Internet. In *Canadian Conference on Electrical and Computer Engineering 2005*, pages 1660– 1663, May 2005.

[25] S. Floyd, M. Handley, and J. Padhye. A comparison of equation-based and AIMD congestion control. Manuscript available at http://www.aciri.org/tfrc, May 2000.

[26] S. Floyd and E. Kohler. Profile for DCCP congestion control ID 2: TCP-like congestion control. Work in progress, 2005.

[27] S. Floyd, E. Kohler, and J. Padhye. Profile for DCCP congestion control ID 3: TFRC congestion control. Work in progress, March 2005.

[28] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (SACK) option for TCP. RFC 2883, Internet Engineering Task Force, July 2000.

[29] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking (TON)*, 7(4):458–472, August 1999.

[30] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications: the extended version. Technical Report TR-00-003, International Computer Science Institute (ICSI), Berkeley, California, USA, March 2000.

[31] R. Frederick. Experiences with real-time software video compression. In *Sixth International Workshop on Packet Video, Portland, Oregon, USA*, July 1994.

[32] Satoshi Futemma, Kenji Yamane, and Eisaburo Itakura. TFRC-based rate control scheme for real-time JPEG 2000 video transmission. In *IEEE Consumer Communications and Networking Conference*, January 2005.

[33] D. Le Gall. MPEG-1. *Communications of the ACM*, 34(4):47–58, April 1991.

[34] L. Gharai, C. Perkins, G. Goncher, and A. Mankin. RTP payload format for society of motion picture and television engineers (SMPTE) 292M video. RFC 3497, Internet Engineering Task Force, March 2003.

[35] L. Gharai and C. S. Perkins. RTP payload format for uncompressed video. RFC 4175, Internet Engineering Task Force, September 2005.

[36] Ladan Gharai. RTP profile for TCP-friendly rate control. Work in progress, July 2005.

[37] Andrew Gibson. The H.264 video compression. Master's thesis, Queen's University, Kingston, Ontario, Canada, September 2002.

[38] Andrei Gurtov and Jouni Korhonen. Effect of vertical handovers on performance of TCP-friendly rate control. *Mobile Computing and Communications Review*, 8(3):73–87, 2004.

[39] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP friendly rate control (TFRC): Protocol specification. RFC 3448, Internet Engineering Task Force, January 2003.

[40] M. Handley, C. Perkins, and E. Whelan. Session Announcement Protocol. RFC 2974, Internet Engineering Task Force, October 2000.

[41] Sven Hessler and Michael Welzl. An empirical study of the congestion response of RealPlayer, Windows MediaPlayer and Quicktime. In *Proceedings of 10th IEEE International Symposium on Computers and Communications (ISCC'2005) (La Manga del Mar Menor, Cartagena, Spain, June 27-30 2005)*, pages 591–596. IEEE Computer Society Press, 2005.

[42] O. Hodson and C. S. Perkins. Robust-audio tool, version 4. http://www-mice.cs.ucl.ac.uk/multimedia/software/rat/, 2006.

[43] D. Hoffman, G. Fernando, V. Goyal, and R. Civanlar. RTP payload format for MPEG1/MPEG2 video. RFC 2250, Internet Engineering Task Force, January 1998.

[44] ITU-T. Recommendation H.261 - video codec for audiovisual services at p x 64 kbit/s. *Geneva, Switzerland*, March 1993.

[45] ITU-T. Recommendation G.114 - one-way transmission time. *Geneva, Switzerland*, February 1996.

[46] ITU-T. Recommendation H.263 - video coding for low bit rate communication. *Geneva, Switzerland*, March 1996.

[47] C. Huitema J. Rosenberg, R. Mahy. TURN: traversal using relay NAT. Work in progress, July 2004.

[48] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures and Protocols (Stanford, CA, USA, August 16-18 1988)*, pages 314–329, New York, NY, USA, August 1988. ACM Press.

[49] Shudong Jin, Liang Guo, Ibrahim Matta, and Azer Bestavros. A spectrum of TCP-friendly window-based congestion control algorithms. *IEEE/ACM Transactions on Networking (TON)*, 11(3):341–355, 2003.

[50] Y. Kikuchi, T. Nomura, S. Fukunaga, Y. Matsui, and H. Kimata. RTP Payload Format for MPEG-4 Audio/Visual Streams. RFC 3016, Internet Engineering Task Force, November 2000.

