Equation-Based Congestion Control for Unicast and Multicast Data Streams

Inauguraldissertation zur Erlangung
des akademischen Grades eines
Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Dipl. Wirtsch.–Inf. Jörg Carsten Widmer
aus Heidelberg

Mannheim, 2003

Dekan:	Professor Dr. Herbert Popp, Universität Mannheim
Referent:	Professor Dr. Wolfgang Effelsberg, Universität Mannheim
Korreferent:	Professor Dr. Jean-Yves Le Boudec, EPFL, Lausanne/Schweiz

Tag der mündlichen Prüfung: 15. Mai 2003

Abstract

We believe that the emergence of congestion control mechanisms for relatively-smooth congestion control for unicast and multicast traffic can play a key role in preventing the degradation of end-to-end congestion control in the public Internet, by providing a viable alternative for multimedia flows that would otherwise be tempted to avoid end-to-end congestion control altogether. The design of good congestion control mechanisms is a hard problem, even more so for multicast environments where scalability issues are much more of a concern than for unicast.

In this dissertation, equation-based congestion control is presented as an alternative form of congestion control to the well-known TCP protocol. We focus on areas of equation-based congestion control which were not yet well understood and for which no adequate solutions existed. Starting from a unicast congestion control mechanism which in contrast to TCP provides smooth rate changes, we extend equation-based congestion control in several ways. We investigate how it can work together with applications which can only operate in a very limited region of available bandwidth and whose rate can thus not be adapted to the network conditions in the usual way. Such a congestion control mechanism can also complement conventional equation-based congestion control in regimes where available bandwidth is too low for further rate reduction. When extending unicast congestion control to multicast, it is of paramount importance to ensure that changes in the network conditions anywhere in the multicast tree are reported back to the sender as quickly as possible to allow the sender to adjust the rate accordingly. A scalable feedback mechanism that allows timely congestion feedback in the face of potentially very large receiver sets is one of the contributions of this dissertation. But also other components of a congestion control protocol, such as the rate increase/decrease policy or the slow-start mechanism, need to be adjusted to be able to use them in a multicast environment. Our resulting multicast congestion control protocol was implemented in a simulation environment for extensive protocol testing and turned into a library for the use in real-world applications. In addition, a simple video transmission tool was built for test purposes that uses this congestion control library.

Acknowledgements

First and foremost, I wish to thank Prof. Effelsberg for his guidance and advice throughout the three years of my PhD. He gave me all possible freedom in the pursuit of my research and provided the best work atmosphere one could possibly want.

A number of others played an important role in shaping this work, in particular my colleagues Martin Mauve, Holger Füßler, Jürgen Vogel, Dirk Farin, Robert Denda, and Matthias Transier. It was also a pleasure to work together with Thomas Fuhrmann; much of this work is integrated in the thesis. I am grateful for all of their much appreciated ideas, support, and feedback. I further thank Jan Peter Damm and Jürgen Hötzel for their contributions to this work in the form of diploma theses and Christina Boucher for her work as a research assistant.

I am deeply thankful to Mark Handley, Sally Floyd, and Jitendra Padhye for the excellent cooperation over the years and to Scott Shenker and Joachim Beer for making my stays at ICIR/ICSI possible. Thanks also to Luigi Rizzo and Brian Whetten for helpful discussions.

I am indebted to Betty Weyerer, who went out of her way to help by last-minute proofreading papers and providing guidance through the jungle of university administration, and to Walter Müller for eventually preventing my PC from erasing the hard drive every other week.

Many thanks to Prof. Jon Crowcroft for making possible a very interesting and helpful short term research stay at University College London, to Tristan Henderson and Sokrates Varakliotis for the valuable discussions, and to COST264 for providing the funding.

My work benefited significantly from a research stay at EPFL, Switzerland. I particularly would like to thank Prof. Jean-Yves Le Boudec and Catherine Boutremans for the cooperation. It was a pleasure to work with them and I was delighted to find in Catherine not only an excellent colleague but a friend.

Finally, I wish to thank my parents and my sister for supporting me in every imaginable way.

Contents

List of Figures

List of Tables

Abbreviations

ACK	Acknowledgement
AD	Administrative Domain
AIAD	Additive Increase, Additive Decrease
AIMD	Additive Increase, Multiplicative Decrease
ALF	Application Level Framing
CoV	Coefficient of Variation
DSL	Digital Subscriber Line
DVMRP	Distance Vector Multicast Routing Protocol
ECN	Explicit Congestion Notification
EWMA	Exponentially Weighted Moving Average
FEC	Forward Error Correction
FIFO	First In First Out
FTP	File Transfer Protocol
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IP(v4/v6)	Internet Protocol (version 4 / version 6)
ISM	Inter-packet Space Modulation
ISP	Internet Service Provider
LAN	Local Area Network
MIAD	Multiplicative Increase, Additive Decrease
MIMD	Multiplicative Increase, Multiplicative Decrease
MOSPF	Multicast Open Shortest Path First
MTU	Maximum Transmission Unit
NACK	Negative Acknowledgement

PCC	Probabilistic Congestion Control
PIM(-DM/-SM)	Protocol Independent Multicast (Dense Mode/Sparse Mode)
RED	Random Early Detection
RTCP	RTP Control Protocol
RTO	Retransmission Timeout Value
RTP	Real-Time Transport Protocol
RTT	Round-Trip Time
SACK	Selective Acknowledgement
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TDACK	Triple Duplicate ACK
TFMCC	TCP-Friendly Multicast Congestion Control
TFRC	TCP-Friendly Rate Control Protocol
TTL	Time To Live
UDP	User Datagram Protocol
WAN	Wide Area Network

Chapter 1

Introduction

In the Internet, data packets are sent from a source to the destination via intermediate routers. A router stores a packet until it can be transferred to the next router along the path. This continues in succession until a packet ultimately reaches its destination. When the demand for bandwidth exceeds available network resources, network paths can get congested. If the short-term packet arrival rate is higher than the maximum packet processing rate or the outgoing link bandwidth of a router, packets are temporarily stored in a buffer. A router can sustain high packet arrival rates for short periods of time, where the duration of the period depends on the buffer size. However, when the buffer is full, the router has to discard packets. These packets already consumed resources on the way to the router where they are dropped. A network should be designed in a way that the network load is reduced when excessive packet drop rates occur to ensure that the network operates in a regime of reasonable efficiency.

The end-to-end argument [SRC84] is one of the main design principles of the Internet. It states that “functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level”. In a networking context, this translates to the requirement that as much functionality as possible should be implemented at the edges of the network or in the end hosts rather than within the network. For this reason, congestion in the Internet is resolved by simply discarding excessive packets at congested routers. This very simple reaction to congestion is complemented by more sophisticated congestion control and congestion avoidance mechanisms in the end hosts.

The dominant transport protocol in the Internet is the Transmission Control Protocol (TCP), constituting 83% of all IP packets and carrying roughly 90% of the overall traffic volume [MC00].

TCP is a reliable transport protocol that provides window-based congestion control as well as flow control. The current stability of the Internet largely depends on this end-to-end congestion control mechanism. The ‘sending rate’ of a TCP flow is controlled by a congestion window which is reduced in size when the flow experiences packet drops. The congestion window limits the number of packets inserted into the network in times of congestion. It is halved for every window of data containing a packet drop, and increased by roughly one packet per window of data otherwise. It can be shown that this Additive Increase Multiplicative Decrease (AIMD) mechanism leads to stable network conditions [CJ89].

Some form of end-to-end congestion control of best-effort traffic is obviously required to avoid the congestion collapse of the Internet [FF99]. While TCP congestion control is appropriate for applications such as for example bulk data transfer, halving the sending rate in response to a single congestion indication is unnecessarily severe for some applications. TCP’s abrupt changes in the sending rate have been a key impediment to the deployment of TCP for emerging applications such as streaming multimedia, as they noticeably reduce the user-perceived quality [TZ99b]. A number of further issues arise when using TCP for these applications, such as TCP’s reliability mechanism, but here we will focus on congestion control.

Equation-based congestion control was first proposed in [MF97]. In contrast to TCP’s additive increase and multiplicative decrease of the congestion window which indirectly controls the sending rate, the rate of a flow is directly adapted to the current network conditions. This rate is determined using a control equation that explicitly gives the maximum acceptable sending rate as a function of the recent loss event rate and the round-trip time. Since equation-based congestion control can provide smoother changes of the sending rate than the algorithm currently implemented in TCP it seems to be a promising candidate for the aforementioned applications.

For traffic that competes with TCP, the appropriate control equation can be derived from a model for the steady-state sending rate of TCP. As a consequence, over a timescale of many round-trip times such equation-based congestion control should achieve roughly the same throughput as TCP under the same network conditions. Any form of congestion control that is significantly more aggressive is likely to harm TCP traffic when competing for resources on the same bottleneck link and is therefore not recommendable for use in the Internet (without additional protection of other traffic). A slightly more formal description of this requirement is given in the definition of TCP-friendliness, discussed in a later section of this dissertation.

TCP-Friendly Rate Control (TFRC) [FHPW00a] is an example of an equation-based unicast congestion control mechanism intended for applications that require a smoother, more predictable transmission rate than TCP can achieve. Basically TFRC works as follows:

- The receiver measures the packet loss rate and feeds this information back to the sender in feedback packets which are sent at regular intervals.
- The sender uses timestamps contained in the packet headers to calculate the round-trip time to the receiver.
- The sender uses the control equation to derive an acceptable transmission rate from the measured loss rate and round-trip time.
- The sender's transmission rate is then adjusted to match the calculated transmission rate.

The behavior of TFRC is determined to a large degree by the measurement mechanisms used to estimate the loss rate and the round-trip time. A stable and yet fair TFRC sending rate can only be achieved if the methods of measurement do not introduce unnecessary noise but react to changes in the level of congestion sufficiently fast. Sender and receiver functionality do not have to be split up exactly as described above. For example, it is also possible to have the receiver apply the control equation and report the desired sending rate back to the sender.

1.1 Problem Statement

TFRC has emerged as an adequate unicast congestion control mechanism for applications such as streaming media. Yet, there exist a variety of applications for which equation-based congestion control in general is a suitable solution but TFRC cannot be used directly.

While for most applications reducing the sending rate is the appropriate response to network congestion, there are cases where the data rate of an individual flow is determined almost exclusively by the application and can only be adjusted to a small degree to the network conditions. Data traffic of networked computer games or audio and video transmission that are already at the lowest possible quality level are typical examples. For game traffic, a certain loss of available bandwidth can be compensated for by techniques such as dead reckoning [DG99], but there is generally a certain minimum bandwidth requirement beyond which applications become unusable. For this class of applications there are only two acceptable states: either a flow is *on* and the sender transmits (at least) at the minimum data rate determined by the application or it is *off* and no data is transmitted at all. While some form of congestion control is necessary to ensure the safety of the network, it is not possible to achieve it by simply setting the application's sending rate to a TCP-friendly rate.

A different application area for which TFRC is not designed is multicast. IP multicast provides data distribution for one-to-many and many-to-many communication. Hosts participating in a so-called multicast session are organized in a tree topology. Data packets are duplicated at the branching points of the tree (the multicast routers) so that only a single copy of the original packet travels over each branch. Multicast greatly reduces the number of data packets compared to sending a copy of the packet from the sender to each of the receivers via unicast.

Particularly the applications for which equation-based congestion control is a suitable rate control mechanism are often applications designed for one-to-many communication. Audio and video conferencing and distributed computer games may involve hundreds or even thousands of participants. With potentially very large multicast receiver sets, the efficient use of network and end system resources is of paramount importance in the design of an end-to-end transport protocol working on top of IP multicast.

A multicast congestion control mechanism must also take into account that due to potentially very large receiver sets it may no longer be possible to request feedback from all the receivers participating in a multicast session. While timely receiver feedback is an important factor for the responsiveness of the mechanism to changes in the network conditions, too many feedback messages in a short time interval can overwhelm the sender so that it is not capable of processing all those messages. Thus, feedback control for multicast congestion control needs to address two important issues:

- The mechanism must prevent a feedback implosion in case the multicast group is very large and responses from all or most of the receivers would overwhelm the sender. It has to guarantee that a sufficient number of feedback messages is suppressed.
- At the same time, the feedback mechanism must ensure that feedback from the "right" receivers is not suppressed. Those receivers are the ones that should determine the sending rate of the session and their feedback is crucial for the fairness of the congestion control mechanism.
- Moreover, it is necessary to ensure than any additional delay imposed to avoid feedback implosion does not adversely affect the fairness toward competing protocols.

There are strong reasons to design elements of a multicast congestion control mechanism to be more conservative than their unicast counterparts. Since a multicast protocol can affect the network conditions on many links of the multicast tree, it can do much more harm than a unicast protocol if not designed with great care. In particular, this applies to the feedback mechanism, the increase/decrease policy, and slow-start (the rate increase algorithm used at the startup time of a new session).

On the other hand, all the recipients of a multicast session benefit from the transmitted packets whereas a unicast transmission is only targeted at a single receiver. One could argue that a multicast session is of higher value than a unicast session and should therefore achieve a higher sending rate. It is necessary to take these considerations into account when defining fairness and TCP-friendliness in a multicast context.

1.2 Contributions

In this dissertation, equation-based congestion control is extended in several ways. We investigate how it can work together with applications whose sending rate cannot be adapted to the network conditions in the usual way, how to extend unicast congestion control to multicast, and how to ensure that multicast congestion control scales to large groups of receivers. The resulting multicast congestion control protocol is developed in a simulation environment for extensive protocol testing. We further implement it as a library for the use in real-world applications and present a simple video transmission tool based on this library that proves the applicability of our algorithms in practice.

Probabilistic Congestion Control

With Probabilistic Congestion Control (PCC) we present a congestion control scheme for non-adaptable flows. These types of flows carry data at a rate determined by the application. They cannot be adapted to the level of congestion in the network in any way other than by suspending the entire flow. Existing congestion control mechanisms that adjust the sending rate are thus not viable for non-adaptable flows.

Instead of striving for TCP-friendliness for each single network flow, PCC works by suspending individual flows in a way that the aggregation of all non-adaptable flows on a given link behaves in a TCP-friendly manner. The decision about suspending a given flow is made by means of random experiments. If a random experiment fails, the corresponding flow is suspended for a configurable amount of time until it may resume its transmission. This type of congestion control can complement conventional congestion control in the regime where rate adaptation is no longer possible.

Scalable Multicast Feedback Mechanisms

Equation-based congestion control, as a rate control component of a transport protocol, is very amenable to an extension to multicast. To ensure that on no link of the multicast tree the sending rate exceeds the TCP-friendly rate, the sender needs to continuously obtain feedback from the receiver or receivers experiencing the worst network conditions. The main complexity of the design of a multicast congestion control protocol lies in the feedback mechanism.

With receiver sets of perhaps several thousand receivers it is critical to ensure that the sender gets timely feedback from the receivers without being overwhelmed by an excessive number of feedback messages. To this end, a feedback suppression mechanisms is used. Generally, time is divided into feedback rounds, and at the start of each feedback round, each receiver sets a randomized timer. If the receiver hears feedback from another receiver that makes it unnecessary for it to send its own feedback, it cancels its timer. Otherwise, a feedback message is sent when the timer expires.

In this dissertation, we analyze three prototypical suppression algorithms, with particular focus on the suppression characteristics in face of an inaccurate group size estimation. We further improve upon the most promising of these algorithms, exponential feedback suppression, in case feedback of some extreme value of the group is needed. We discuss two orthogonal methods to improve the quality of the feedback (i.e., how close the best reported value is to the true optimal value of the group). If no information is available about the distribution of values at the receivers, a safe method to obtain better feedback is to modify the suppression mechanism to allow the sending of *important* feedback messages even after the first feedback was given (i.e., feedback messages cannot be suppressed by feedback reporting less optimal values). We specify exact bounds for the expected increase in feedback messages for a given improvement in feedback quality. If more information about the distribution of feedback values is available or certain worst-case distributions are very unlikely, it is furthermore possible to bias the feedback timer. The better the feedback value, the earlier the feedback is sent, thus suppressing later feedback with less optimal values. The modified suppression mechanism and the feedback biasing can be used in combination to further improve the feedback process.

Multicast Congestion Control

Based on TFRC and the aforementioned feedback suppression mechanism, we design a TCP-Friendly Multicast Congestion Control Protocol (TFMCC). TFMCC is an equation-based single-rate multicast congestion control mechanism intended to scale to groups of several thousand receivers. The challenges in the design of TFMCC lie in scalable round-trip time measurements, appropriate feedback suppression, and in ensuring that feedback delays in the control loop do not adversely affect fairness towards competing flows. The key component of end-to-end multicast congestion control schemes is the feedback control mechanism which largely determines the overall protocol behavior.

With a single-rate congestion control protocol that adapts to the slowest receiver, the sending rate may be very low in a large group of heterogeneous receivers. In order to enable the application to remove receivers that unduly impair protocol performance, it is necessary to provide the application with information about the degree of heterogeneity of the group. This information is (partially) available to the congestion control mechanism through the receiver feedback process and can be made available to the application. With an approximate distribution of TCP-friendly rates of the receivers, the application can then calculate the expected improvement in the sending rate when certain receivers are removed from the group and take appropriate actions.

TFMCC is extensively evaluated through analysis and simulation. To further demonstrate that TFMCC is a suitable congestion control mechanism for streaming media, it is implemented as a library and integrated into a simple multicast video transmission tool for MPEG.

1.3 Structure of the Dissertation

After a brief introduction to TCP congestion control and TCP fairness in Chapter 2, we give an overview of the TCP-friendly Rate Control protocol (TFRC) in Chapter 3. TFRC forms the basis for the more advanced mechanisms proposed in this dissertation. In Chapter 4, we discuss a probabilistic congestion control mechanism (PCC) for non-adaptable flows, which is not based on rate adaptation of a single flow but on the control of traffic aggregates.

A short introduction to IP multicast as well as a definition of TCP-friendliness for multicast transmission is given in Chapter 5. In Chapter 6 we analyze desirable characteristics of feedback mechanisms for multicast congestion control in detail. We further investigate to what extent existing mechanisms are suitable for the task and propose a new feedback control scheme that specifically allows to favor feedback from certain receivers, namely the ones that are important for setting the sending rate.

In Chapter 7 we introduce the TCP-Friendly Multicast Congestion Control protocol (TFMCC), an equation-based congestion control mechanism that extends the TFRC protocol from the unicast to the multicast domain. We describe the protocol mechanism in detail and demonstrate the behavior of the protocol by means of extensive simulations. Furthermore, we give an overview of the TFMCC library that was developed in this context and present a video transmission tool which is based on this library.

Finally, we conclude the dissertation with a summary and directions for future research in Chapter 8.

Chapter 2

Congestion Control Fundamentals

TCP [Pos81, Ste94] is a connection-oriented unicast transport protocol, used to transport data for most common Internet applications such as web browsers, e-mail programs, and FTP clients. It is the most widely used transport protocol in the Internet. Alternative congestion control mechanisms intended for use in the Internet have to work in an environment dominated by TCP congestion control and should therefore be TCP-compatible. In this chapter, we give a brief overview of the TCP protocol, discuss how TCP's long-term throughput can be modeled, and discuss forms of end-to-end congestion control different from TCP congestion control.

2.1 TCP Protocol Description

TCP is used to transport a stream of bytes from one end host to another. Since the underlying routing protocol only provides the routing of data packets, the byte stream has to be divided into segments and these segments are then sent in individual packets.

Before data is transmitted, TCP establishes a full-duplex connection between the communicating end hosts. Segments are acknowledged by the TCP receiver to ensure end-to-end reliability.[1] In case packets are dropped by a network layer below TCP their loss is detected through the missing acknowledgements, and they are retransmitted by the TCP sender. In addition to reliability, TCP offers flow control as well as congestion control. Flow control allows a receiver to slow down the sender to prevent it from sending segments faster than the receiver can process them, while

[1] In fact, TCP uses a sequence number in units of bytes and the receiver acknowledges the next byte to be transmitted by the sender.

congestion control protects the network from excessive network load. Flow control is done by means of a sliding window mechanism, where the amount of data the sender is allowed to send is limited by the flow control window. The window is maintained at the receiver and its size is advertised in the acknowledgements. When segments are received from the sender, the size of the available window is reduced, and as the receiving application reads data from TCP's receive buffer the window size increases again.

2.1.1 Self-clocking Characteristics of TCP

The temporal spacing of the data packets depends on the link capacity (i.e., the packet frequency increases with the available bandwidth). In equilibrium, TCP only allows sending of a packet when another packet is acknowledged. The ACKs from the receiver arrive at the TCP sender at the same pace at which the data packets leave the sender. The algorithm is self-clocking and provides a very elegant way to limit the number of outstanding packets in the network.

The time it takes a packet to travel from sender to receiver and the time it takes the ACK to travel back to the sender is called the round-trip time (RTT). During this time the sender has to be able to continually send data at the link speed (or the fair share of the bandwidth when competing with other flows) to be able to fully utilize the available bandwidth.

2.1.2 TCP Congestion Control

In addition to the flow control window, TCP maintains a congestion window that further limits the number of outstanding unacknowledged data packets in the network. The congestion window size is decreased when congestion is detected and increased in the absence of congestion. The TCP sender may only transmit the minimum of the flow control window and the congestion window before it has to wait for new ACKs from the receiver. Both, the flow control window and the congestion window are measured in bytes.