[51] K. Kobayashi, A. Ogawa, S. Casner, and C. Bormann. RTP Payload Format for DV (IEC 61834) Video. RFC 3189, Internet Engineering Task Force, January 2002.

[52] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. To appear in Proceedings of ACM SIGCOMM, 2006.

[53] Georgios Parissidis Kostas Katrinis and Bernhard Plattner. A comparison of frameworks for multimedia conferencing: SIP and H.323. In *Proceedings of the 8th IASTED International Conference on Internet Multimedia Systems and Applications (IMSA 2004) (Kauai, Hawai, USA, August 17-19 2004)*, 2004.

[54] Charles Krasic, Kang Li, and Jonathan Walpole. The case for streaming multimedia with TCP. In *Proceedings of the 8th International Workshop on Interactive Distributed Multimedia Systems (IDMS '01) (Lancaster, UK, September 4-7 2001)*, volume 2158, pages 213–218, London, UK, 2001. Springer-Verlag.

[55] Nikolaos Laoutaris and Ioannis Stavrakakis. Intrastream synchronization for continuous media streams: A survey of playout schedulers. *IEEE Network Magazine*, 16(3), May 2002.

[56] A. Legout and E. Biersack. Beyond TCP-Friendliness: A new paradigm for end-to-end congestion control. In *Reliable Multicast Research Group (RMRG) meeting (Pisa, Italy, June 5-9 1999)*, June 1999.

[57] Mingzhe Li, Mark Claypool, Robert Kinicki, and James Nichols. Characteristics of streaming media stored on the web. *ACM Transactions on Internet Technology (TOIT)*, 5(4):601–626, November 2005.

[58] Michael R. Macedonia and Donald P. Brutzman. MBone provides audio and video across the internet. *IEEE Computer*, 27(4):30–36, 1994.

[59] M. Masry and S. Hemami. An analysis of subjective quality in low bit rate video. In *Proceedings of the International Conference on Image Processing (Thessaloniki, Greece, November 7-10 2001)*, volume 1, pages 465–468, 2001.

[60] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *Proceedings of the ACM SIGCOMM'96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (Palo Alto, California, August 28-30 1996)*, pages 281–291, New York, NY, USA, 1996. ACM Press.

[61] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. RFC 2018, Internet Engineering Task Force, 1996.

[62] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, July 1997.

[63] S. McCanne and S. Floyd. UCB/LBNL network simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns, 1999.

[64] Steven McCanne and Van Jacobson. vic : A flexible framework for packet video. In *Proceedings of the ACM MULTIMEDIA'95 International Conference on Multimedia (San Francisco, California, USA, November 05-09 1995)*, pages 511–522, New York, NY, USA, 1995. ACM Press.

[65] MPEG. MPEG video draft proposal. *ISO/IEC JTC1/SC2/WG11 ISO-11172-2*, August 1991.

[66] J. Nichols, M. Claypool, R. Kinicki, and M. Li. Measurements of the congestion responsiveness of windows streaming media. Technical Report TR-04-07, CS at Worcester Polythechnic Institute, March 2004.

[67] Telecommunication Standardization Sector of ITU. Packet-base multimedia communication systems. ITU-T Recommendation H.323, September 1999.

[68] T. Ott, J. Kemperman, and M. Mathis. The stationary distribution of ideal TCP Congestion Avoidance. Manuscript available at http://networks.ecse.rpi.edu/natun/papers/tcp-equn.ps, 1996.

[69] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 28(4):303–314, October 1998.

[70] V. Paxson. End-to-end internet packet dynamics. *IEEE/ACM Transactions of Networking (TON)*, 7(3):277–292, June 1999.

[71] V. Paxson and M. Allman. Computing TCP's retransmission timer. RFC 2988, Internet Engineering Task Force, November 2000.

[72] Vern E. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD dissertation, University of California, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, 1998. UMI Order No. GAX98-0332.

[73] C. Perkins, L. Gharai, T. Lehman, and A. Mankin. Experiments with delivery of HDTV over IP networks. In *Proceedings of the 12th International Packet Video Workshop (PV'2002) (Pittsburgh, PA, USA, April 24-26 2002)*, 2002.

[74] C. S. Perkins and L. Gharai. UltraGrid: A high definition collaboratory. NSF award 0230738, November 2002. http://ultragrid.east.isi.edu/.

[75] T. Phelan. Datagram congestion control protocol (DCCP) user guide. Work in progress, April 2005.

[76] J. Postel. User Datagram Protocol. RFC 768, Internet Engineering Task Force, August 1980.

[77] J. Postel. Internet Protocol. RFC 791, Internet Engineering Task Force, September 1981.

[78] J. Postel. Transmission Control Protocol. RFC 793, Internet Engineering Task Force, September 1981.

[79] A. Puri and A. Eleftheriadis. MPEG-4: An object-based multimedia coding standard supporting mobile applications. *ACM Mobile Networks and Applications*, 3(1):5–32, June 1998.

[80] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, Internet Engineering Task Force, September 2001.

[81] Reza Rejaie, Mark Handley, and Deborah Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99) (New York, NY, USA, March 21-25 1999)*, volume 3, pages 1337–1345. IEEE Press, 1999.

[82] I. Rhee, V. Ozdemir, and Y. Yi. TEAR: TCP emulation at receivers – flow control for multimedia streaming. NCSU technical report (draft), Department of Computer Science, NCSU, Raleigh, NC, USA, April 2000.

[83] Injong Rhee and Lisong Xu. Limitations of equation-based congestion control. *ACM SIGCOMM Computer Communication Review*, 35(4):49–60, October 2005.

[84] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, January 1997.

[85] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[86] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489, Internet Engineering Task Force, March 2003.

[87] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 3550, Internet Engineering Task Force, July 2003.

[88] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, Internet Engineering Task Force, April 1998.

[89] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement (IMC '04) (Taormina, Sicily, Italy, October 2004)*, pages 41–54, New York, NY, USA, 2004. ACM Press.

[90] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: the TCP/UDP bandwidth measurement tool, May 2005. http://dast.nlanr.net/Projects/Iperf/.

[91] C. Villamizar and C. Song. High performance TCP in ANSNet. *ACM SIGCOMM Computer Communication Review*, 24(5):45–60, October 1994.

[92] M. Vojnovic and J. Boudec. Some observations on equation-based rate control. In *Proceedings of the 17th International Teletraffic Congress (ITC-17) (Salvador da Bahia, Brazil, September 24-28 2001)*, pages 173–184, 2001.

[93] Milan Vojnovic and Jean-Yves Le Boudec. On the long-run behavior of equation-based rate control. *ACM SIGCOMM Computer Communication Review*, 32(4):103–116, October 2002.

[94] Naoki Wakamiya, Masaki Miyabayashi, Masayuki Murata, and Hideo Miyahara. MPEG-4 video transfer with TCP-friendly rate control. In *Proceedings of the 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS'01) (Chicago, IL, USA October 29 - November 1 2001)*, pages 29–42, London, UK, 2001. Springer-Verlag.

[95] Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM. Special issue on digital multimedia systems*, 34(4):30–44, April 1991.

[96] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via TCP: An analytic performance study. In *Proceedings of the 12th ACM MULTIMEDIA'04 International Conference on Multimedia (New York, NY, USA, October 10-16 2004)*, pages 908–915, New York, NY, USA, 2004. ACM Press.

[97] Jorg Widmer, Catherine Boutremans, and Jean-Yves Le Boudec. End-to-end congestion control for TCP-friendly flows with variable packet size. *ACM SIGCOMM Computer Communication Review*, 34(2):137–151, April 2004.

[98] Jörg Widmer, Robert Denda, and Martin Mauve. A survey on TCP-friendly congestion control (extended version). Technical Report TR-01-002, Department for Mathematics and Computer Science, University of Mannheim, February 2001.

[99] Jörg Widmer and Mark Handley. Extending equation-based congestion control to multicast applications. In *Proceeding of ACM SIGCOMM'01 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (San Diego, CA, USA, August 27-31 2001)*, pages 275–285, New York, NY, USA, 2001. ACM Press.

[100] Y. Richard Yang, Min Sik Kim, and Simon S. Lam. Transient behaviors of TCP-friendly congestion control protocols. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 41(2):193–210, February 2003.

[101] Lixia Zhang, Stephen Deering, and Deborah Estrin. RSVP: A new resource ReSerVation protocol. *IEEE Network Magazine*, 7(5):8–18, September 1993.