On start-up, TCP performs a *slow-start* to quickly reach a fair share of the available network capacity without overwhelming the network with packets. Let s be the maximum segment size of a TCP connection in bytes. During slow-start, each acknowledgement increases the size of the congestion window by s and thus the size of the congestion window doubles every round-trip time (exponential increase). Slow-start ends either after a certain window size threshold ($ssthresh$) is reached or after the first packet loss occurs [Jac88].

After the slow-start phase, TCP uses an *additive increase multiplicative decrease mechanism* (AIMD) to detect additional available bandwidth and to react to congestion, where congestion is indicated by packet loss. Upon reception of an ACK the TCP sender increases the congestion window size $cwnd$ by $s^2/cwnd$ [APS99], resulting in an increase of approximately one segment per round-trip time. If a data packet is not acknowledged by the receiver within a time span of the retransmission timeout value, the sender assumes severe congestion, the congestion window is reduced to one segment, and the unacknowledged packet is retransmitted. TCP then reenters the slow-start phase. The retransmission timeout value has a significant impact on TCP performance and is therefore continuously adapted to variations in the TCP round-trip time.

In addition to timeouts, a second mechanism is used to detect packet loss. Upon packet arrival the TCP receiver acknowledges the last segment that arrived in order. If intermediate segments are lost, the segment before the lost segments will be acknowledged when new segments arrive at the receiver. Therefore, packet loss and packet reordering result in duplicate acknowledgements. Four acknowledgements for the same sequence number, called a triple duplicate ACK (TDACK), are a strong indication that one or more segments were lost. As a consequence, the sender reduces the congestion window to half of its previous size and starts retransmitting the segments it assumes were lost. Figure 2.1 shows a typical example of the evolvement of TCP's congestion window over time.

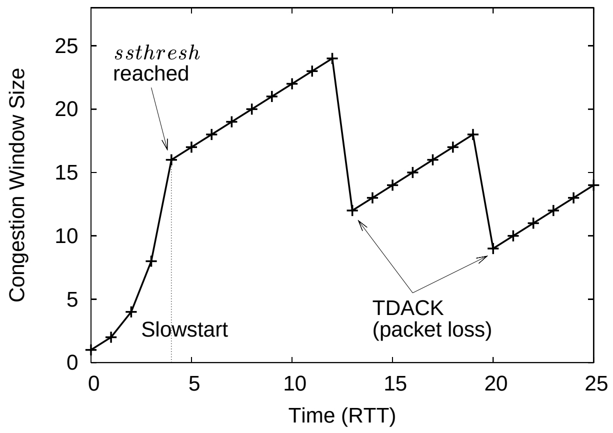


Figure 2.1: Evolvement of the TCP congestion window over time

Since the first TCP implementations, TCP has been improved in several ways. Today, different versions of TCP are in use, the most common being TCP Reno and TCP Sack. An overview of some of the different TCP flavors and their implications on protocol performance is given in [FF96].

2.2 Modeling TCP Throughput

The throughput of TCP depends mainly on the parameters round-trip time t_{RTT}, retransmission timeout value t_{RTO}, segment size s, and packet loss rate p. Using these parameters, an estimate of TCP's throughput can be derived. A very basic model that gives an upper bound on TCP's steady-state throughput R_{TCP} is given in [FF99]. Following the analysis in the appendix of the paper, let us assume completely periodic packet loss with one drop per $1/p$ packets and a congestion window size of W whenever a packet loss occurs. The loss causes a reduction of the window size to $\frac{1}{2}W$. No other losses occur for the next $1/p$ packets, and during this time, the congestion window size increases by one segment per RTT. After $\frac{1}{2}W$ RTTs, the congestion window reaches its original size W again. The number of packets delivered in one increase-decrease cycle is at least

$$\left(1+\frac{1}{2}\right)\cdot\left(\frac{W}{2}\right)^2 = \frac{3W^2}{8}$$

Since there occurs exactly one loss per cycle, loss rate and window size are related as follows:

$$p \leq \frac{8}{3W^2} \qquad \text{or} \qquad W \leq \sqrt{\frac{8}{3p}}$$

The throughput R_{TCP} can then be computed as the amount of data transmitted in a cycle over the duration of the cycle.

$$R_{TCP} \leq \frac{s\cdot\frac{3}{8}W^2}{\frac{1}{2}W\cdot t_{RTT}} = \frac{s\cdot\frac{3}{4}W}{t_{RTT}} = \sqrt{\frac{3}{2}}\,\frac{s}{t_{RTT}\sqrt{p}}$$

If an ACK is sent only for every other data packet, the common practice in current implementations, the throughput is only:

$$R_{TCP} \leq \sqrt{\frac{3}{4}}\,\frac{s}{t_{RTT}\cdot\sqrt{p}} \tag{2.1}$$

This analysis is only valid as long as window size reductions are caused by TDACKs. When a high packet drop rate causes frequent TCP timeouts, the model overestimates the rate a TCP flow can achieve. Similar models for TCP throughput can be found for example in [OKM96, MSMO97].

Equation (2.2), presented in [PFTK00], gives a more complex model of TCP throughput; b is the number of packets acknowledged by each ACK, and W_m is the maximum size of the congestion window. Unlike the previous model, this complex model takes rate reductions due to TCP timeouts into account and models TCP more accurately in a network environment with high loss rates.

$$R_{TCP} = \min\left(\frac{W_m \cdot s}{t_{RTT}}, \frac{s}{t_{RTT}\sqrt{\frac{2bp}{3}} + t_{RTO}\min\left(1, 3\sqrt{\frac{3bp}{8}}\right)p(1+32p^2)}\right) \tag{2.2}$$

For most applications it suffices to set t_{RTO} to a multiple of t_{RTT} and to ignore the impact of b and W_m, resulting in the following approximation:

$$R_{TCP} = \frac{s}{t_{RTT}\left(\sqrt{\frac{2p}{3}} + \left(12\sqrt{\frac{3p}{8}}\right)p\,(1+32p^2)\right)} \tag{2.3}$$

Figure 2.2 shows that the complex model (PFTK) and the simple model (SQRT) arrive at a similar throughput estimate in terms of packets per RTT at low loss rates, while the throughput of the complex model decreases faster at higher loss event rates. In a highly congested network environment, nearly all of the TCP congestion window reductions are caused by timeouts. Thus, the complex model is far more accurate under such conditions.

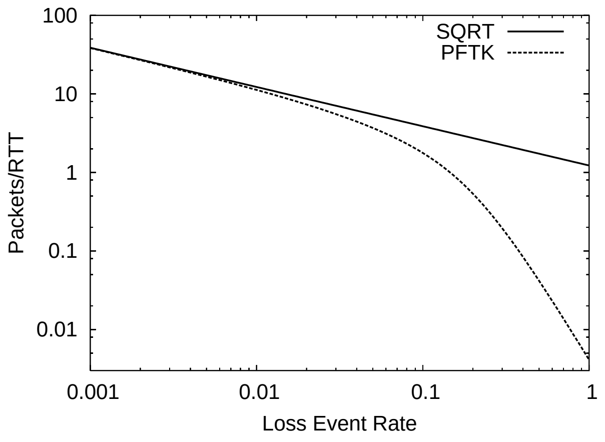


Figure 2.2: Comparison of the SQRT model and the PFTK model

Among other assumptions, both models require that the round-trip time and the loss rate be independent of the estimated rate (i.e., they do not take into account that a changing TCP rate can affect the round-trip time and the loss rate). Nevertheless, as measurements and simulations have shown, they do model TCP throughput surprisingly well even when these assumptions are not or only partly met.

2.3 TCP-Friendliness

In [FF99], non-TCP flows are defined as TCP-friendly when "their long-term throughput does not exceed the throughput of a conformant TCP connection under the same conditions". We prefer to use a slightly more restrictive definition of the term TCP-friendliness. The definition used throughout this dissertation focuses on the effect that a non-TCP flow has on competing TCP flows rather than on the throughput of the non-TCP flow.

TCP-Friendliness for Unicast: A unicast flow is considered *TCP-friendly* (or *TCP-compatible*) 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.

A flow that sends at a lower than the TCP-friendly rate (e.g., because of limited demand) would still be characterized as TCP-compatible. However, if such a flow were considerably less aggressive, it could encounter starvation when competing with TCP traffic in a FIFO queue.

Similar to unicast TCP-friendliness we define multicast TCP-friendliness by requiring unicast TCP-friendliness on each path in the multicast distribution tree.

TCP-Friendliness for Multicast: A multicast flow is defined as *TCP-friendly* when for each sender-receiver pair, the multicast flow 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.

There is an ongoing debate on the correct definition of the term TCP-friendliness for multicast. An alternative to the definition given above is to allow multicast flows to use a greater amount of bandwidth than unicast flows, since they serve multiple receivers. In [WS98], the term *bounded fairness* is introduced to define a situation where the following equation holds true:

$$a \cdot r_{TCP} \leq r \leq b \cdot r_{TCP} \tag{2.4}$$

where r is the rate of the multicast flow on the bottleneck link, r_{TCP} is the rate a TCP flow would have under the same conditions, and a as well as b are functions of the number of receivers of the flow. For $b = 1$ the two definitions are equivalent. While the latter approach is perfectly valid, here we will use the definition that is more rigid in the protection of competing TCP flows.

2.4 Non-AIMD Congestion Control

The models for long-term TCP throughput presented in Section 2.2 can be used to design a rate-based congestion control mechanism where the sending rate is adjusted to the TCP-friendly rate predicted by the model. An equation-based congestion control mechanism does not modify the sending rate in the same way as TCP's AIMD congestion control does. In particular, such a mechanism does not reduce its sending rate by half in response to a single congestion indication. Given that the stability of the current Internet rests on AIMD congestion control, a proposal for non-AIMD congestion control requires justification in terms of its suitability for the global Internet.

As discussed in [FF99, FHPW00a], the principal threat to the stability of end-to-end congestion control in the Internet comes not from flows using alternate forms of TCP-compatible congestion control, but from flows that do not use any end-to-end congestion control at all. Furthermore, it has been shown that preserving some form of "fairness" against competing TCP traffic does not require a reaction as drastic as halving of the rate in response to a single congestion indication. Ultimately, only large scale deployment in the Internet can show what impact non-TCP congestion control mechanisms will have on the stability of the Internet, but the papers mentioned above indicate that such forms of congestion control are indeed viable and TCP-compatible.

2.4.1 Design Space for Congestion Control Mechanisms

The following characteristics are desirable for an end-to-end congestion control protocol:

- Fairness: the protocol should compete fairly with other flows, in particular TCP flows.
- Stability: a sending rate that remains fairly stable even with an increased level of noise in the network.
- Responsiveness: fast response to permanent changes in network conditions.
- Wide adaptive range: the ability to sustain performance over a wide range of network conditions and cope with heterogeneity in the network.
- Performance: no performance penalty, resulting from low computational overhead and little control traffic; no underutilization of resources caused by the congestion control scheme.

Not all of these characteristics can be achieved at the same time; for example a tradeoff exists between the responsiveness and the stability of the protocol. When the sending rate is more stable and less sensitive to noise, it will be less responsive to changes in the network conditions. A similar tradeoff exists between a wide adaptive range and protocol responsiveness. On the other hand, fairness and good performance should be consistently achieved by the protocol.

In the following we will briefly outline the possible design choices for congestion control mechanisms different from TCP. A more detailed classification of congestion control schemes can be found in [WDM01].

Window-Based versus Rate-Based

One possible classification criterion for congestion control is whether the offered network load is adapted based on a congestion window or on the transmission rate. Algorithms that belong to the window-based category use a congestion window at the sender or at the receiver(s) to ensure TCP-friendliness. The sender is allowed to transmit packets only when free slots in the window are available. The size of the congestion window is increased in the absence of congestion indications and decreased when congestion occurs.

Rate-based congestion control achieves TCP-friendliness by dynamically adapting the transmission rate according to some network feedback mechanism that indicates congestion. It can be subdivided into simple AIMD schemes and model-based congestion control. Simple AIMD schemes mimic the behavior of TCP congestion control. The resulting rate shows the typical short term sawtooth-like behavior of TCP. Alternatives to the increase by one, decrease by half policy of TCP have been proposed in [YL00, BB01]. There, this simple policy is extended to more general increase/decrease policies, while preserving the TCP-friendliness of the mechanism. Model-based congestion control uses a TCP model such as the ones presented earlier in this chapter. By adapting the sending rate to the average long-term throughput of TCP, model-based congestion control can produce much smoother rate changes. Such schemes do not mimic TCP's short-term sending rate but are still TCP-friendly over longer time scales. However, the congestion control mechanism does not resemble TCP congestion control, and great attention has to be paid to the rate adjustment mechanism to ensure fair competition with TCP or other flows. A more detailed comparison of AIMD and equation-based congestion control is given in [FHP00].

End-to-End versus Router-Supported

End-to-end congestion control mechanisms are designed for networks that do not provide any additional router mechanisms to support the protocols. The network only provides feedback as to the level of congestion (e.g., by dropping or marking packets [RF99]) but the end-systems are relied upon to take appropriate action. Consequently, end-to-end congestion control mechanisms can be further separated into sender-based and receiver-based approaches. In sender-based approaches the sender uses information about the network congestion and adjusts the rate or window size to achieve TCP-friendliness, while receivers only provide feedback. Receiver-driven congestion control is usually used for multicast in combination with layered congestion control, as described below. Here, the receivers decide whether to subscribe to or unsubscribe from layers to increase or decrease the receive rate.

The design of congestion control protocols and particularly the fair sharing of resources can be considerably facilitated by placing intelligence in the network (e.g., in routers or separate agents). Congestion control schemes that rely on additional functionality in the network are called router-supported. Particularly, multicast protocols can benefit from additional network functionality such as feedback aggregation, hierarchical round-trip time measurements, management of (sub-)groups of receivers, or modification of the routers' queuing strategies. *Generic Router Assist (GRA)* [CST00], for instance, is a recent initiative that proposes general mechanisms located at routers to assist transport control protocols, which would greatly ease the design and implementation of effective congestion control protocols. Furthermore, with router assistance it is possible to enforce conformant behavior without having to rely on the cooperation of the end-systems. While this is a promising approach (see for example [KHR02]), the Internet currently lacks adequate congestion control enforcement and protocol support by the routers. Since the router infrastructure in the Internet is slow to change, there is no alternative to end-to-end congestion control for the time being.

Single-Rate versus Multi-Rate Congestion Control

In single-rate congestion control, the sender adjusts the sending rate, and the receiver or the receivers obtain the data at the same rate. In contrast, multi-rate congestion control allows different receive rates for the participating receivers. Multi-rate congestion control is only defined in multicast environments. The data is distributed over multiple multicast session, and receivers join the appropriate number of sessions with respect to their current network conditions. Whether to chose multi-rate or single-rate transport is a fundamental decision in the design of a multicast congestion control mechanism. When the receiver set is very heterogeneous, a multi-rate approach is better able to cater to the different requirements of the receivers. At the same time it is only applicable if the data to be transmitted can be divided into layers, adjusting the number of layers does not allow an adaptation of the receive rate as fine-grained as single-rate congestion control does (since a receiver is either subscribed to a layer or not), and it is usually much more complex to build. A typical example is video multicast where each additional layer of data improves the visual quality for the receiver (see for example [MJV96, Kuh01]). Single-rate congestion control mechanisms can be used for a wide range of applications but it is necessary to ensure that adjusting the sending rate to a single receiver does not excessively hurt performance for the other receivers. The responsibility to remove receivers from a single-rate session clearly lies with the application, but a congestion control protocol can help to improve the decision process by providing information to the application.

Chapter 3

TCP-Friendly Rate Control

3.1 Introduction

For most unicast flows that want to transfer data reliably and as quickly as possible, the best choice is simply to use TCP directly. In contrast, the TCP-friendly Rate Control protocol (TFRC) [FHPW00a] is intended for applications that require a smoother, more predictable transmission rate than TCP can achieve. The cost of this smoothness is a more moderate response to transient changes in congestion. TFRC typically runs over UDP but the congestion control mechanism would work for any connectionsless protocol.

TFRC is an equation-based unicast congestion control mechanism. The primary goal of equation-based congestion control is not to aggressively find and use available bandwidth, but to maintain a relatively steady sending rate while still being responsive to congestion. Thus, several of the design principles of equation-based congestion control can be seen in contrast to the behavior of TCP:

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

Additional design goals for equation-based congestion control for unicast traffic include:

- The receiver should report feedback to the sender at least once per round-trip time if it has received any packets in that interval.

- If the sender has not received feedback after several round-trip times, then the sender should reduce its sending rate, and ultimately stop sending completely. This prevents the sending of unnecessary packets in case of a severe network failure where (almost) all packets are dropped.

3.2 Related Work

We first review related work from the literature. The unreliable unicast congestion control mechanisms closest to TCP maintain a congestion window which is used directly [JE96] or indirectly [ROY00] to control the transmission of new packets. We believe that since [JE96] uses the TCP mechanisms directly, the resulting behavior will be very similar to TCP congestion control as described in the previous chapter. In the TEAR protocol (TCP Emulation at the Receivers) from [ROY00], the receiver emulates the congestion window modifications of a TCP sender, but then translates them from a window-based to a rate-based congestion control mechanism. The receiver maintains an exponentially weighted moving average of the congestion window, and divides this by the estimated round-trip time to obtain a TCP-friendly sending rate.

In [YL00], TCP's increase by one, decrease by half behavior is extended to arbitrary increase and decrease factors. The resulting protocol is called general AIMD (GAIMD). The authors analyze the impact of these factors on the long-term sending rate of GAIMD and examine which relationship between the factors results in TCP-friendliness. The so-called binomial congestion control mechanisms proposed in [BB01] allow for even higher flexibility in the increase/decrease policy. The congestion window is increased inversely proportional to a power k of the current window size ($cwnd \leftarrow cwnd + \alpha\, cwnd^{-k}$) and decreased multiplicatively proportional to a power l of the current window ($cwnd \leftarrow cwnd - \beta\, cwnd^{l}$). Again, the authors show which combinations of parameters result in TCP-friendliness and give an in-depth analysis of some specific parameter combinations (e.g., $k = 0, l = 1$ for TCP, $k = \frac{1}{2}, l = \frac{1}{2}$ for SQRT where increase and decrease are proportional to the square root of the current window, etc.).

A class of unicast congestion control mechanisms one step removed from those of TCP are those that use additive increase, multiplicative decrease (AIMD) in some form, but do not apply AIMD to a congestion window. The Rate Adaption Protocol (RAP) presented in [RHE99] is a simple AIMD scheme for unicast flows. Each data packet is acknowledged by the receiver. The ACKs are used to detect packet loss and infer the round-trip time. When the protocol experiences congestion it halves the sending rate. In periods without congestion, the sending rate increases by one packet per round-trip time, thus mimicking the AIMD behavior of TCP. The decisions on rate increase or decrease are made once per round-trip time. To provide additional fine-grained delay-based congestion avoidance, the ratio of a short-term round-trip time average and a long-term round-trip time average is used to modify the inter-packet gap between consecutive data packets. These fine-grained rate adjustments result in a smoother sending rate. RAP achieves rates similar to TCP in an environment where TCP experiences no or few timeouts since RAP's rate reductions resemble TCP's reaction to triple duplicate ACKs. However, RAP does not take timeouts into account and is therefore more aggressive when TCP's throughput is dominated by timeout events.

Unlike many other schemes, the Loss-Delay Based Adaption Algorithm LDA+ [SW00] does not devise its own feedback mechanism to control the sending rate but relies solely on the feedback messages provided by the Real Time Transport Control Protocol [SCFJ96]. While LDA+ is essentially an AIMD congestion control scheme, it uses some interesting additional elements. The increase and decrease factors for AIMD are dynamically adjusted to the network conditions. An estimate of the bottleneck bandwidth is obtained using packet pairs.[1] The amount of additive increase is then determined as the minimum of three independent increase factors to ensure that (1) flows with a low bandwidth can increase their rate faster than flows with a higher bandwidth, (2) flows do not exceed the estimated bottleneck bandwidth, and (3) flows do not increase their bandwidth faster than a TCP connection would do. If receivers report loss the sending rate is decreased by multiplying by the factor $(1 - \sqrt{l})$, where l is the loss rate. Additionally, the rate is reduced at most to the rate given by the TCP model as described in Equation 2.2. Using the maximum of the AIMD rate and the equation rate may result in a long-term average that exceeds the average rate of the two separate schemes and can be more aggressive than TCP. only for unicast used for

Equation-based congestion control is probably the class of TCP-compatible congestion control mechanisms most removed from the AIMD mechanisms of TCP. In [TZ99a] the authors describe a simple equation-based congestion control mechanism for unicast, unreliable video traffic. The receiver measures the RTT and the loss rate over a fixed multiple of the RTT. The sender then uses this information, along with the version of the TCP response function from [MF97], to control the sending rate and the output rate of the associated MPEG encoder. The main focus of [TZ99a] is not the congestion control mechanism itself, but the coupling between congestion control and error-resilient scalable video compression.

[1]With packet pairs, the time interval between the receipt of two packets that were sent back-to-back is used as a hint of the current maximum rate for a flow.

The TCP-Friendly Rate Control Protocol (TFRCP) [PKT99] uses an equation-based congestion control mechanism for unicast traffic where the receiver acknowledges each packet. At fixed time intervals, the sender computes the loss rate observed during the previous interval and updates the sending rate using the TCP response function described in [PFTK98]. Since the protocol adjusts its send rate only at fixed time intervals, the transient response of the protocol is poor at lower time scales. In addition, computing loss rates at fixed time intervals makes the protocol vulnerable to changes in RTT and sending rate.

3.3 The TFRC Protocol

Applying the TCP response function (Equation (2.2)) as the control equation for congestion control requires the following:

- The parameters t_{RTT} and p have to be determined. The loss event rate p must be calculated at the receiver, while the round-trip time t_{RTT} could be measured at either the sender or the receiver.
- The receiver sends either the parameter p (in the case of sender-based RTT measurements) or the calculated value of the allowed sending rate R_{TCP} (in the case of receiver-based RTT measurements) back to the sender.
- The sender increases or decreases its transmission rate based on R_{TCP}.

For unicast the functionality could be split in a number of ways. In our proposal, the receiver only calculates p and feeds it back to the sender.

3.3.1 Sender Functionality

In order to use the control equation, the sender determines the values for the round-trip time t_{RTT} and retransmit timeout value t_{RTO}.

The sender and receiver together use timestamps for measuring the round-trip time. Every time the receiver sends feedback, it echoes the timestamp from the most recent data packet, along with the time that passed since that packet was received. This way, the sender can measure the round-trip time through the network. If the receiver does not send feedback immediately after the reception of a data packet, it has to adjust the echoed timestamp by the time interval that passed between receiving the last data packet and sending the feedback message.

The sender smoothes the measured round-trip time using an exponentially weighted moving average. This weight determines the responsiveness of the transmission rate to changes in round-trip time.

The sender could derive the retransmit timeout value t_{RTO} using the usual TCP formula:

$$t_{RTO} = t_{RTT} + 4 * RTT_{var}$$

where RTT_{var} is the RTT variance. However, in practice t_{RTO} only critically affects the allowed sending rate when the packet loss rate is very high. Different TCP implementations use drastically different clock granularities to calculate retransmit timeout values, so it is not clear that equation-based congestion control can accurately model a *typical* TCP implementation. Unlike TCP, TFRC does not use this value to determine whether it is necessary to retransmit, and so the consequences of inaccuracy are less serious. In practice the simple empirical heuristic of $t_{RTO} = 4t_{RTT}$ works reasonably well to provide fairness with TCP.

The sender obtains the value of p in feedback messages from the receiver at least once per round-trip time.

Every time a feedback message is received, the sender calculates a new value for the allowed sending rate R_{TCP} using the response function from Equation (2.2). If the actual sending rate R_{send} is less than R_{TCP}, the sender will increase its sending rate.

If R_{send} is greater than R_{TCP}, the sender must decrease the sending rate. While several design choices exist how to decrease the rate, the most simple form of directly setting R_{send} to R_{TCP} works well and is the behavior used in all the results presented in this chapter.

3.3.2 Receiver Functionality

The receiver provides feedback to allow the sender to measure the round-trip time (RTT). The receiver also calculates the loss event rate p and feeds it back to the sender. The calculation of the loss event rate is one of the most critical parts of TFRC. There is a clear trade-off between measuring the loss event rate over a short period of time and being able to respond rapidly to changes in the available bandwidth, versus measuring over a longer period of time and getting a signal that is less noisy.

The method of calculating the loss event rate has been the subject of much discussion and testing, and over that process several guidelines have emerged:

- The estimated loss event rate should track the actual loss event rate relatively smoothly in an environment with a stable steady-state loss event rate.
- The estimated loss rate should measure the *loss event rate* rather than the packet loss rate, where a *loss event* can consist of several packets lost within a round-trip time. This is discussed in more detail in Section 3.3.4.
- The estimated loss event rate should respond strongly to loss events in several successive round-trip times.
- The estimated loss event rate should increase only in response to a new loss event. (We note that this property is not satisfied by some of the methods described below.)
- Let a *loss interval* be defined as the number of packets between loss events. The estimated loss event rate should decrease only in response to a new loss interval that is longer than the previously-calculated average, or a sufficiently-long interval since the last loss event.

Obvious methods we looked at include the Exponentially Weighted Moving Average (EWMA) Loss Interval method, the Dynamic History Window method, and the Average Loss Interval method which is the method we chose.

The EWMA Loss Interval method uses an exponentially weighted moving average of the number of packets between loss events. Whenever a loss event occurs, the average loss interval is computed as

$$\hat{s} \leftarrow \alpha\hat{s} + (1-\alpha)s$$

where s is the number of packets received since the last loss event. As can be seen from Figure 3.1, depending on the weighting factor α the calculated loss event rate is either composed mainly of the most recent loss interval or the weights drop of very gradually and old lossintervals have an impact on the loss event rate for a long time. Both alternatives are undesirable for the loss measurement process. The former is very susceptible to noise while the latter is slow to react to changes in the network conditions.

The Dynamic History Window method uses a history window of packets whose length is determined by the current transmission rate. This method suffers from the effect that even with a

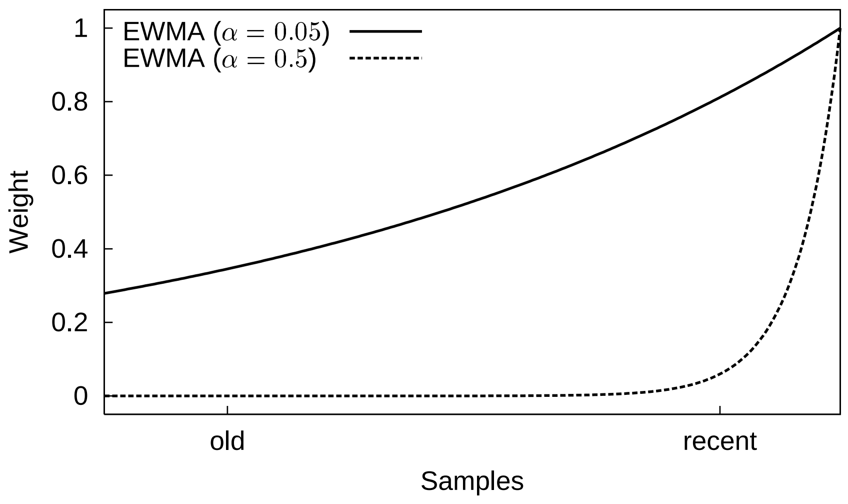


Figure 3.1: Weighting the loss intervals with an EWMA

perfectly periodic loss pattern, loss events entering and leaving the window cause changes to the measured loss rate, and hence add unnecessary noise to the loss signal.

The Average Loss Interval method computes the average loss rate over the last n loss intervals. By itself, the naive Average Loss Interval method suffers from two problems: the interval since the most recent loss is not necessarily a reflection of the underlying loss event rate, and there can be sudden changes in the calculated rate due to unrepresentative loss intervals leaving the n intervals we are looking at. These concerns are addressed below.

The full Average Loss Interval method differs from the naive version in several ways. Again, let s_i be the number of packets in the i-th most recent loss interval, and let the most recent interval s_0 be defined as the interval containing the packets that have arrived *since the last loss*. The first difference addresses the most recent loss interval s_0. When a loss occurs, the loss interval that has been s_0 now becomes s_1, all of the following loss intervals are correspondingly shifted down one, and the new loss interval s_0 is empty. As s_0 is not terminated by a loss, it is different from the other loss intervals. It is important to ignore s_0 in calculating the average loss interval unless s_0 is large enough that including it would increase the average. This allows the calculated loss interval to track smoothly in an environment with a stable loss event rate.

The second difference from the naive method reduces the sudden changes in the calculated loss rate that could result from unrepresentative loss intervals leaving the set of loss intervals used to calculate the loss rate. The full Average Loss Interval method uses an FIR (Finite Impulse

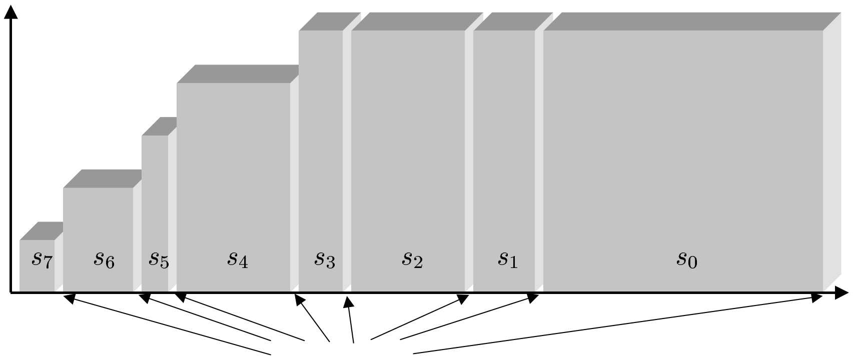


Figure 3.2: Weighted intervals between loss used to calculate loss probability

Response) filter to compute the average loss interval. The average loss interval $\hat{s}_{(1,n)}$ is calculated as a weighted average of the last n intervals as follows:

$$\hat{s}_{(1,n)} = \frac{\sum_{i=1}^{n} w_i s_i}{\sum_{i=1}^{n} w_i},$$

for weights w_i:

$$w_i = \begin{cases} 1 & \text{for } 1 \leq i \leq n/2, \\ 1 - \frac{i-n/2}{n/2+1} & \text{for } n/2 < i \leq n. \end{cases} \tag{3.1}$$

For $n = 8$, this gives weights of 1, 1, 1, 1, 0.8, 0.6, 0.4, and 0.2 for w_1 through w_8, respectively.

To determine whether to include s_0, the interval since the most recent loss, the full Average Loss Interval method also calculates $\hat{s}_{(0,n-1)}$:

$$\hat{s}_{(0,n-1)} = \frac{\sum_{i=0}^{n-1} w_{i+1} s_i}{\sum_{i=1}^{n} w_i}.$$

The final average loss interval $\hat{s}$ is $\max(\hat{s}_{(1,n)}, \hat{s}_{(0,n-1)})$, and the reported loss event rate is $1/\hat{s}$.

The sensitivity to noise of the calculated loss rate depends on the value of n. Figure 3.3 shows how well the current estimate calculated over n intervals matches the future loss event rate (as measured in the Internet experiments described in a later section). The left graph shows the average error and its standard deviation for equal weights ($w_1 = ... = w_n$) while the right graph is for decreasing weights as described above. A value of $n = 8$, with the most recent four samples equally weighted and decreasing weights for the older samples, appears to be a lower bound that still achieves a reasonable balance between resilience to noise and responding quickly to real changes in network conditions.

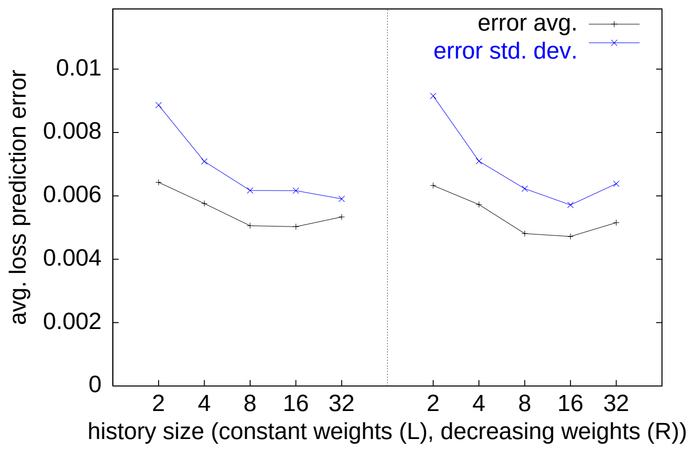


Figure 3.3: Prediction quality for different n

Because the Average Loss Interval method averages over a number of loss intervals, rather than over a number of packet arrivals, the naive Average Loss Interval method responds reasonably rapidly to a sudden increase in congestion, but is slow to respond to a sudden decrease in the loss rate. For this reason we deploy history discounting as a component of the full Average Loss Interval method, to allow a more timely response to a sustained decrease in congestion. History discounting is used by the TFRC receiver after the identification of a particularly long interval since the last dropped packet, to smoothly discount the weight given to older loss intervals.

The details of the discounting mechanism are as follows: If $s_0 > 2\hat{s}_{(i\geq1)}$, then the most recent loss interval s_0 is considerably longer than the recent average, and the weights for the older loss intervals are discounted correspondingly. The weights for the older loss intervals are discounted by using the following discount factor:

$$d_i = \max\left(0.5, \frac{2\hat{s}_{(i\geq1)}}{s_0}\right), \quad \text{for} \quad i > 0,$$

$$d_0 = 1.$$

The lower bound of 0.5 on the discount factor ensures that past losses will never be completely forgotten, regardless of the number of packet arrivals since the last loss.

When history discounting is invoked, this gives the following estimated loss interval:

$$\hat{s} = \frac{\sum_{i=0}^{n-1} d_i w_{i+1} s_i}{\sum_{i=1}^{n} d_{i-1} w_i}.$$

When loss occurs and the old interval s_0 is shifted to s_1, then the discount factors are also shifted, so that once an interval is discounted, it is never un-discounted, and its discount factor is never increased. In normal operation, in the absence of history discounting, d_i = 1 for all values of i. History discounting (also called proportional deweighing) is described in more detail in [Wid00] in Sections 3.7 and 4.8.1.

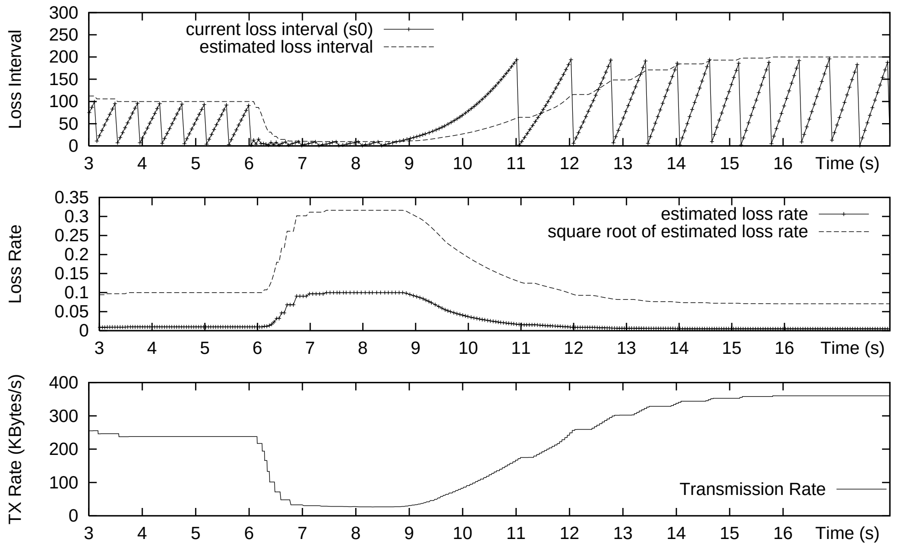


Figure 3.4: Illustration of the Average Loss Interval method with idealized periodic loss

Figure 3.4 shows a simulation using the full Average Loss Interval method for calculating the loss event rate at the receiver. The link loss rate is 1% before time 6, then 10% until time 9, and finally 0.5% until the end of the run. This simulation is rather unrealistic because the loss is periodic, but this illustrates the mechanism clearly.

In the top graph, the solid line shows the number of packets in the most recent loss interval, as calculated by the receiver once per round-trip time before sending a status report. The smoother dashed line shows the receiver's estimate of the average loss interval. The middle graph shows the receiver's estimated loss event rate p, which is simply the inverse of the average loss interval, along with $\sqrt{p}$. The bottom graph shows the sender's transmission rate which is calculated from p.

Several things are noticeable from these graphs:

- Before t=6, the loss rate is constant, and the Average Loss Interval method gives a completely stable measure of the loss rate.
- When the loss rate increases the transmission rate is rapidly reduced.
- When the loss rate decreases the transmission rate increases in a smooth manner, with no step increases even when older (10 packet) loss intervals are excluded from the history. With naive loss interval averaging we would have seen undesirable step-increases in the estimated loss interval, and hence in the transmission rate.

3.3.3 Slow-Start

The initial rate-based slow-start procedure should be similar to the window-based slow-start procedure followed by TCP where the sender roughly doubles its sending rate each round-trip time (exponential increase). However, TCP's ACK-clock mechanism provides a limit on the overshoot during slow-start. No more that two outgoing packets can be generated for each acknowledged data packet, so TCP cannot send at more than twice the bottleneck link bandwidth.

A rate-based protocol does not have this natural self-limiting property, and so a slow-start algorithm that doubles its sending rate every measured RTT can overshoot the bottleneck link bandwidth by significantly more than a factor of two. A simple mechanism to limit this overshoot is to have the receiver feed back the rate at which packets arrived at the receiver during the last measured RTT. If loss occurs, slow-start is terminated, but if loss doesn't occur the sender sets its rate to:

$$R_{send,i+1} = min\big(2R_{send,i}, 2R_{recv,i}\big)$$

This limits the slow-start overshoot to be no worse than that of TCP.

When the loss occurs that causes slow-start to terminate, there is no appropriate loss history from which to calculate the loss fraction for subsequent RTTs. The interval until the first loss is not very meaningful as the rate changes very rapidly during this time. The solution is to assume that the appropriate initial data rate is half of the rate when the loss occurred; the factor of one-half results from the delay inherent in the feedback loop. We then calculate the expected loss interval that would be required to produce this data rate and use this synthetic loss interval to seed the history mechanism. Real loss-interval data then replaces this synthetic value when it becomes available.

3.3.4 Discussion of Protocol Characteristics

Loss Fraction vs. Loss Event Fraction

The obvious way to measure a loss rate is as a loss fraction calculated by dividing the number of packets that were lost by the number of packets transmitted. However this does not accurately model the way TCP responds to loss. Different variants of TCP cope differently when multiple packets are lost from a window; Tahoe, NewReno, and Sack TCP implementations generally halve the congestion window once in response to several losses in a window, while Reno TCP typically reduces the congestion window twice in response to multiple losses in a window of data (i.e., the new rate will be $1/4$ of the old rate).

Because we are trying to emulate the best behavior of a conformant TCP implementation, we measure loss as a *loss event fraction*. Thus we explicitly ignore additional losses within a round-trip time that follow an initial loss, and model a transport protocol that reduces its window at most once for congestion notifications in one window of data. This closely models the mechanism used by most TCP variants.

The authors of [FHPW00b] explore the difference between the loss-event fraction and the regular loss fraction in the presence of random packet loss. It is shown that for a stable steady-state packet loss rate and a flow sending within a factor of two of the rate allowed by the TCP response function, the difference between the loss-event fraction and the loss fraction is at most 10%.

When routers use Random Early Detection (RED) [FJ93] as queue management mechanism, multiple packet drops in a window of data are less common, but with drop-tail queue management it is common for several packets in the same round-trip-time to be lost when the queue overflows. These multiple drops can result in multiple packets dropped from a window of data from a single flow, resulting in a significant difference between the loss fraction and the loss event fraction for that flow. A transient period of severe congestion can also result in multiple packets dropped from a window of data for a number of round-trip times, again resulting in a significant difference between the loss fraction and the loss event fraction during that transient period.

Increasing the Transmission Rate

One issue to resolve is how to increase the sending rate when the rate given by the control equation is greater than the current sending rate. As the loss event rate is not independent of the transmission rate, to avoid oscillatory behavior it might be necessary to provide damping, perhaps in the form of restricting the increase to be small relative to the sending rate during the period that it takes for the effect of the change to show up in feedback that reaches the sender.

In practice, the calculation of the loss event rate by the average loss interval method above provides sufficient damping, and there is little need to explicitly bound the increase. As shown in Appendix A.1 of [FHPW00a], given a fixed RTT and no history discounting, the increase in transmission rate is limited to about 0.14 packets per RTT every RTT (using Equation (2.2)). With history discounting, the TFRC mechanism increases its sending rate by at most 0.28 packets/RTT.

As changes in measured RTT are already damped using an EWMA, even with the maximum history discounting ($w = 1$), this increase rate does not exceed one packet per RTT every RTT, which is the rate of increase of a TCP flow in congestion avoidance mode.

Response to Persistent Congestion

Simulations in Appendix A.2 of [FHPW00a] show that, in contrast to TCP, TFRC requires from three to eight round-trip times to reduce its sending rate in half in response to persistent congestion. This slower response to congestion is coupled with a slower increase in the sending rate than that of TCP. In contrast to TCP's increase of the sending rate by one packet/RTT for every round-trip time without congestion, TFRC generally does not increase its sending rate at all until a longer-than-average period has passed without congestion. Thus the milder decrease of TFRC in response to congestion is coupled with a considerably milder increase in the absence of congestion.

3.4 Experimental Evaluation

To demonstrate that it is feasible to widely deploy TFRC we need to demonstrate that it co-exists well when sharing congested bottlenecks with TCP traffic of different flavors. We also need to demonstrate that it behaves well in isolation (i.e., with no parallel TCP flows), and that it performs acceptably over a wide range of network conditions.

We have tested TFRC extensively across the public Internet and with the *ns*-2 network simulator. The results give us confidence that TFRC is remarkably fair when competing with TCP traffic, that situations where it performs very badly are rare, and that it behaves well across a very wide range of network conditions. For the sake of brevity, we can only give a very brief overview of some TFRC of the experiments and refer the interested reader to [FHPW00a, FHPW00b, Pad00, Wid00] for more detailed results.

3.4.1 Simulation Results

Figure 3.5 illustrates the fairness of TFRC when competing with TCP traffic in both DropTail and RED [FJ93] queues. In these *ns*-2 simulations n TCP and n TFRC flows share a common bottleneck; we vary the number of flows and the bottleneck bandwidth, and scale the queue size with the bandwidth. The graph shows the mean TCP throughput over the last 60 seconds of simulation, normalized so that a value of one would be a fair share of the link bandwidth. The network utilization is always greater than 90% and often greater than 99%, so almost all of the remaining bandwidth is used by the TFRC flows. These figures illustrate that TFRC and TCP co-exist fairly across a wide range of network conditions, and that TCP throughput is similar to what it would be if the competing traffic was TCP instead of TFRC.

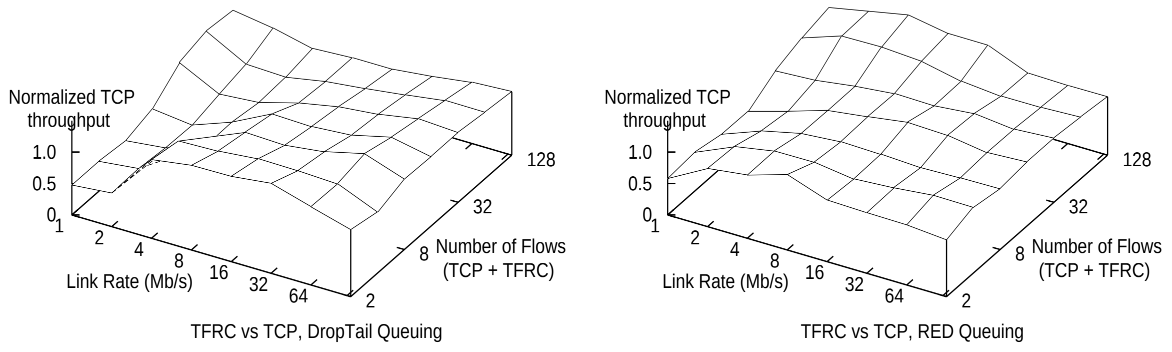


Figure 3.5: TCP sending rate while co-existing with TFRC

The graphs do show that there are some cases (typically where the mean TCP window is very small) where TCP suffers. This appears to be because TCP is more bursty than TFRC. When we modify TFRC to send two packets every two inter-packet intervals, TCP competes more fairly in these cases. However this is not something we would recommend for normal operation.

Although the mean throughput of the two protocols is rather similar, the variance can be quite high. This is illustrated in Figure 3.6 which shows the data points from Figure 3.5 for the simulations with a 15 MBit/s bottleneck link. Each column represents the results of a single simulation, and each data point is the normalized mean throughput of a single flow. Typically, the TCP flows have higher variance than the TFRC flows. In general, the variance between flows increases as the bandwidth per flow decreases. This is to be expected as Equation 2.2 indicates that TCP (and hence also TFRC) becomes more sensitive to loss as the loss rate increases, which it must do at lower bandwidths.

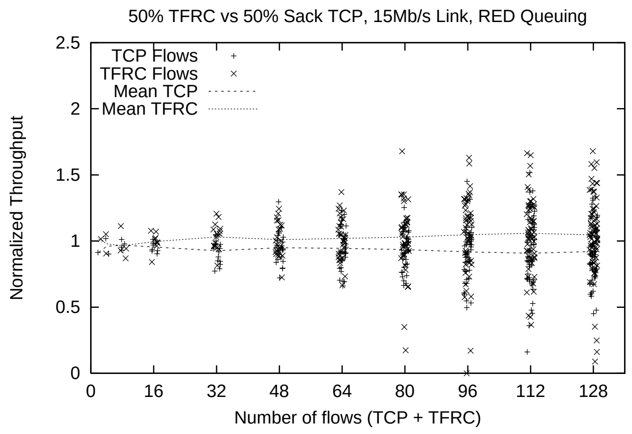


Figure 3.6: TCP competing with TRFC (15 MBit/s bottleneck with RED queue)

3.4.2 Implementation Results

We have implemented the TFRC algorithm and conducted many experiments to explore the performance of TFRC in the Internet. TFRC was implemented in C under the FreeBSD operating system (FreeBSD 3.3) and further tested under Solaris and Linux. Our tests include two different transcontinental links, and sites connected by a microwave link, T1 link, OC3 link, cable modem, and dial-up modem. In addition, conditions unavailable to us over the Internet were tested against real TCP implementations in Dummynet [Riz98]. Again, we only present a very small selection of simulation results and refer to the aforementioned literature for a detailed discussion of all the experiments.

Figure 3.7 shows a typical experiment with three TCP flows and one TFRC flow running concurrently from London to Berkeley, with the bandwidth measured over one-second intervals. In this case, the transmission rate of the TFRC flow is slightly lower, on average, than that of the TCP flows. At the same time, the transmission rate of the TFRC flow is smooth, with a low variance; in contrast, the bandwidth used by each TCP flow varies strongly even over relatively short time periods.

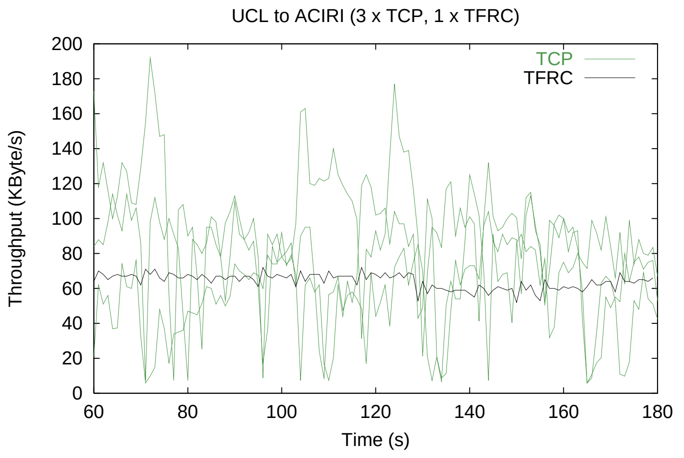


Figure 3.7: Three long-distance TCP flows and one TFRC flow over the Internet

The analysis of TFRC's and TCP's coefficient of variation (CoV)[2] further indicates that TFRC throughput is much more stable than TCP throughput under comparable network conditions. Over a variety of timescales, TFRC's CoV is consistently less than that of TCP on different Internet paths as shown in Figure 3.8.

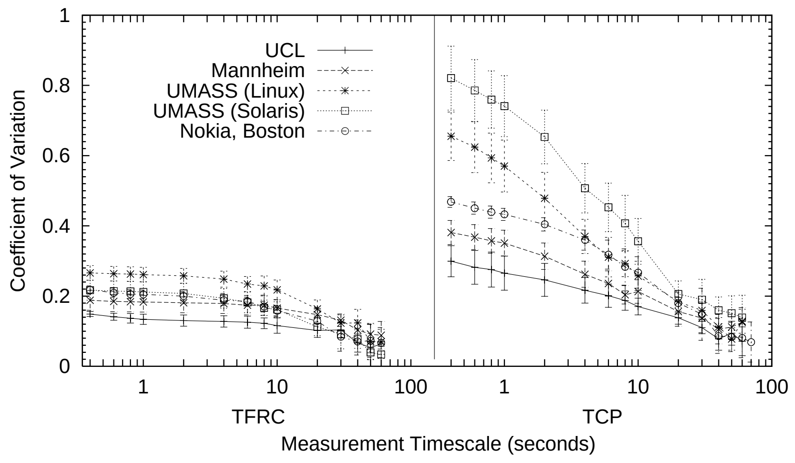


Figure 3.8: CoV of TFRC (left) and TCP (right) over different Internet paths

To summarize the results, TFRC is generally fair to TCP traffic across the wide range of network types and conditions we examined. The TFRC simulations and analysis contained in this chapter and the other publications mentioned above lead us to conclude that TFRC is a suitable protocol

[2]The coefficient of variation is defined as the standard deviation over the mean. It is dimensionless and independent of scale.

for unicast congestion control for rate-adaptive applications. In the following chapters, we will extend this basic approach to provide a higher degree of flexibility and to meet the requirements of applications for which basic unicast congestion control is inappropriate.

Chapter 4

Probabilistic Congestion Control

4.1 Introduction

One prime aim of many congestion control schemes is to share the available bandwidth in a fair manner with TCP-based applications, thus falling into the category of TCP-friendly congestion control mechanisms. TCP, as well as existing TCP-friendly congestion control algorithms, require that the data rate of an individual flow can be adapted to network conditions. Using TCP, it may take a variable amount of time to transmit a fixed amount of data, or with TCP-friendly congestion control, the quality of an audio or video stream may be adapted to the available bandwidth.

While this is not a problem for a large number of applications, there are cases where the data rate of an individual flow is determined by the application and cannot be adjusted to the network conditions. Live audio and video transmissions with a fixed minimum quality, below which reception is useless, are a typical example. Rate adaptation is only possible up to this lower limit. Networked computer games fall into the same category, considering the fact that players are very reluctant to accept the delayed transmission of information about a remote player's actions. For this class of applications there are only two acceptable states: either a flow is *on* and the sender transmits data the at rate determined by the application, or it is *off* and no data is transmitted at all. We call network flows produced by these applications non-adaptable flows.

In this chapter we describe a TCP-friendly end-to-end congestion control mechanism for non-adaptable unicast flows called Probabilistic Congestion Control (PCC) [WMD02]. Unlike conventional congestion control mechanisms, it does not require an adaptation of the sending rate. The main idea of PCC is

- to calculate a probability for the two possible states (on/off) so that the expected average rate of the flow is TCP-friendly,
- to perform a corresponding random experiment to determine the new state of the non-adaptable flow, and
- to repeat the previous steps continuously to account for changes in network conditions.

Through this mechanism it is ensured that the aggregate of multiple PCC flows behaves in a TCP-friendly manner.

4.2 Related Work

A discussion of TCP-friendly congestion control mechanisms can be found for example in [WDM01]. TCP, as well as all existing TCP-friendly congestion control schemes, require that the bandwidth consumed by a flow be adapted to the level of congestion in the network. By definition, non-adaptable flows cannot use these congestion control mechanisms.

It is conceivable to use reservation mechanisms such as Intserv/RSVP [BZB$^+$97] or Diffserv [BBC$^+$98] for non-adaptable flows so as to prevent congestion altogether. However, these mechanisms require that the network supports the reservation of resources or provides different service classes. This is currently not the case for the Internet.

Recently, there have been proposals for distributed admission control schemes [GK99, KKZ00, Kel01]. Such schemes do not require extensive state in the network routers but the admission control decisions are performed locally at the end hosts (end-to-end admission control). Still, distributed admission control schemes either admit a flow for its whole duration or not at all. Thus, their granularity is too coarse for quickly changing levels of congestion.

In contrast, PCC is an end-to-end mechanism that allows to “partly” admit a flow and to continuously adjust the number of flows to network conditions. Flows that were not admitted at first may later resume data transmission, and flows that were admitted may be suspended at a later time, providing a more flexible congestion control framework for applications that can cope with flows being temporarily suspended.

4.3 Non-Adaptable Flows

A non-adaptable flow is defined as a data flow with a sending rate that is determined by the application and which cannot be adjusted to the level of congestion in the network. It has exactly two possible states: either it is in the state *on*, carrying data at the rate determined by the application, or it is *off*, meaning that no data is transmitted at all.

Examples of applications using non-adaptable flows are audio or video transmissions with a fixed bit rate or a fixed quality.[1] There are two main reasons why it may not be possible to scale down a media flow: either the user does not accept a lower quality, or the quality is already at the lowest possible level. The second reason indicates that a congestion control mechanism for non-adaptable flows can complement congestion control schemes that adapt the rate of a flow to current network conditions.

Other examples of applications with non-adaptable flows are commercial network games such as Diablo II, Quake III, Ultima Online, and Everquest. These games typically employ a client-server architecture. The data rate of the flows between client and server is determined by the fact that the actions of the players must be transmitted instantaneously. Although the sending rate required by such flows is typically low, thousands of users may participate in the same game, placing a significant burden on the network. Similar restrictions hold for the flows between participants of distributed virtual environments without a centralized server. If a congestion control scheme delays the transmission of actions too long, the application quickly becomes unusable. This can easily be experienced by experimenting with a state-of-the-art TCP-based networked computer game during peak hours. For this reason, a number of applications resort to UDP and circumvent congestion control.

A situation with either no congestion control at all or vastly reduced utility in the face of moderate congestion is not desirable. A preferable approach is to turn the flows of some participants off and to inform the applications accordingly. All other participants do not need to react to the congestion. On average, all users should be able to participate in the session for a reasonable amount of time between off-periods to ensure utility of the application. At the same time, off-periods should be distributed fairly among all participants.

[1]While a fixed quality may result in a variable bit rate, this rate is still exclusively determined by the encoder and reducing it to adapt to congestion would alter the quality.

4.4 The PCC Protocol

The Probabilistic Congestion Control scheme (PCC) provides congestion control for non-adaptable unicast flows by suspending flows at appropriate times. PCC is an end-to-end mechanism and does not require the support of routers or other intermediate systems in the network.

4.4.1 PCC Requirements

The key aspect of PCC is that – as long as there is a sufficiently high level of statistical multiplexing – it is not important that each single non-adaptable flow behave in a TCP-friendly manner at any specific point of time. What is important is that the aggregation of all non-adaptable flows on a given link behave as if the flows were TCP-friendly. Due to the law of large numbers this can be achieved if each PCC flow has an expected average rate which is TCP-friendly and if each link is traversed by a sufficiently large number of independent PCC flows.

At first glance, the latter requirement may be considered problematic because it is possible that a link is traversed only by a small number of PCC flows. However, further reflection reveals that in this case the PCC flows will only be significant in terms of network congestion if each individual PCC flow occupies a high percentage of the link's bandwidth. We therefore relax the condition as follows: a single PCC flow is expected to have a rate that is only a small fraction of the available bandwidth on any link that it crosses. Given the current development of available bandwidth in computer networks, this is a condition that is likely to hold true.

Altogether, the following requirements have to be fulfilled for PCC to be applicable:

R1: *Network conditions are relatively independent of the actions of a single PCC flow.* The network has a high level of statistical multiplexing, or single PCC flows occupy only a small fraction of the available bandwidth.

R2: *No synchronization of PCC flows at startup.* PCC flows start up independent of each other.

R3: *The average rate of a PCC flow can be predicted.* In order for PCC to work, it must be possible to predict the average rate of a PCC flow.[2]

[2]There are multiple ways in which this can be done, ranging from a constant bit-rate flow where this prediction is trivial, to the usage of application level knowledge or prediction based on past samples of the data rate.

R4: *The average rate of a TCP flow under the same conditions can be estimated.* We expect that there is a reasonably accurate method to estimate the average bandwidth that a TCP flow would have under the same network conditions.

4.4.2 Architecture

Figure 4.1 gives an overview of the PCC architecture. A PCC receiver monitors the network conditions and estimates a TCP-friendly rate using a model of long-term TCP throughput. Whenever a PCC receiver observes a degradation in network conditions, it conducts a random experiment to determine whether or not the flow should be suspended. In case the experiment fails, a control packet is sent to notify the sender that it is temporarily required to stop. After a certain off-period, the sender may resume data transmission. For PCC, we chose to allocate as much functionality as possible to the receiver in order to facilitate a future extension of PCC to multicast.

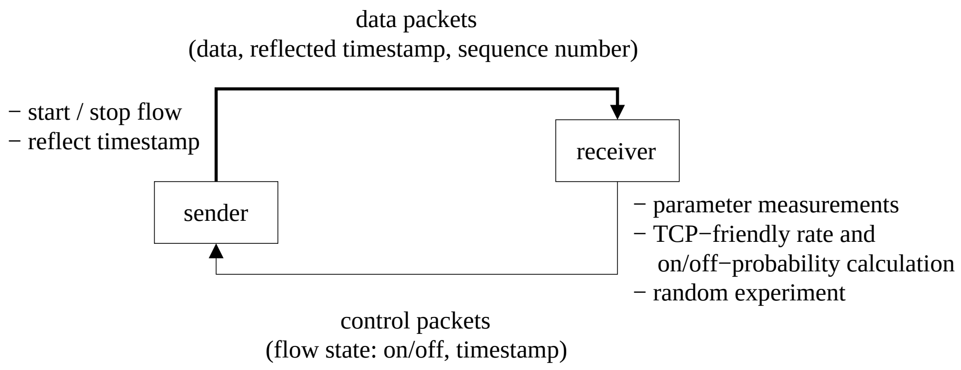


Figure 4.1: PCC architecture

While a flow is in the on-state, control packets are sent by the receivers at certain time intervals. They are needed to continuously measure the round-trip time required to determine the TCP-friendly rate, and they serve as a backup mechanism in case of very heavy network congestion. In the absence of these periodic control messages, the sender stops sending, thus safeguarding against the loss of notifications to stop. As long as the flow is in the on-state, the data packets are transmitted at the rate determined by the application. The sender includes in each data packet the timestamp of the most recent control packet it received as well as the time interval between receiving the control packet and sending the data packet. With this information the receiver is able to determine the round-trip time. Data packets also contain a sequence number to allow the receiver to detect packet losses.

For PCC, we will use the simplified version of the TCP throughput formula given in Equation (2.3) to estimate a TCP-friendly rate. The loss event rate for the model is determined in the same way as in the TFRC protocol (see Section 3.3.2). However, PCC is independent of the specific mechanism used to estimate the throughput of a TCP flow for given network conditions, as long as it determines a sufficiently accurate TCP-friendly rate. A possible alternative, for example, would be to use the rate calculation mechanism of TCP Emulation At Receivers (TEAR) [ROY00].

4.4.3 Basic PCC Mechanism

To determine the probability with which a PCC flow is allowed to send for a certain time interval T, it is necessary to compare the average rate R_{NA} of PCC to the TCP-friendly rate R_{TCP}:

$$p \cdot T \cdot R_{NA} = T \cdot R_{TCP} \quad \Longrightarrow \quad p = \frac{R_{TCP}}{R_{NA}} \tag{4.1}$$

where p denotes the ratio of R_{NA} to R_{TCP}. When solving the equation, two outcomes are possible:

- $p \geq 1$: The non-adaptable flow consumes less than or the same amount of bandwidth that would be carried by TCP and should therefore stay on.
- $0 \leq p < 1$: The non-adaptable flow consumes more bandwidth than a comparable TCP-friendly flow. In this case, p is taken as a probability, and the non-adaptable flow should be turned off with probability $1 - p$.

For $p \in [0, 1]$, a uniformly distributed random number x is drawn from the interval $(0, 1]$. If $x > p$ holds, the PCC flow is turned off for a time of T. After that time interval the flow may be turned on again. If $x \leq p$, the flow remains in the on-state. Since we require a sufficient level of statistical multiplexing (*R1*) and because of the law of large numbers, the aggregation of all PCC flows over a given link behaves in a TCP-friendly manner.

T is an application-specific parameter that is crucial for the utility of the protocol and thus for the user acceptance of the congestion control mechanism. For example, if short news clips are transmitted, T should be equal to the length of these clips. If a networked computer game is played, T should be determined so that in “normal” congestion situations the player is able to perform some meaningful tasks during the average time the flow stays on. If the network is designed to carry the required traffic (i.e., congestion is low), then the average on-time will be a large multiple of T. T should be adjusted by some random offset to prevent synchronization of PCC flows in case several flows with the same value for T were forced to cease sending simultaneously due to heavy congestion.

When setting T to very small values, PCC behaves more and more like a conventional rate-based congestion control algorithm. In the extreme case, the application rate is the line speed of the outgoing interface of the PCC sender, the on-time is the time required to send a single packet and the off-time corresponds to the inter-packet interval, resulting in a protocol that behaves very much like TFRC. At the other end of the spectrum, for very large values of T, PCC behaves like an admission control scheme based on snapshots of the current network conditions as determined with Equation (2.3).

4.4.4 Continuous Evaluation

Under the assumption of a relatively constant level of congestion, the further behavior of PCC is very simple. After a time of T, a flow that is in the on-state will repeat the random experiment using the same R_{TCP}. However, in a real network the level of congestion is not constant but may change significantly within a time much shorter than T. There are two cases to consider: network conditions may improve (increasing R_{TCP}) or the congestion may get worse.

The first case is not problematic since it does not endanger the network itself. PCC flows may be treated unfairly in that they are turned off with a higher probability than they should be. However, after a time of T the decision will be reevaluated with the correct probability, and PCC will adjust to the new level of congestion.

The second case is much more dangerous to the network. In order to prevent unfair treatment of competing adaptive flows or even a congestion collapse, it is very important that PCC flows respond quickly to an increase in congestion. Therefore, during T PCC periodically updates the value for p and performs further random experiments if necessary.

Obviously, it is not acceptable to simply recalculate p without accounting for the fact that the flow could have been turned off during one of the previous experiments. Without any adjustments, PCC would continue to perform the same random experiment again and again and the probability to survive those experiments would drop to 0. The general idea of how to avoid this drop-to-zero behavior is to adjust the rate used in the equations to represent the current expected average data rate of the flow.

PCC modifies the value R_{NA}, taking into account the last random experiments that have been performed for the flow. To this end, PCC maintains a set P of the probabilities p_i with which the flow stayed on in the random experiments during the last T seconds.[3] The so-called effective rate R_{EFF} is determined according to the following equation:

$$R_{EFF} = \begin{cases} R_{NA} \prod_{p_i \in P} p_i & \text{for } P \neq \emptyset \\ R_{NA} & \text{for } P = \emptyset \end{cases} \tag{4.2}$$

For the continuous evaluation and the random experiments R_{EFF} replaces R_{NA} in Equation (4.1). After a time span of T, the corresponding p_i is removed from the set. Thus, the overall probability for a PCC flow to stay on for a given interval T corresponds to the worst network conditions PCC experienced during that interval. Continuous evaluation is further studied by means of an example scenario in Section 4.5.

4.4.5 Initialization

At the initial startup and when a suspended flow restarts, a receiver does not have a valid estimate of the current condition of the network and thus is not able to instantaneously compute a meaningful TCP-friendly rate. To avoid unstable behavior, a flow will stay in the on-state for at least a protected time T', where T' is the amount of time required to get the necessary number of measurements to obtain a sufficiently accurate estimate of the network conditions.

After T', PCC determines whether it should cease to send or may continue. In order to take the data transmitted during the protected time into account, the probability of turning the flow off is increased during the first interval of T so that the average amount of data transmitted during $T' + T$ is equal to that carried by a competing TCP flow. Let R'_{NA} denote the average rate of the non-adaptive flow during the protected time and R'_{TCP} the average rate a TCP flow would have achieved during the same time. For

$$T' \cdot R'_{NA} + p' \cdot T \cdot R_{NA} = T' \cdot R'_{TCP} + T \cdot R_{TCP}$$

[3]Note that $p_i = 1$ if the corresponding $p \geq 1$.

the adjusted ratio p' can be calculated as

$$\begin{aligned} \Longrightarrow \quad p' &= \frac{T \cdot R_{TCP} + T' \cdot (R'_{TCP} - R'_{NA})}{T \cdot R_{NA}} \\ &= p - \frac{T'(R'_{NA} - R'_{TCP})}{T \cdot R_{NA}} \end{aligned} \tag{4.3}$$

Again, for $0 \leq p' \leq 1$ we use p' as the probability for the random experiment. If the flow is turned off, the application may resume sending after it has been off for at least T seconds, starting again with the initialization step. If the flow is not turned off, then the flow will stay on for at least T more seconds, provided that the congestion situation of the network does not get worse.

Note that it is now possible that $p' \leq 0$ if the non-adaptable flow transmits more data during T' than a TCP flow would during $T' + T$. Obviously, in this case p' cannot be used as a probability for the random experiment. Instead, it is necessary to turn the flow off and to increase T, so that $p' = 0$.

Through the above mechanism the excess data transmitted during the protected time T' is distributed over a time span of T. At time T', $R'_{TCP} = R_{TCP}$ and $R'_{NA} = R_{NA}$ but in contrast to R'_{TCP} and R'_{NA}, R_{TCP} and R_{NA} continue to be updated after T'.

When a random experiment has to be conducted, it is necessary to calculate not only p' but also the corresponding p. Each is included in their respective set P' and P. As long as PCC is in the first T slot and the protected time has to be accounted for, the values in P' are used to calculate the effective rate and thus the on-probability. Later on, the set P is used.

It may be considered problematic to let a flow send at its full rate for T' as this violates the idea of exploring the available bandwidth as is done, e.g., by TCP slow-start. However, requirements *R1* (high level of statistical multiplexing) and *R2* (no synchronization at startup) prevent this from causing excessive congestion. In addition, the value of T' will usually decrease the more congested the network is since the measurement of the loss event rate makes up most of the time interval T'. Loss events become more frequent as congestion increases, and therefore the estimate of the network conditions converges faster to the real value. How PCC could gradually probe for available bandwidth *before* the flow is turned on is described in the next section.

While R_{TCP} is determined, the receiver also calculates the average rate of the non-adaptable flow R_{NA}.[4] Summing up, three important values are determined during initialization: R_{TCP}, R_{NA}, and T'.

4.4.6 PCC Options

While the current version of PCC works as described above, there are a number of options and possible improvements that we have investigated. In the following we outline possible modifications to PCC.

Loss Rate Monitoring

PCC flows do not take into account the impact of their own actions on the network conditions. Assume that the random experiments of a number of PCC flows fail due to increased congestion, but that the congestion was largely caused by these PCC flows. Then too many flows will be suspended since it is impossible to include the expected improvement in the network conditions in the calculation of the on-probability. Similarly, when the bandwidth consumed by PCC flows during the protected time is a significant fraction of the bottleneck link bandwidth, severe congestion may be inevitable. Even after the protected time, the changes in network conditions caused by PCC flows that consume a large fraction of the bandwidth are undesirable. In any case, a link in the inner network dominated by PCC flows should be rare since TCP flows make up between 90% and 95% of today's Internet traffic.

For these reasons it is vital that the condition of a sufficient level of statistical multiplexing holds and that the PCC flows do not consume too large a fraction of the bandwidth of the bottleneck link. By continuously monitoring the packet loss rate (e.g., through probe packets) and correlating it with the on- and off-times of a PCC flow, it is possible to estimate the impact of the flow on the network conditions. If the PCC flow causes very large variations in the loss rate when it is switched on or off, the flow should be suspended permanently. With this extension it is possible to use PCC in environments where it is unclear whether the condition of a sufficient level of statistical multiplexing is fulfilled.

[4]In our implementation, we use an exponentially weighted moving average of past PCC rates, but as noted in requirement *R3*, other options are possible.

Fixed On-Time

Through the continuous evaluation, a flow stays on for a variable amount of time (possibly for the whole session if network conditions permit) before being suspended. Applications that require a fixed on-time to transmit a certain amount of data but are flexible with regard to the time they are turned off, may reverse this behavior. It is possible to modify PCC so that flows stay on for a time of T and during that time determine how long they have to be suspended in order to be TCP-friendly. However, under such circumstances the value of T should be within reasonable bounds of the timescale over which changes in the network conditions occur. If network conditions deteriorate significantly after a number of PCC flows are turned on, it is not acceptable for a congestion control mechanism to not react at all for a time T if T is much larger than the time over which the change occurs. Furthermore, network conditions might already have improved when the flows are finally suspended in which case suspending the flows would be futile.

Probe While Off

PCC flows may on average receive less bandwidth than competing TCP flows since a flow that has been turned off may resume only after a time of T, even if network conditions improve earlier. This degrades PCC's performance, particularly if T is large. In order to improve average PCC throughput, flows that are off could monitor network congestion by sending probe packets at a very low rate from the sender to the receiver. The rate R_{OFF} produced by the probe packets needs to be taken into account in Equations (4.1) and (4.3) by including an additional factor $(1 - p) \cdot R_{OFF} \cdot T$.

If the loss rate and the round-trip time of the probe packets signal that R_{TCP} has improved, a flow that has been turned off may be turned on again immediately, without waiting for the remainder of T to pass, and without performing an initialization step. This may be done only if, under the new network conditions, all experiments within the last T interval would have been successful. If the congestion situation worsens later on, it must be checked whether any of the experiments during the last T interval would have failed. If this is the case, the flow must be turned off again. Only after the last entry in set P has timed out may the flow resume normal operation. For *Probe While Off* to work correctly, it is of major importance that estimating the network parameters is independent of the packet rate at which the measurements are performed.

Probe Before On

In PCC, a flow is turned on upon initialization. This has two drawbacks. First, it violates the idea of exploring the available bandwidth as in TCP slow-start. Second, the flow may be turned off immediately after the initialization is complete, so that the user perceives only a brief moment where the application seems to work, before it is turned off. An alternative would be to send probe packets at an increasing rate before deciding whether or not to turn on the flow. Only after the parameters have been estimated and the random experiment has succeeded will real data for the flow be transmitted.

The current version of PCC does not include *Probe Before On* or *Probe While Off*. The drawback of *Probe Before On* is that bandwidth is wasted by probe packets and that the initial startup of a flow is delayed by T'. *Probe While Off* improves PCC performance under quickly changing network conditions but leads to more frequent changes between the states "on" and "off", which is likely to be distracting to the user of the application. The mechanism can be improved by including a threshold, so that the flow is turned on again only if the available bandwidth increases significantly.

4.5 Example of PCC Operation

To provide a better understanding of the behavior of PCC, let us demonstrate how PCC operates by means of an example. As depicted in Figure 4.2, the sender starts transmitting at the rate determined by the application. After $T' = 10$ seconds the receiver arrives at an initial estimate of $R_{NA} = 100$ kBit/s and $R_{TCP} = 80$ kBit/s. Furthermore, let us assume that the application developer decided that $T = 50$ seconds is a good value for the given application. Now p can be calculated as:

$$p = \frac{80 \text{ kBit/s}}{100 \text{ kBit/s}} = 0.8$$

The value of p is included in the set P and p' is calculated since we are in the first T interval and have to make up for the data transmitted during the protected time.

$$p' = p - \frac{10s \cdot (100 \text{ kBit/s} - 80 \text{ kBit/s})}{50s \cdot 100 \text{ kBit/s}} = 0.8 - 0.04 = 0.76$$

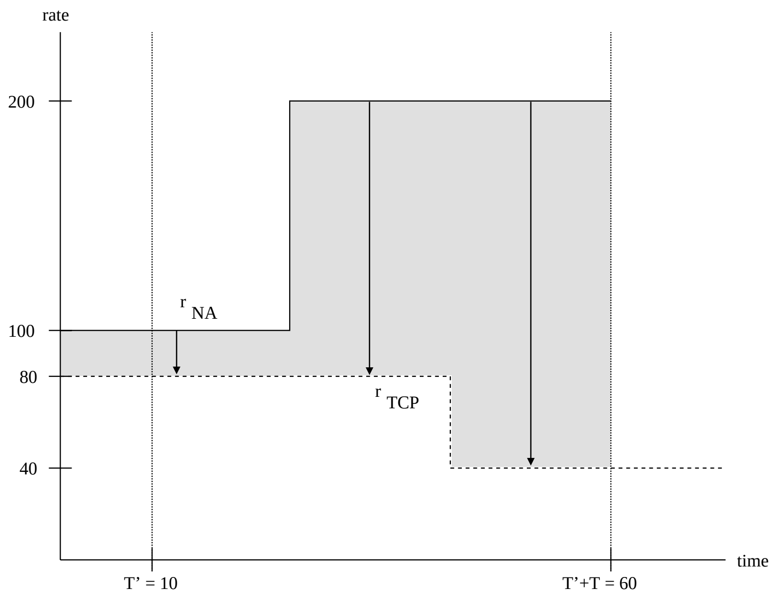


Figure 4.2: Example of PCC operation

Now a random number is drawn from the interval $(0, 1]$, deciding whether the flow will stay on or be turned off. Given a high level of statistical multiplexing, this will result in roughly 1 out of 4 PCC flows being turned off, with the aggregation of the remaining PCC flows using a fair, TCP-friendly share of the bandwidth.

Let us assume that the random number drawn is smaller than p' and that the flow will stay in the on-state. As depicted in Figure 4.2, at some later point in time, the bandwidth required by the application increases to $R_{NA} = 200$ kBit/s. A new value for p is then calculated as follows:

$$p = \frac{80 \text{ kBit/s}}{200 \text{ kBit/s} \cdot 0.8} = 0.5$$

This value for p is saved to the set P for later use. The adjusted probability p' has to be calculated based on the past value of p'.

$$\begin{aligned} p' &= \frac{80 \text{ kBit/s}}{200 \text{ kBit/s} \cdot 0.76} - \frac{10s \cdot (100 \text{ kBit/s} - 80 \text{ kBit/s})}{50s \cdot 200 \text{ kBit/s} \cdot 0.76} \\ &= 0.5 \end{aligned}$$

Let the random number drawn for this decision be below 0.5 so that the flow remains on. A few seconds after this decision the TCP-friendly rate drops to $R_{TCP} = 40$ kBit/s. Consequently new values for p and p' are calculated:

$$\begin{aligned}
p &= \frac{40 \text{ kBit/s}}{200 \text{ kBit/s} \cdot 0.8 \cdot 0.5} = 0.5 \\
p' &= \frac{40 \text{ kBit/s}}{200 \text{ kBit/s} \cdot 0.76 \cdot 0.5} - \frac{10s \cdot (100 \text{ kBit/s} - 80 \text{ kBit/s})}{50s \cdot 200 \text{ kBit/s} \cdot 0.76 \cdot 0.5} \\
&= 0.47
\end{aligned}$$

The value p is stored in P. Again, let the random number drawn be below p'.

At $T' + T = 60$ seconds two things happen: first, the data transmitted during the protected time need no longer be accounted for since PCC has made up for that during the past T interval. Therefore p' is no longer calculated. Second, the first value within P times out and is removed from the set. If the network situation has not changed this will result in the following new value for p:

$$p = \frac{40 \text{ kBit/s}}{200 \text{ kBit/s} \cdot 0.5 \cdot 0.5} = 0.8$$

This time let us assume that the random number is larger than p. As a consequence the flow is suspended for the next T interval before it may start again with a protected time.

It should be noted that this example was designed to demonstrate how PCC works, and the values used here may not be reasonable for real scenarios. Usually, the protected time is much shorter than 10 seconds, random experiments will be more frequent as network conditions change continuously, and an application rate of five times the TCP-friendly rate strongly indicates that the network is simply not provisioned for the application.

4.6 Simulations

In this section, we use network simulation to analyze PCC's behavior. Simulations are based on the dumbbell topology (Figure 4.3) since it is sufficient to analyze PCC fairness, and the results can be compared to those of other congestion control protocols evaluated with the same topology. For the same reason, simulations are carried out with the *ns*-2 network simulator [BEF$^+$00], commonly used to evaluate such protocols. For the routers, drop-tail queuing (with a buffer size of 50 packets) is used as queuing strategy. We use the standard *ns* implementation of TCP SACK for the flows competing with PCC.

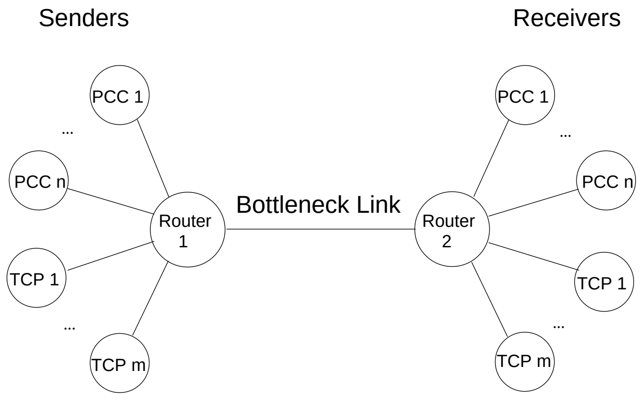


Figure 4.3: Simulation topology

4.6.1 TCP-Friendliness

A typical example of PCC behavior is shown in Figure 4.4. For this simulation, 32 PCC flows and 32 TCP flows are run over the same bottleneck link with 32 MBit/s capacity. At an application sending rate of 750 kBit/s, the PCC flows should ideally be in the on-state for two thirds of the time. In this example, T is set to 60 seconds, leading to an expected average on-time of 120 seconds. The graph depicts the throughput of one sample TCP flow and one sample PCC flow, as well as the average throughput of all 32 PCC flows. The starting time of the PCC flows is spread out over the first 50 seconds to avoid synchronization.

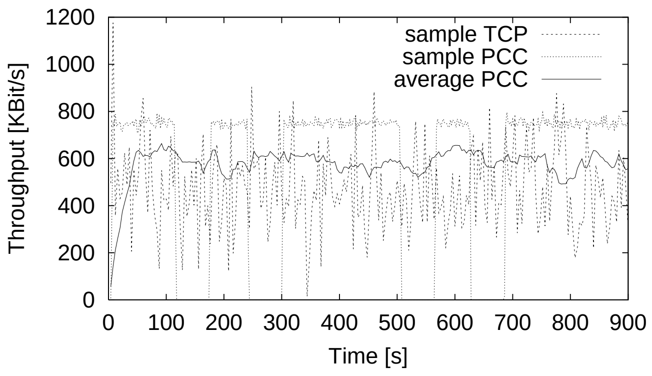


Figure 4.4: PCC and TCP throughput

The TCP rate shows the usual oscillations around the fair rate of 500 kBit/s. PCC's behavior is nearly perfect, with an average rate that closely matches the fair rate and an on-off ratio of two to one. Naturally, not all of the 32 PCC flows achieve exactly this ratio; some stay on for more, some for less time.

While 64 flows in total may not be considered a high level of statistical multiplexing, it suffices to render the overall network conditions relatively independent of the behavior of a single flow. Simulations with a higher number of flows resulted in very similar behavior.

4.6.2 Intra-Protocol Fairness

Usually, it is desirable to evenly distribute the necessary off-times over all PCC flows instead of severely penalizing only a few. To examine PCC's intra-protocol fairness, a simulation setup similar to the previous one is used, yet the number of concurrent PCC and TCP flows varied between 2 and 128. The probability density function of the throughput distribution from these simulations is shown in Figure 4.5. As expected, the throughput range is larger for PCC. The coefficient of variation (standard deviation over mean) for PCC throughput is 15% compared to a TCP coefficient of variation of only about 3%, resulting from the timescale for changes in the states of the PCC flows being 60 seconds instead of a few RTTs for TCP flows.

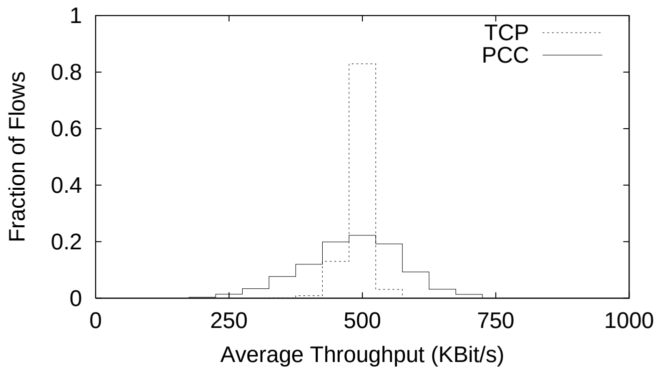


Figure 4.5: Distribution of flow throughput

There is a direct tradeoff between the parameter T and the intra-protocol fairness. Long on-times, achieved by a large T, are desirable for many applications but they are offset by a decrease in intra-protocol fairness as other PCC flows may be suspended for a long period of time. Taken to the extreme, for very large T, flows may stay on for the whole duration of the session, or they are not turned on at all, leading to a form of distributed admission control scheme.

4.6.3 Responsiveness

In addition to inter- and intra-protocol fairness, sufficient responsiveness of a flow to changes in the network conditions is important to ensure acceptable protocol behavior. TCP adapts almost immediately to an increase in congestion (manifest in the form of packet loss). Through the continuous evaluation at timescales of less than T, as described in Section 4.4.4, PCC can react nearly as fast as TCP to increased congestion, however, it will react to improved network conditions on a timescale of T. Figure 4.6 depicts the average throughput of 32 PCC flows, again with parameter T set to 60 seconds, and 32 TCP flows. A rather dynamic network environment is chosen where the loss rate increased abruptly from 2.5% to 5% from time 200 seconds to 300 seconds and from time 400 seconds to 420 seconds.

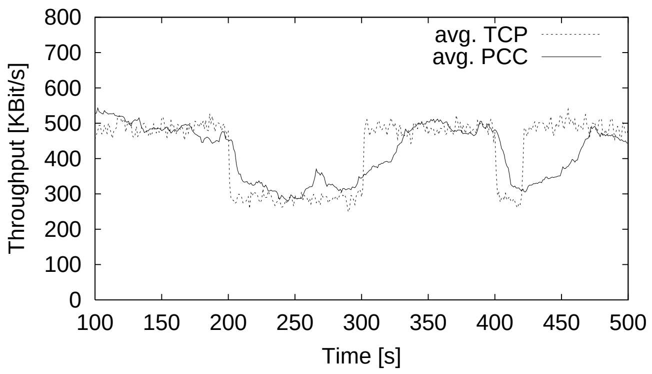


Figure 4.6: Loss bursts

When the loss rate changes at time 200 seconds, PCC does not adapt as fast as TCP but still achieves an overall average rate that is quite close to the TCP rate after only a few seconds. 60 seconds later we can see a little spike in the average PCC rate, resulting from the PCC flows that reenter the protected time T' to probe for bandwidth once their off-time is over. With a randomized offset as mentioned in Section 4.4.3, this spike can be avoided. Since the loss rate is still high at that time, the average PCC rate once again settles at the appropriate TCP-friendly rate shortly thereafter. As soon as the loss rate is reduced to its original value, the probability that suspended flows reentering the protected time will immediately be suspended again (and the probability that the random experiment of flows in the on-state will fail) decreases. Thus, after time 300 seconds, the random experiments of more and more flows succeed, until about 50 seconds later the TCP-friendly rate is reached again. Although PCC reacts more slowly than TCP, the average throughput of TCP and PCC up to time 350 seconds is very similar. In contrast to long periods with a high loss rate, short loss spikes (or in general a very high variability in the network conditions) hurt PCC performance much more than TCP performance. When the loss rate increases again at time 400 seconds, suspended PCC flows will stay in the off-state for at least 60 seconds, while the actual congestion persists for only 20 seconds. From the time the congestion ends until the time the PCC flows are allowed to reenter the protected time, TCP throughput is considerably higher than PCC throughput. However, we can also see from the graph that during periods of congestion PCC throughput does not quite drop to the level of TCP throughput but remains slightly higher. In Section 4.6.5 on PCC throughput for different application sending rates we will analyze this effect in more detail.

4.6.4 Varying Off Times

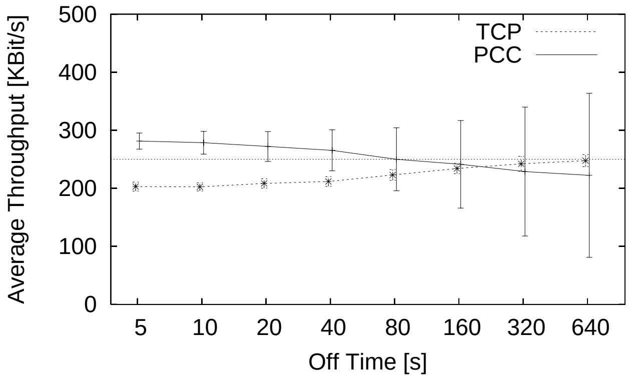
As discussed in the protocol section, the parameter T plays a major role in terms of usability of PCC and with respect to protocol fairness. To analyze the impact of the parameter, we set up a simulation scenario with 32 PCC and TCP flows over a common bottleneck of 32 MBit/s with a PCC application bit rate of 750 kBit/s. An additional 32 TCP flows are started every 200 seconds and stay on for 100 seconds for the whole duration of the experiment of 1000 seconds to introduce changes in the network conditions and force PCC to adjust the number of PCC flows.

Figure 4.7 shows TCP and PCC throughput and throughput variance for different T, averaged over a number of experiments. When T is low, the frequent protected times of PCC hurt TCP performance. The larger T gets, the less often PCC flows restart with a new protected time and the more likely it is that PCC flows cannot immediately make use of improved network conditions, since flows are suspended for longer periods of time. For $T \geq 160$, the latter effect outweighs the former, and PCC throughput drops below TCP throughput.[5] Furthermore, large values for T reduce PCC's intra-protocol fairness (see Section 4.6.2).

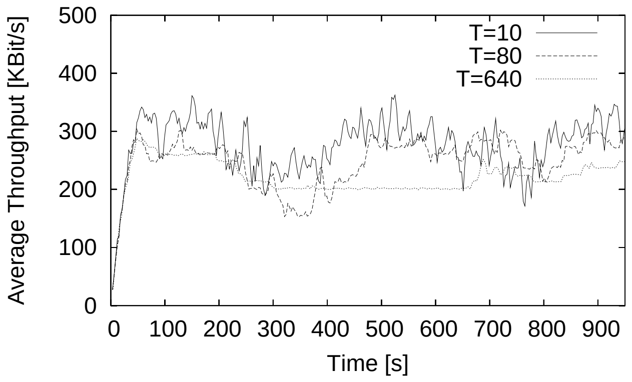
4.6.5 Fairness at Different Application Sending Rates

Ideally, no PCC flows should be suspended as long as the PCC application sending rate is below the TCP-friendly rate. For higher application sending rates the average PCC rate should remain at exactly the fair rate through the use of the random experiments. In Figure 4.8 we notice that an average PCC rate of exactly the fair rate is not reached when the application sending rate equals

[5]This value for T is not fixed for all network conditions but depends on the particular simulation setup (i.e., topology, number of flows, and buffer size at the bottleneck).



a) Average throughput



b) Throughput over time

Figure 4.7: PCC with different off times

the fair rate but for an application sending rate that is about 25% higher. The latter effect can be explained by PCC's susceptibility to dynamic network conditions. TCP's typical sawtooth-like sending rate results in variations in the network conditions which cause undue suspension of PCC flows.

When we compare the average PCC throughput to TCP throughput for high PCC application sending rates, we find that PCC throughput and thus PCC's aggressiveness continues to increase with the application sending rate once the fair rate has been reached. The effect of increased aggressiveness at higher application sending rates can be attributed to the TCP model used by PCC. As stated in [PFTK00], the TCP model is based on the so-called loss event rate. A loss event occurs when one or more packets are lost within a round-trip time, and the loss event rate is consequently defined as the ratio of loss events to the number of packets sent. The denominator of the loss-event rate increases as more and more packets are sent during a round-trip time due

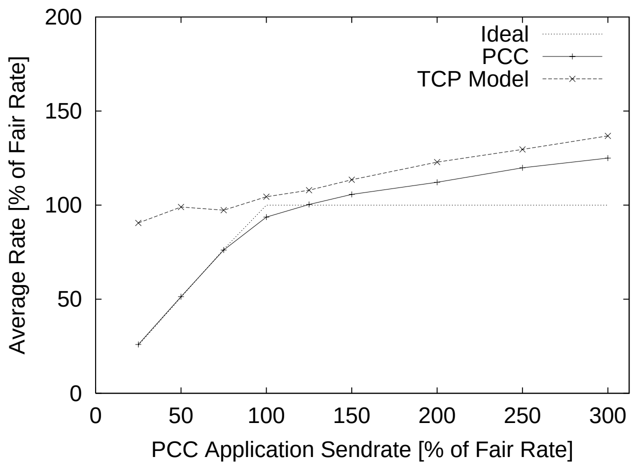


Figure 4.8: Comparison with estimated TCP-friendly rate

to a higher application sending rate. At the same time, the number of loss events does not increase to the same extent since more and more lost packets are aggregated to a single loss event. An in-depth analysis of this effect can be found in [RR99]. When relating the estimated TCP-friendly rate at different application sending rates to the average PCC rate achieved in these simulations, it becomes obvious that PCC's aggressiveness is not caused by PCC's congestion control mechanism but by the dependence of the TCP model on the measurement of the loss event rate at sending rates close to the actual TCP rate. If the measurement is performed at a sending rate that is much higher or lower than the rate of a TCP flow, the resulting loss event rate is likely to be different from the one experienced by TCP.

4.6.6 Low Levels of Statistical Multiplexing

As stated in Section 4.4.1, PCC is suitable for environments where the number of PCC flows is sufficiently high or PCC throughput represents only a small fraction of overall throughput on the bottleneck link. By continuously monitoring the loss rate, a PCC flow should be able to tell whether switching it on significantly changed the network conditions and thus the above requirements are not met. To analyze PCC behavior in such an environment, we chose the following simulation setup. Four PCC flows and four TCP flows compete for the same bottleneck link with a capacity of 2 MBit/s. The application rate is set to 500 kBit/s, so that the PCC flows alone are able to completely fill the link and a fair resource distribution is reached for two PCC flows in the on state.

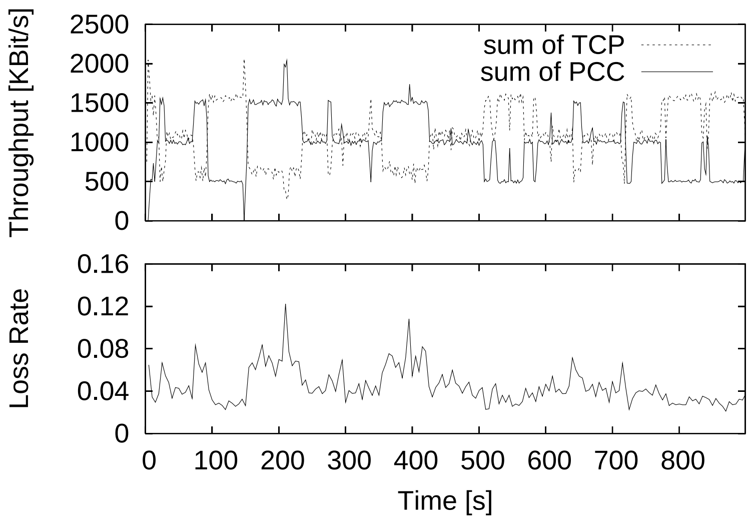


Figure 4.9: Throughput and loss rate

Figure 4.9 depicts cumulative PCC and TCP throughput as well as the overall loss rate at the bottleneck. Averaged over the simulation time, PCC and TCP achieve about the same throughput. About half the time, a fair bandwidth distribution is achieved. For most of the other half of the time, either one or three PCC flows are on. Only for a very short period of time does PCC occupy the link completely before backing off a few seconds later. Whenever a new PCC flow is switched on, we can observe a distinct increase in the loss rate at the bottleneck. Thus, loss monitoring would help to detect that traffic conditions are determined to a large degree by the PCC flows.

When running the same experiments with a higher number of TCP flows, the PCC flows only have a marginal effect on the network conditions and do not harm the TCP flows even if more than the ideal number of PCC flows are on at some point in time. If the number of PCC flows increases, the probability that far too many flows are on at a certain point in time is very small. Even more so, if many PCC flows compete with few TCP flows, the large variations in throughput caused by TCP's AIMD congestion control cause more than the fair number of PCC flows to be suspended, as discussed in Section 4.6.3.

4.6.7 Fairness for Different Combinations of Flows

Figure 4.10 shows the average throughput achieved by PCC for different combinations of PCC and TCP flows when the fair rate is 500 kBit/s and the application sending rate is 750 kBit/s.

Generally, PCC throughput increases with the number of TCP flows since the higher the level of statistical multiplexing, the lower the variations in the network conditions that degrade PCC performance. This effect is the more pronounced, the lower the number of PCC flows is. For a more detailed analysis of PCC and further network simulations we refer the reader to [Dam01].

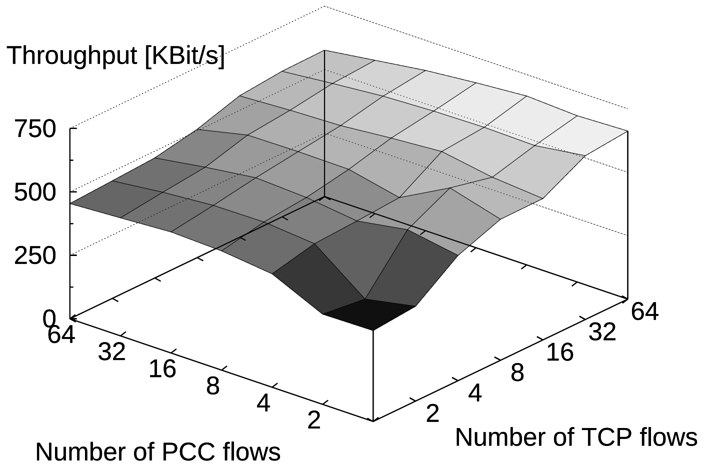


Figure 4.10: Average PCC throughput for different numbers of flows

PCC, together with the previously discussed TFRC protocol, can provide suitable congestion control for a wide variety of unicast applications. In particular, TFRC and PCC can be combined in a way that an application uses TFRC as long as the TCP-friendly rate is within the adaptive range of the application, and the application switches to PCC as soon as the TCP-friendly rate falls below the lower bound of the adaptive range. Whereas unicast is sufficient for many applications, some applications, in particular those streaming media to a number of participants, benefit significantly from the use of multicast routing instead of unicasting. In the remainder of this dissertation, we will identify requirements for and important components of multicast congestion control and present a comprehensive multicast congestion control framework that complements the unicast protocols presented thus far.

Chapter 5

A Short Introduction to Multicast

For unicasting, packets are routed through the network from a source to a single destination. Particularly in the area of group communication, a number of applications exist where multiple receivers form a group that has to be supplied with the same data. If the number of receivers is small compared to the number of nodes in the network, it is possible to unicast the data to each of the receivers. If most or all of the nodes of the network are receivers, broadcast is a useful paradigm. Multicast covers the middleground between those two extremes, where a group is composed of a substantial number of receivers but is still small compared to the size of the network. Sending a message to a group of receivers is called multicasting, and the corresponding algorithm to distribute the packets is called multicast routing. To this end, a multicast tree is constructed between the sender(s), and the receivers and data packets are duplicated and forwarded along multiple outgoing interfaces of a router, where the multicast tree splits. With multicast, a given packet is sent only once over each link of the multicast tree (Figure 5.1).

Deering and Cheriton proposed a receiver-based version of IP multicast in [DC90]. IP multicast follows the concepts of IP-style semantics (packets can be multicast at any time without connection setup), open groups (sources do not keep track of group membership, and any receiver is allowed to join), and dynamic groups (receivers can join and leave at any time during the multicast session). Multicast routing was first tested on a larger scale for the transmission of an IETF meeting in 1992. As of yet, widespread multicast deployment is complicated by the fact that it “requires a non-trivial amount of state and complexity in both core and edge routers” [Alm00] which contradicts the paradigm that complexity should be pushed towards the edges of the network. Also, error localization, network management, and accounting are difficult, and therefore ISP’s hesitate to deploy it. Nevertheless, recent efforts show an increased interest in the

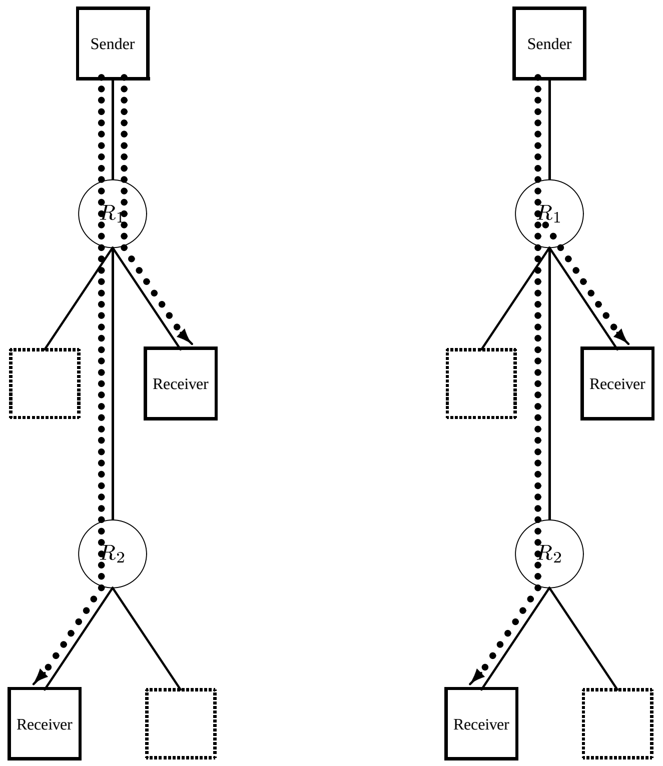


Figure 5.1: Unicast vs multicast data distribution

deployment of multicast IP [DLL+00]. The potential savings in bandwidth and the possibility to offer additional services to users may offset the additional management overhead incurred by multicast.

5.1 Multicast Routing

A multicast session is identified by the multicast group address. In IPv4, the address range from 224.0.0.0 to 239.255.255.255 is reserved for multicast. Receivers may join and leave a session at will and are included in and excluded from the multicast distribution tree accordingly. Various different multicast routing protocols can be used to construct such a distribution tree.

Multicast routing protocols can be divided into intra-domain protocols, to be run within an administrative domain (AD), and inter-domain protocols to be run between ADs. The first multicast routing protocol, used during the early stages of multicast deployment in the Internet, is DVMRP [WPD88]. DVMRP creates a reverse-shortest-path-tree and multicast packets are sent along the edges of the tree. If packets arrive at leaf routers that have subscribers for the multicast group, the packet is forwarded to the local area network, otherwise, a so-called prune message is generated and sent toward to the next router on the path towards the sender. Intermediate routers that have received prune messages on all their downstream interfaces generate a prune message themselves. Gradually, the multicast tree is pruned such that only routers on the paths to those receivers interested in the session continue to receive packets for the group. Periodically, prune state in the routers is discarded, and the whole network is flooded with data until pruning sets in again. DVMRP works well in dense networks where most of the receivers are part of the multicast group but the periodic flooding produces excessive overhead when the number of routers increases and only a small fraction of receivers is part of the group. More advanced dense mode routing protocols are MOSPF [Moy94], a multicast variant of the Open Shortest Path First routing algorithm for unicast, and Protocol Independent Multicast Dense Mode (PIM-DM) [ANS02].

For intra-domain routing in arbitrary (preferably sparsely populated) networks, a sparse mode version of PIM called PIM-SM [FHHK02, DEF$^+$96] is a promising solution. Receivers subscribe to a central rendezvous point (RP), and the subscription messages create forwarding state in the routers on the paths towards the RP. The multicast sender establishes a unicast tunnel with the RP, and from there, data packets sent to the group are only forwarded along network paths to actual receivers. Through the explicit join messages a flooding of the network is unnecessary. However, the RP is a single point of failure and a hot spot for traffic. The algorithm can be ameliorated by a bootstrap mechanism that selects alternative RPs in case one RP fails and by switching from a multicast tree rooted at the RP to a multicast tree rooted at the sender. Both options are supported in PIM-SM, as described in [FHHK02].

In addition to a multicast routing algorithm, IP multicast requires a group management protocol which enables receivers to join and leave groups. In IP networks, group management is provided by the Internet Group Management Protocol (IGMP) [Fen97], which is run between the receivers and the last-hop router. With IGMP packet forwarding by the last-hop router into his LAN is turned on and off.

5.2 Multicast Transport Protocols

Multicast IP only provides routing functionality in the way plain IP does for unicast. In addition, multicast transmission requires a transport protocol which in the simplest case is UDP. No multicast variant of TCP exists, but a number of enhanced transport services have been implemented on top of multicast UDP in case multicast applications need a more sophisticated transport layer.

The Real-Time Transport Protocol (RTP) [SCFJ96] is currently the most widely used multicast transport protocol in the Internet. It is designed to carry media flows such as video and audio streams. To be able to support a wide variety of different media types, RTP can be tailored to the specific needs of a medium using profiles and payload type definitions. It is complemented by the RTP Control Protocol (RTCP), which provides meta-information such as reception quality feedback, participant identification, etc. to all session participants. RTP is typically integrated into the application itself. It follows the concept of application level framing (ALF) [CT90] which preserves application data units as the pipelining units (i.e., data packets). Recent RTP drafts [SCFJ01, SC01] mandate the use of an appropriate congestion control mechanism, however no specific congestion control mechanism is specified since application requirements will vary depending on the type of media to be transported.

RTP is well suited to transport various types of non-interactive media streams, but more complex mechanisms are necessary for distributed interactive media, media that involve communication over a computer network as well as user interactions with the medium itself. RTP/I [MHKE01] is derived from RTP and reuses many of its components while it is adapted to meet the demands of distributed interactive media. By identifying and supporting the common aspects of distributed interactive media RTP/I allows the reuse of key functionality in form of generic services.

The Scalable Reliable Multicast (SRM) protocol described in [FJL$^+$97] provides a simple reliability mechanism for multicast transmission. The data as well as repair requests and responses are multicast to the receivers. Repair request and responses can be TTL limited or limited to a so-called local recovery group to reduce the burden on receivers not involved in the repair process. To prevent a missing packet from causing a flood of repair request (or a flood of subsequent responses), requests and responses are delayed by a random amount of time. If the another request is seen during that time, the timeout period is increased exponentially. A repair request is only sent if the missing packet is not repaired before the timer expires. Like RTP, SRM uses application-level framing for the packetization of the data.

The Pragmatic General Multicast (PGM) framework [SCG$^+$01] provides reliable, duplicate-free, ordered or unordered multicast transmission. PGM makes use of network elements (e.g., located at routers within the network) to provide aggregation of negative acknowledgements from the receivers, constrained forwarding of repair information, and similar tasks in a scalable fashion. Likewise, Generic Router Assist (GRA) [CST00], proposes generic mechanisms to be implemented at routers to assist transport protocols with the above tasks. Further protocols exist to describe, announce, and initiate multicast sessions [Han98, HJ98, HSSR99] which are beyond the scope of this dissertation. Instead, we will concentrate on components for multicast transport that will help such protocols to scale to very large receiver sets.

Many of the aforementioned protocols rely on feedback from the receivers and use some form of feedback suppression like SRM. If no network elements as in the PGM or GRA framework exist, the design of scalable multicast feedback mechanism that cater to the specific requirements of an application is not easy. In the next chapter we will discuss existing feedback control mechanisms and examine how they can be modified to meet specific application requirements.

These feedback control schemes also form the most important component of multicast congestion control. The lack of multicast congestion control has been identified as one of the factors inhibiting widespread deployment of multicast. Citing from [MRBP98], “in today’s Internet, reliable multicast protocols could do great damage through causing congestion disasters if they are widely used and do not provide adequate congestion control”. We will base the multicast congestion control mechanism that is developed in this dissertation on the aforementioned feedback suppression mechanisms.

Chapter 6

Multicast Feedback Control

6.1 Introduction

Many multicast protocols require receiver feedback. For example, feedback can be used for negative acknowledgements in reliable multicast [FJL$^+$97], for control and identification functionality for multicast transport protocols [SCFJ96], for status reporting from receivers for congestion control [Riz00], and for the reporting of allocation clashes in multicast address allocation mechanisms [Han98]. In such scenarios, the size of the receiver set is potentially very large. Sessions with thousands or even millions of participants may be common in the future, and without an appropriate feedback control mechanism severe feedback implosion is possible, overloading the sender and the network links.

Some multicast protocols arrange receivers in a tree hierarchy. This hierarchy can be used to aggregate receiver feedback at the inner nodes of the tree to effectively solve the feedback implosion problem. However, in many cases such a tree will not be available (e.g., for satellite links) or cannot be used for feedback aggregation (e.g., in networks without router support). For this reason, we will focus on feedback control using timer-based feedback suppression throughout the remainder of the chapter.

Pure end-to-end feedback suppression mechanisms do not need any additional support except from the end-systems themselves and can thus be used for arbitrary topologies. The basic mechanism of feedback suppression is to use random feedback timers at the receivers. Feedback is sent when the timer expires unless it is suppressed by a notification that another receiver (with a smaller timeout value for its feedback timer) already sent feedback.

The principal purpose of feedback suppression mechanisms is to prevent a feedback implosion in case feedback from a potentially very large group of responders is required. Usually, these mechanisms assume that it is not necessary to discriminate between feedback from different receivers. However, for many applications this is not the case; feedback from receivers with certain response values is preferred (e.g., highest loss or largest delay). We present modifications to timer-based feedback suppression mechanisms that introduce such a preference scheme to differentiate between receivers. The modifications preserve the desirable characteristic of reliably preventing a feedback implosion. Depending on the amount of knowledge about the distribution of the values to be reported we distinguish extremum detection and feedback bias. With the former we just detect extreme values without forcing early responses from receivers with extreme values. With the latter we exploit knowledge about the value distribution by biasing the timers of responders.

6.2 Related work

The necessity to employ scalable feedback algorithms in order to avoid feedback implosion has been recognized for a long time. Besides hierarchical [Gro97, Hof96, PSLB97] and token-based [CM83, CM84, WMK94] approaches several random distributions have been studied: Floyd et al. [FJL$^+$97, SEFJ97] use equally distributed timers for their SRM (Scalable Reliable Multicast) protocol. The duration of the response interval is adjusted according to the individual network latencies and the number of responses received. The latter method is inspired by various medium access protocols. Bolot, Turletti, and Wakeman [BTW94] use an exponentially growing subspace of randomly assigned keys for their IVS video-conferencing system.

The recent advancements in the deployment of multicast in the Internet have further stimulated the interest in large multicast groups. Nonnenmacher and Biersack [NB99] study the statistical properties of three different timer distributions. Based on analytical and simulation results they derive optimized parameters for the algorithms and recommend exponential feedback suppression as the one most suited for large groups. Lately some research has also been concerned with group size estimation based on feedback messages. Liu and Nonnenmacher [LN00] use the Poisson approximation for a maximum likelihood estimation of the group size. Friedman and Towsley [FT99] base their study on the binomial distribution.

The schemes discussed so far do not discriminate between different feedback values. Feedback from each receiver is equally important. A feedback scheme with a different focus, namely to gradually improve the values reported by the receivers, is presented in [BG99]. Receivers continuously give feedback to control the sending rate of a multicast transmission. Since the lowest rate of the previous round is known, feedback can be limited to receivers reporting this rate or a lower rate. It is necessary to further adjust the rate limit by the largest possible increase during one round to be able to react to improved network conditions. After several rounds, the sending rate will reflect the smallest feedback value of the receiver set. While not specifically addressed in the paper, this scheme could be used in combination with exponential feedback timers for suppression within the feedback rounds to reliably prevent a feedback implosion. However, with this scheme it may still take a number of rounds to obtain the optimum feedback value.

Other algorithms not directly concerned with feedback suppression but with the detection of extremal values have been studied in the context of medium access control and resource scheduling [WJ85, JW89]. The station allowed to use a shared resource is the one with the smallest contention parameter of all stations. A simple mechanism to determine this station is to use a window covering a subset of the possible contention parameters. Only stations with contention parameters within this window are allowed to respond and thus to compete for the resource. Depending on whether zero, one, or several stations respond, the window boundaries are adjusted until the window only contains the minimum contention parameter. In the above papers, strategies how to optimally adjust the window with respect to available knowledge about the distribution of the contention parameters are discussed.

To our knowledge, the only earlier work that is directly concerned with altering a non-topology based feedback suppression mechanism to solicit responses from receivers with specific metric values is presented in [DA01]. The authors discuss two different mechanisms, Targeted Slotting and Damping (TSD) and Targeted Iterative Probabilistic Polling (TIPP). For TSD, response values are divided into classes, and the feedback mechanism is adjusted such that response times for the classes do not overlap. Responders within a better class always get to respond earlier than lower-class responders. Thus, the delay before feedback is received increases linearly with the number of empty high classes. Furthermore, it is not possible to obtain real values as feedback without the assignment of classes. To prevent implosion when many receivers fall into the same class, the response interval of a single class is divided into subintervals and the receivers are randomly spread over these intervals. It was shown in [NB99, FW01] that a uniform distribution of response times scales very poorly to large receiver sets. TIPP provides better scalability by using a polling mechanism based on the scheme presented in [BTW94], thus having more favorable characteristics than uniform feedback timers. However, separate feedback rounds are still used for each possible feedback class. This results in very long feedback delays when the number of receivers is overestimated and the number of feedback classes is large. Underestimation will lead to feedback implosion. As a solution, the authors propose estimating the size of the receiver set before starting the actual feedback mechanism. Determining the size of the receiver set requires one or more feedback rounds. In contrast, the mechanisms discussed in this chapter only require a very rough upper bound on the number of receivers and will result in (close to) optimal feedback values within a single round. A further assumption for TSD and TIPP is that the distribution of the response values is known by the receivers. In most real scenarios this distribution is at best partially known or even completely unknown. If the distribution were known, a feedback mechanism could be built that would guarantee optimum response values and at the same time prevent feedback implosion. Such a mechanism is presented in section 6.8.3.

6.3 Analysis of Three Basic Feedback Algorithms

In this section, we examine three feedback algorithms that are prototypical for algorithms currently being deployed or recently proposed. They all address the *at-least-one* scenario in which a single response to a request suffices but multiple identical responses from different group members will do no harm except for the superfluous network load. An example is reliable multicast where a single feedback message suffices to trigger retransmission by the sender. We assume that the original message is multicast to the whole group, whereas the corresponding responses are unicast to the sender. In order to allow for the suppression of further responses, the sender confirms the reception by multicasting a confirmation back to the group.

Compared to the case where the responses themselves are multicast, this scenario causes less traffic but increases the network-latency.[1] Furthermore, it can also be applied in single-source multicast networks where only the sender but not the receivers can multicast packets. Multicasting or unicasting feedback does not affect the general suppression characteristics of the investigated feedback mechanisms. For the mathematical analysis we additionally assume the latency to be constant for all group members. The question of packet-loss and heterogeneous network latencies will be dealt with in the following sections. It will be shown that in the latter case the feedback latency is further reduced.

[1]Note that for e.g. unidirectional satellite links [DDI$^+$99, Fuh00] the network latency is *not* increased since all traffic is passed through a central network node.

6.3.1 Equally Distributed Feedback

The classical algorithm for feedback suppression with random timers uses equally distributed response probabilities. A typical implementation of this algorithm is SRM [FJL$^+$97]. The algorithm[2] can be described as follows:

Algorithm 1 *(Equally Distributed Feedback):*
Let T be a constant upper time limit. Upon reception of a feedback request, sample $x \in [0, 1)$ from a uniform distribution and start a timer t. If a feedback response is confirmed before $t \geq xT$ holds, the clock is stopped and no feedback response is sent. Otherwise, a response is sent as soon as the condition is satisfied.

Let n be the number of potential responders. We begin our analysis by noting that $(1 - x)^n$ is the probability that all x_i with $i = 1 \dots n$ are larger than x. Hence the probability that $x_{min} = \min\{x_1, \dots, x_n\} \in [x, x + dx]$ is $n(1 - x)^{n-1}dx$. The time corresponding to that choice of x is $t = xT$. Hence we obtain as expected value of the feedback latency L

$$E[D] = T \int_0^1 n(1 - x)^{n-1} x \, dx = \frac{T}{n + 1} \tag{6.1}$$

Since the feedback responses are distributed equally over the interval T the expected value for the number R of responses is given by

$$E[M] = n\frac{\tau}{T} \tag{6.2}$$

where τ is the network latency.

6.3.2 Independent Feedback Intervals

Let us now study an algorithm that makes use of of the concept of feedback intervals [Vog99]. It can be phrased as follows:

Algorithm 2 *(Independent Feedback Intervals):*
Let τ be the network latency and $p \in (0, 1]$ a constant. Upon reception of a feedback request sample $x \in [0, 1)$ from a uniform distribution, start a timer t, and immediately send a response

[2]We do not consider the adaptation of the answer interval to the group size and network-latency here. This topic will be discussed in the following sections.

if and only if $x < p$. If after the time $t = \tau$ no response has been confirmed start a new interval (i.e., act as if another feedback request was received).

Again, let n be the number of hosts that can send a response. The number of feedback intervals can easily be calculated as $1 + (1-p)^n + (1-p)^{2n} + \cdots = \frac{1}{1-(1-p)^n}$. Hence the number of additional intervals after the first interval is $\frac{1}{1-(1-p)^n} - 1 = \frac{(1-p)^n}{1-(1-p)^n}$. From this we immediately obtain the expected value for the feedback latency:

$$E[D] = \tau \frac{(1-p)^n}{1-(1-p)^n} \tag{6.3}$$

The algorithm does not guarantee the reception of a feedback message within a certain time limit T.

The expected value for the number of feedback responses is given by the product of the number of intervals and the expected value for each interval. The latter is given by np. Hence we obtain:

$$E[M] = \frac{np}{1-(1-p)^n} \tag{6.4}$$

This seems surprising since one might have expected np as result arguing that only the last interval contributes to the number of responses and thus the number of intervals must not be taken into account. Following this approach, one would however have to use the expected value of responses under the condition that at least one response is sent. Both approaches arrive at the same result.

A contour plot of the two expected values is shown in Figure 6.1. In order to simplify the comparison with other feedback algorithms the response probability p has been expressed as $p = 1/N$ which is a reasonable value.

6.3.3 Exponential Feedback Suppression

A third alternative originally proposed by Bolot, Turletti, and Wakeman [BTW94] and in an improved version extensively studied by Nonnenmacher and Biersack [NB99] is the exponential adaptation of the feedback probability (or the selection of the timer value from an exponential distribution). For feedback suppression with exponentially distributed timers, each receiver gives feedback according to the following mechanism:

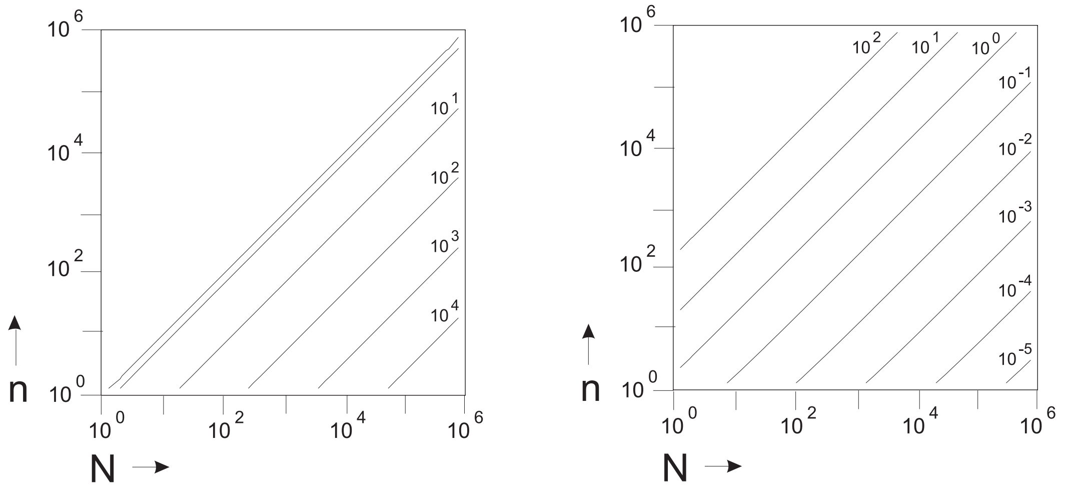


Figure 6.1: Feedback latency (left) and feedback responses (right) for independent feedback intervals (double-logarithmical plot). Contour lines *logarithmically* indicate values from $10^{-1}\tau$ to $10^4\tau$ and surplus responses from 10^2 to 10^{-5}.

Algorithm 3 *(Exponential Feedback Suppression):*
Let N be an estimated upper bound on the number of potential responders[3] *and T an upper bound on the time by which the sending of the feedback can be delayed in order to avoid feedback implosion.*

Upon receipt of a feedback request each receiver draws a random variable x uniformly distributed in $(0, 1]$ and sets its feedback timer to

$$t = T \max(0; 1 + \log_N x) \tag{6.5}$$

When a receiver notices that another receiver already gave feedback, it cancels its timer. If the feedback timer expires without the receiver having received such a notification, the receiver sends the feedback message.

Extending the suggestions in [NB99], this algorithms sets the parameter of the exponential distribution to its optimal value $\lambda = \ln N$ and additionally introduces an offset of N^{-1} at $t = 0$ into the distribution that further improves the feedback latency.

[3]The set of potential responders is formed by the participants that simultaneously want to give feedback. If no direct estimate is possible, N can be set to an upper bound on the size of the entire receiver set.

The expected delay until the first feedback is sent is

$$\begin{aligned} E[D] &= \frac{T}{\ln N}\int_{1/N}^{1}\frac{(1-x)^n}{x}dx \\ &\approx T(1-\log_N n) \end{aligned} \tag{6.6}$$

and the expected number of feedback messages sent is

$$E[M] = N^{\tau/T}\left(\frac{n}{N} + \left(1-\frac{1}{N}\right)^n - \left(1-\frac{1}{N^{\tau/T}}\right)^n\right) \tag{6.7}$$

where n is the actual number of receivers. A derivation of Equations (6.6) and (6.7) can be found in Appendix A. From Equation (6.7) we learn that $E[M]$ remains fairly constant over a large range of n (as long as $n \lesssim N$).

A plot of these two expected values is shown in Figure 6.2. In addition, Figure 6.3 shows a three-dimensional plot of $E[M]$ for different values of T and n, with $N = 10,000$.

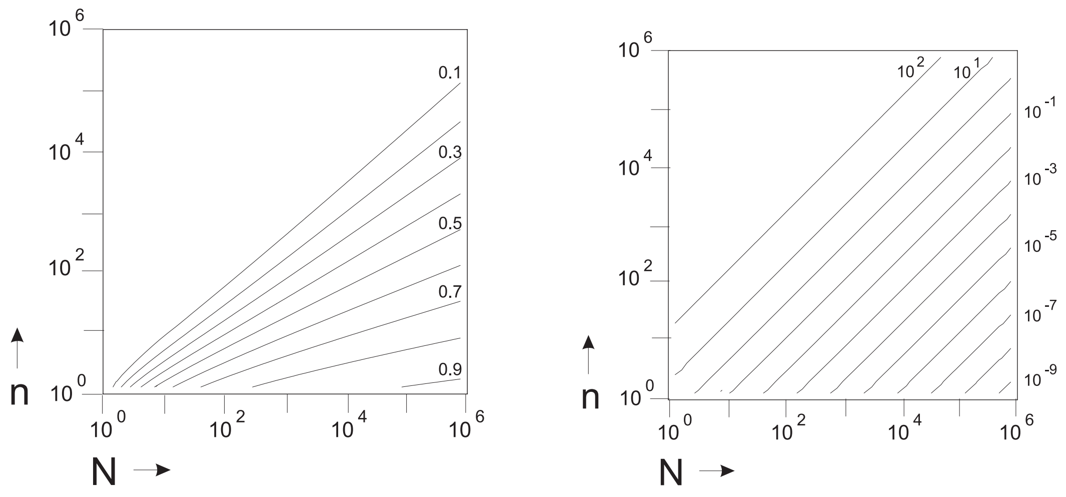


Figure 6.2: Feedback latency (left) and feedback responses (right) for Exponential Feedback Suppression (double-logarithmic plot). Contour lines *linearly* indicate values from $0.1T$ to $0.9T$ and *logarithmically* indicate surplus responses from 10^2 to 10^{-9}.

6.4 Behavior of the Algorithms for Very Large Groups

Based on the results derived above we can now compare the suitability of these three feedback algorithms in the context of very large networks. Concrete values for given n and N up to 10^6 can

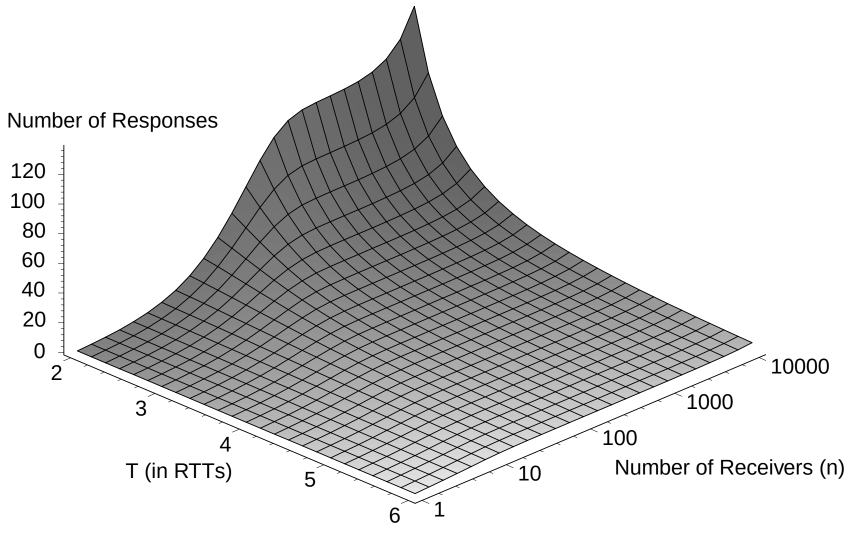


Figure 6.3: Expected number of feedback messages (with $N = 10,000$)

be read off from Figures 6.1 and 6.2. For a more thorough judgment of the algorithms' behavior in the limit of very large groups we will further analyze the formulae derived above.

As one might expect, we need some estimation of the group size N in order to optimize the algorithms' parameters. However, good group size estimations cannot always be found. Consider for example a reliable multicast transmission. Depending on where the packet loss occurs the number of hosts that need to send a negative acknowledgement (NACK) can vary greatly from packet to packet. If the loss occurs near the multicast sender almost all receivers might be potential NACK-senders. Otherwise, only a few hosts need to send a NACK.

Owing to this fact it is important to design a feedback algorithm that is insensitive to large variations of the group size. However, concerning gross underestimation of the group size a fixed limit can be given that no algorithm studied here (i.e., an algorithm based on independent identical hosts in a network with non-vanishing latency) can overcome:

Lemma *Underestimation of the group size results in an asymptotically linear increase of the number of feedback responses.*

Proof: Let $P(t)$ be the probability of answering before time t. Without loss of generality we can assume that $P(t) > 0$ for $t > 0$. Then $P(\tau) > 0$ is the finite probability for a host to answer before the suppression mechanism can be effective. Hence $E[M] \geq nP(\tau)$. On the other hand $\forall \epsilon > 0 : \lim_{n\to\infty} nP(\epsilon) > 1$. Thus for $n \to \infty$ suppression immediately sets in at $t = \tau$, and we have $E[M] = nP(\tau)$.

In order to avoid this linear behavior it is desirable to operate the system such that our estimation is an upper limit for the group size. Unlike the actual group size such a limit can usually be estimated rather easily (e.g., one can assume that the total number of installed hosts is known to the network provider). Due to this characteristic it is hence crucial for the feedback algorithms to be insensitive to overestimations of the group size.

6.4.1 Equally Distributed Feedback

The equally distributed feedback algorithm shows a rather simple behavior in the limit for large groups. Since $\lim_{n\to\infty} E[M] = \infty$ for fixed T we have to adapt T to the group size. Generally, by eliminating T we find a trade-off between feedback-latency $E[D]$ and response duplicates $E[M]$:

$$\lim_{n\to\infty} E[D]\, E[M] = \tau \tag{6.8}$$

Although we expect a feedback latency on the order of the network latency and a number of response duplicates of order one, the accuracy of our estimation N of the group size is crucial. Overestimation or underestimation of N linearly affects both $E[D]$ and $E[M]$. Thus this algorithm is not well suited for very large networks.

6.4.2 Independent Feedback Intervals

As mentioned above, collecting responses for immediate sending at the beginning of each interval can reduce the feedback latency by up to τ while it ideally preserves the number of responses. However, rather than using this simple method we investigate a slightly different algorithm that uses *independent* intervals.

Let us now analyze this algorithm in the limit of large groups: Setting $p = \frac{1}{N}$ and $n = \alpha N$ we have in the limit $N \to \infty$

$$E[D_\infty] = \tau \frac{e^{-\alpha}}{1 - e^{-\alpha}} = \frac{\tau}{e^{\alpha} - 1} \tag{6.9}$$

$$E[M_\infty] = \frac{\alpha}{1 - e^{-\alpha}} \tag{6.10}$$

As expected we see that if we underestimate the size of the group ($\alpha \gg 1$) the number of feedback responses grows asymptotically linearly while the feedback latency vanishes. However, if we overestimate the size of the group ($\alpha \ll 1$) we see that the feedback latency grows according to $E[D] \simeq \frac{\tau}{\alpha+\cdots}$ while the number of feedback responses $E[M] \to 1$.

As with the equally distributed feedback no choice of parameters can guarantee a controllable behavior for all $\alpha \in [0, 1]$. Even worse, due to the independence of the feedback intervals we have $E[D]\, E[M] \simeq \frac{\tau}{\alpha}$ for $\alpha \ll 1$. Thus, this algorithm is as unrecommendable as the previous one for the situation under investigation.

6.4.3 Exponential Feedback Suppression

As above we set $n = \alpha N$ for our analysis of the $N \to \infty$ limit. Additionally, we choose T such that $N^{\tau/T} = e^{\beta} =$ const, i.e., we set

$$\beta = \frac{\tau}{T} \ln N \tag{6.11}$$

With this adaption we now have

$$E[M_\infty] = (\alpha + e^{-\alpha})e^{\beta} \tag{6.12}$$

From the construction of the algorithm we know that T is an upper limit for the feedback latency. With Equation (6.6) we find accordingly

$$E[D_\infty] \approx T(1 - \log_N(\alpha N)) = T \log_N \frac{1}{\alpha} \tag{6.13}$$

Hence, unlike in the two previous algorithms both expected values remain rather insensitive to variations of α. In the limit $\alpha \to 1/N$ and $\alpha \to 1$ both expressions remain finite. This is a strong indication that this mechanism is well suited for very large networks. According to (6.12) the number of duplicates changes only by a factor of 1.3679 between the two extreme cases $n = 1$ and $n = N$. Additionally, even in the worst case scenario the feedback latency remains below the threshold of T. From (6.11) we read that the choice of this threshold also determines the number of expected feedback duplicates.

6.5 Unifying the Feedback Algorithms

Based on the insights gained so far we will now discuss a generalized feedback algorithm that unifies the properties of the three algorithms.

6.5.1 Discrete Feedback Intervals Versus Continuous Feedback Suppression

Comparing the algorithms previously studied we see that sending a response at time t with $0 < t < \tau$ is suboptimal since no suppression can take place before $t = \tau$ but the latency is increased compared to an immediate response at $t = 0$. The second algorithm respects this fact by sending feedback in intervals. It sends a certain number of responses only at the beginning of a feedback interval of length τ. No further responses are sent before the beginning of the next interval.

With the help of this insight both of the other algorithms can be improved as well. A straightforward general implementation of this mechanism can be phrased as follows:

Algorithm for feedback intervals *Divide the interval $[0, T]$ into sub-intervals $[k\tau, (k+1)\tau]$ where $k \in \{0, \ldots, \lfloor \frac{T}{\tau} \rfloor\}$. Send all responses that would be sent in a sub-interval at the beginning of that sub-interval.*

As we will see, this general mechanism reduces the feedback latency while it ideally preserves the number of response duplicates. A detailed analysis of a specific version of exponential feedback suppression with intervals can be found in Bolot et al. [BTW94]. Here, we will therefore focus on more principal aspects.

As depicted in Figure 6.4, the general mechanism for the introduction of feedback intervals raises the response probability for each host at $t = k\tau$ to $P(t+\tau)$. That means that the hosts' response probability is given by a step-function $P_s(t)$ that lies between the original function $P_0(t)$ and $P_1(t) = P_0(t+\tau)$.

Since generally

$$\begin{aligned} E[D] &= \int_0^T t \cdot n(1-P(t))^{n-1} P'(t)\, dt && (6.14) \\ &= \int_0^T (1-P(t))^n \, dt && (6.15) \end{aligned}$$

the feedback latency is given by the area above the function $\hat{P}(t)$ where $\hat{P}(t) = 1-(1-P(t))^n$. Noting that P_1 results in a feedback latency that is reduced by τ as compared to P_0 the feedback latency is generally reduced by ΔL with $0 < \Delta L < \tau$. For $T \gg \tau$ the area between the functions $\hat{P}_0$ and $\hat{P}_s$ approximates the area between the functions $\hat{P}_1$ and $\hat{P}_s$, and we have $\Delta L \simeq \frac{\tau}{2}$.

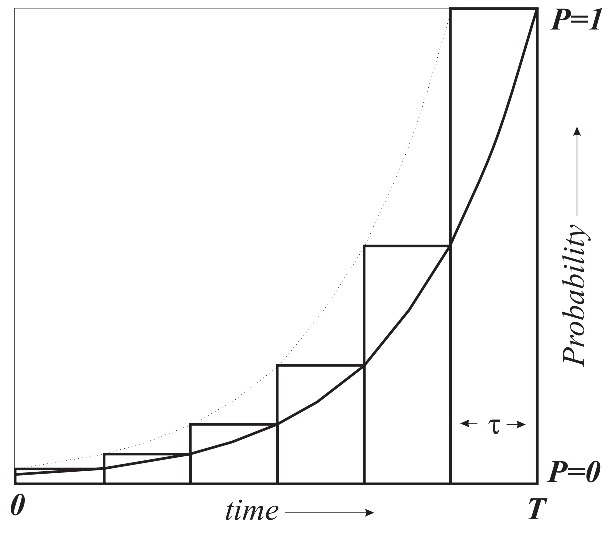


Figure 6.4: Exponential feedback with and without intervals.

However, if we slightly underestimate the network latency τ the hosts will perform an additional superfluous feedback interval. This will result in an increase of feedback responses by a factor of $N^{\tau/T}$. In networks with heterogeneous latencies we must hence either choose τ to be the maximal latency, or a sub-group of hosts will perform an additional superfluous feedback interval. Since on the other hand overestimation of the network latency leads to a linearly increased feedback latency, τ should not be chosen too generously. Resolving this trade-off thus requires a rather good knowledge of the network latencies which is not necessary in the case of continuous feedback.

6.5.2 A Generalized Feedback Algorithm

Having formulated a general mechanism to transform a continuous feedback algorithm into an interval-based feedback algorithm, we can now more deeply analyze the relationship between the algorithms studied above. In order to do so, we transform all three algorithms into the same representation. The specific form of the algorithm is given by a distribution function $f(t)$ which determines the feedback behavior.

General feedback algorithm *Upon reception of a feedback request, sample $x \in [0, 1)$ from a uniform distribution. Send a feedback response as soon as $x \leq f(t)$ where t denotes the time after the reception of the request. If however another group member's response has been confirmed before that time no response should be sent.*

As stated above, the distribution function f is decisive for the feedback behavior. For the first algorithm given in Section 6.3 we read off $f(t) = t/T$ and similarly $f(t) = N^{t/T-1}$ for the third. For the second algorithm f is determined by the probability for a group member to respond before or in the k^{th} interval. In the first interval the probability to respond is p. Hence $f(t) = p$ for $0 < t < \tau$. In order to calculate f for the further intervals we note that $(1-p)^{k+1}$ is the probability that no response is sent before interval $k+1$. Hence the probability that the first response is sent before or in the k^{th} interval is $1-(1-p)^{k+1}$, i.e., $f(t) = 1-(1-p)^{k+1}$ for $k\tau < t < (k+1)\tau$.

In order to simplify the analysis we will now drop the notion of intervals and consider the continuous distribution function $f(t) = 1-(1-p)^{t/\tau+1}$. As already stated, the error that is hereby introduced slightly increases the number of response duplicates while reducing feedback latency. An interval-based algorithm can always be recovered by application of the mechanism described above.

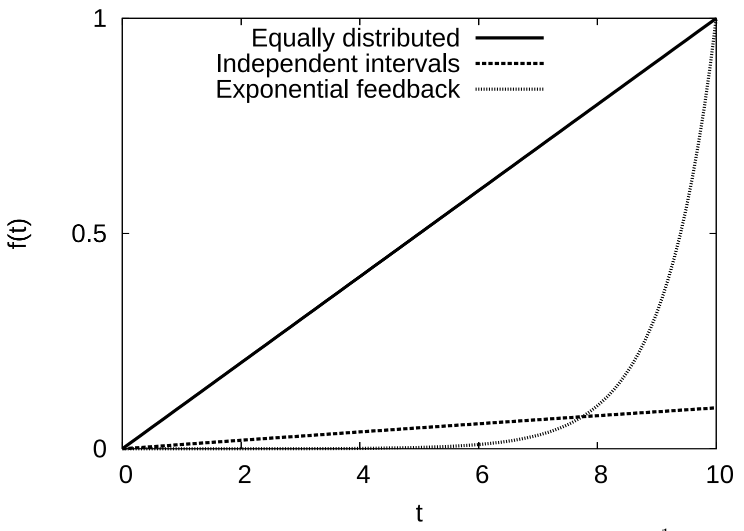
Let $p(t)dt$ be the probability for a host to respond in the time interval $[t, t+dt]$ under the condition that no response was confirmed before t (i.e., no message indicating a response was heard before t). Then the probability $f(t)$ that a response is confirmed by t is determined by the following differential equation:

$$f'(t+\tau) = p(t)(1-f(t)) \tag{6.16}$$

An overview of the three algorithms is given in Figure 6.5.

Assuming that we know the actual size of the group we can always choose $p(0)$ such that with any desired probability at least one response is sent immediately after the request was received. By this, the feedback latency is minimized. Only in the unlikely case that no response is sent at $t = 0$ the feedback process continues. If we are sure about the group size and no overall time limit is given, there is no reason to modify the response probability over time. Hence $p(t) = const$, i.e., the second algorithm is optimal.

If a time limit T is given, we have to choose the response probability such that $\lim_{t\to T} p(t) \to \infty$. This is guaranteed by the first and the third algorithm. The latter furthermore takes into account, that not receiving response confirmations can also be caused by an overestimation of the group size and accordingly adjusts p.



Equally distributed feedback	$f(t) = t/T$	$p(t) = \frac{1}{T-t}$
Independent feedback intervals	$f(t) = 1 - (1 - \frac{1}{N})^{t/\tau+1}$	$p(t) = \frac{1}{\tau}\frac{N-1}{N}\ln\frac{N}{N-1}$
Exponential feedback suppression	$f(t) = N^{t/T-1}$	$p(t) = \frac{N^{\tau/T}\ln N}{T}\frac{1}{N^{1-t/T}-1}$

Figure 6.5: Distribution functions for the three feedback algorithms with $N = 100000$, $T = 10$, and $\tau = 0.001$ (note that the ratio of τ to T is unrealistic for most real-world scenarios and has only been chosen to better visualize independent feedback intervals.)

6.6 General Considerations

6.6.1 Unicast vs. Multicast Feedback Channels

When receivers are able to multicast packets to all other receivers, feedback cancellation is immediate in that the feedback that ends the feedback round is received by other receivers at roughly the same time as by the sender.

However, the mechanism described in the previous section also works in environments where only the sender has multicast capabilities, such as in many satellite networks or networks where source-specific multicast [FHHK00] is deployed. In such environments, feedback is first unicast back to the sender which then multicasts a feedback cancellation message to all receivers. This incurs an additional delay, thus roughly doubling the feedback latency of the system (in the case of symmetric transmission delays between the sender and the receivers and among the receivers themselves.)

6.6.2 Removing Latency Bias

Plain exponential feedback favors low-latency receivers since they get the feedback request earlier and are thus more likely to suppress other feedback. In case the receivers know their own latency τ as well as an upper bound on the latency for all receivers τ_{max}, it is possible to remove this bias. Receivers simply schedule the sending of the feedback message for time $t + (\tau_{max} - \tau)$ instead of t.

In fact, this unbiasing itself introduces a slight bias *against* low-latency receivers in case unicast feedback channels are used. While the first feedback message is unaffected, subsequent duplicates are more likely to come from high-latency receivers, since they will receive the feedback suppression notification from the sender later in time.

If it is not necessary to remove the latency bias, the additional receiver heterogeneity generally improves the suppression characteristics of the feedback mechanism, as demonstrated in [NB99].

6.6.3 Impact of Packet Loss

Since all algorithms discussed here are based on independently acting hosts, a lost response packet does not harm the principal effectiveness of the feedback mechanism. If the response is lost before it is confirmed by the sender, a loss rate of p merely reduces the effective group size from n to $(1 - p)n$. If on the other hand a fraction q of the group receives the feedback, the effective group size is further reduced to $(1 - p)(1 - q)n$. Due to the algorithms' insensitivity to gross variations in the group size, packet loss only marginally affects the results given above.

In order to safeguard against loss of feedback cancellation messages with unicast feedback channels, we note that it may be necessary to let the sender send multiple cancellation messages in case multiple responses arrive at the sender and/or to repeat the previous cancellation message after a certain time interval. Loss of cancellation messages in unicast feedback is critical since a delayed feedback cancellation is very likely to provoke a feedback implosion.

6.6.4 Message Piggybacking

The feedback requests and the cancellation messages from the sender can both be piggybacked on data packets to minimize network overhead. In case a unicast feedback channel is used, the piggybacking of cancellation messages has to be done with great care. The delay introduced by piggybacking at low sending rates may again provoke a feedback implosion. This undesired behavior is likely to occur when the inter-packet spacing between data packets gets close to the maximum feedback delay.

The problem can be prevented by introducing an upper bound on the amount of time by which a cancellation message can be delayed and sending a separate cancellation message when necessary. If separate cancellation messages are undesirable, the maximum feedback delay T has to be increased in proportion to the time interval between successive data packets.

6.7 Simulation Results

To investigate the applicability of the different feedback mechanisms discussed in Section 6.3, a simulation model of the algorithms with unicast feedback channels was studied. For an upper limit of $N = 10^6$ hosts and a maximum feedback delay of $T = 10\tau$ we examined groups with $n = 1$ to $n = 10^6$ actual hosts. To be able to abstract from a specific network topology and independent receiver-to-receiver delays, we considered the case of unicast feedback to the sender as discussed in Section 6.3. Note that this increases the feedback delay and the number of responses and thus represents an upper bound for the general scenario. All results were averaged over 2000 simulation runs to minimize the impact of statistical errors.

The feedback mechanism with independent feedback intervals and constant feedback probability has optimal characteristics when the number of participants is known (i.e., $p = 1/n$). As shown in Figure 6.6, the average number of feedback responses is lower than that of all other mechanisms without impairing the feedback latency. However, in most cases the group size is unknown to the sender (or the feedback mechanism itself is used to estimate the group size [LN00]), and an estimated N has to be used. As discussed in Section 6.3.2, the feedback latency increases proportionally to the ratio of N to n which makes the mechanism unsuitable for such scenarios.

Equally distributed feedback does not take the group size into account and is thus not affected by inaccurate group size estimations. While it provides the lowest feedback latency, it cannot prevent a feedback implosion at the sender.

For exponentially distributed feedback, the number of feedback responses only varies within the limit of the statistical errors over several orders of magnitude. For groups with more than $0.2 \cdot 10^6$ hosts, the values rise slightly above the large plateau of the average fifteen response

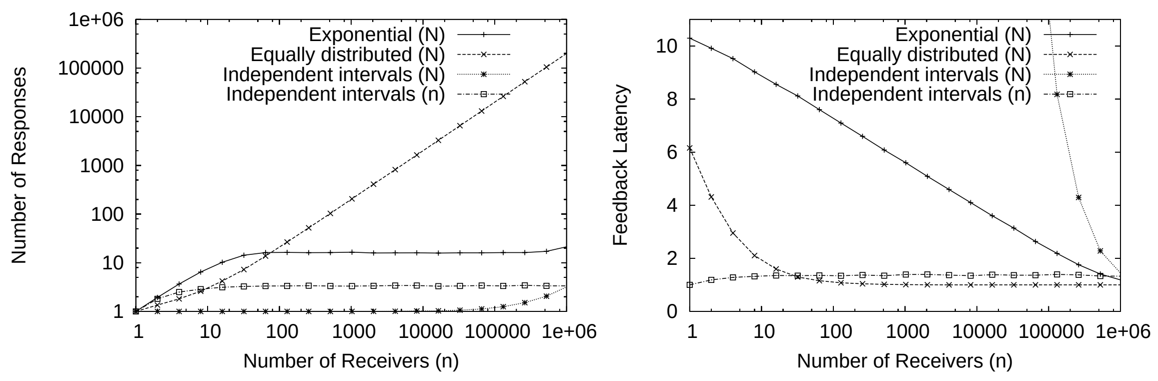


Figure 6.6: Number of responses and feedback latency for the different feedback mechanisms

messages. This increase perfectly complies with the theoretical prediction. Below about 20-30 hosts, the large-group limit no longer applies and the observed number of responses drops below the theoretical value. The observed feedback latency drops exponentially with the number of group members. This behavior was also expected from our mathematical analysis.

These advantageous properties hence recommend the exponential algorithm especially for scenarios with largely varying group sizes.

6.7.1 Heterogeneous Network Latencies

Simulation results to assess the impact of heterogeneous network delays are depicted in Figure 6.7. We assume a uniform distribution of the delays with variations ranging from $\pm 0\%$ to $\pm 90\%$ of the average delay from the sender to the receivers. The higher the variation in the network delays the lower the number of feedback messages, since more feedback can be suppressed by receivers with a low delay.

Heterogeneous network delays also significantly reduce the amount of feedback with the equally distributed feedback mechanism. However, heterogeneity has little impact when independent feedback intervals are used since the number of feedback messages is already very small.

For all mechanisms, the average feedback delay is slightly reduced by the delay variations. In our simulations, the feedback mechanisms thus profited from network heterogeneity.

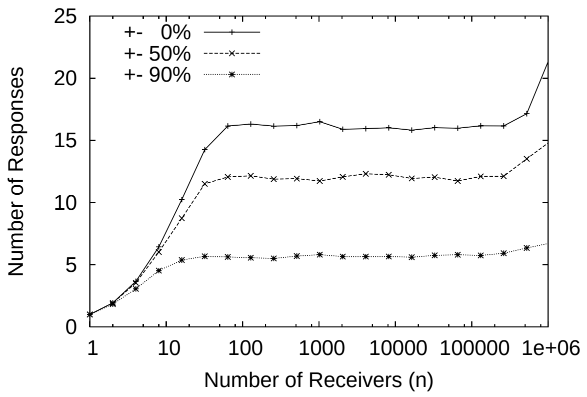


Figure 6.7: Exponential feedback with heterogeneous network delay

6.7.2 Discrete Feedback Intervals

If feedback is sent in intervals at discrete feedback times, the properties of the different mechanisms can be improved for environments with homogeneous network delays.

However, delay heterogeneity adversely affects the feedback properties. Comparing feedback responses of interval-based and continuous feedback mechanisms for increasing delay variations reveals the deficiencies of interval-based feedback. Figure 6.8 depicts the ratio of the number of responses with continuous exponential feedback to the number of responses with interval-based feedback. For a constant network delay, the number of feedback messages can be reduced by up to a factor of 6 when using feedback intervals. Network delay variations of more than ±90% reduce this ratio to 0.7, indicating that discrete feedback intervals result in an increase of the number of responses at the sender. Generally, interval-based feedback can be used to reduce the number of responses when the network latency is known and varies by less than an order of magnitude.

6.7.3 Impact of Clock Granularity

The granularity of the clock used for the feedback timer at the receivers influences the expected number of responses in the same way as the delay introduced by piggybacking. If all the receivers respond later in time due to coarse clock granularities, the length of the time interval over which responses are spread out for the feedback suppression decreases. Similarly, if in the case of unicast feedback channels the sender does not immediately react to receiver feedback but polls the network interface at certain time intervals, the duration of these time intervals has an impact

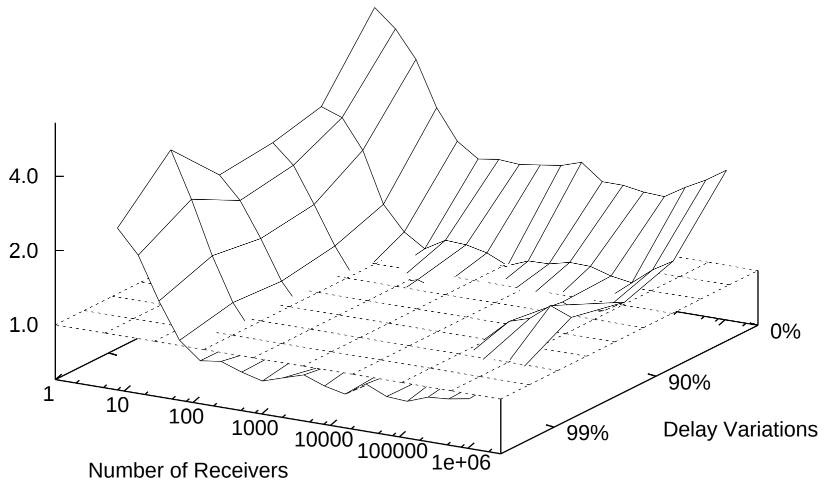


Figure 6.8: Ratio of the number of responses with/without feedback intervals

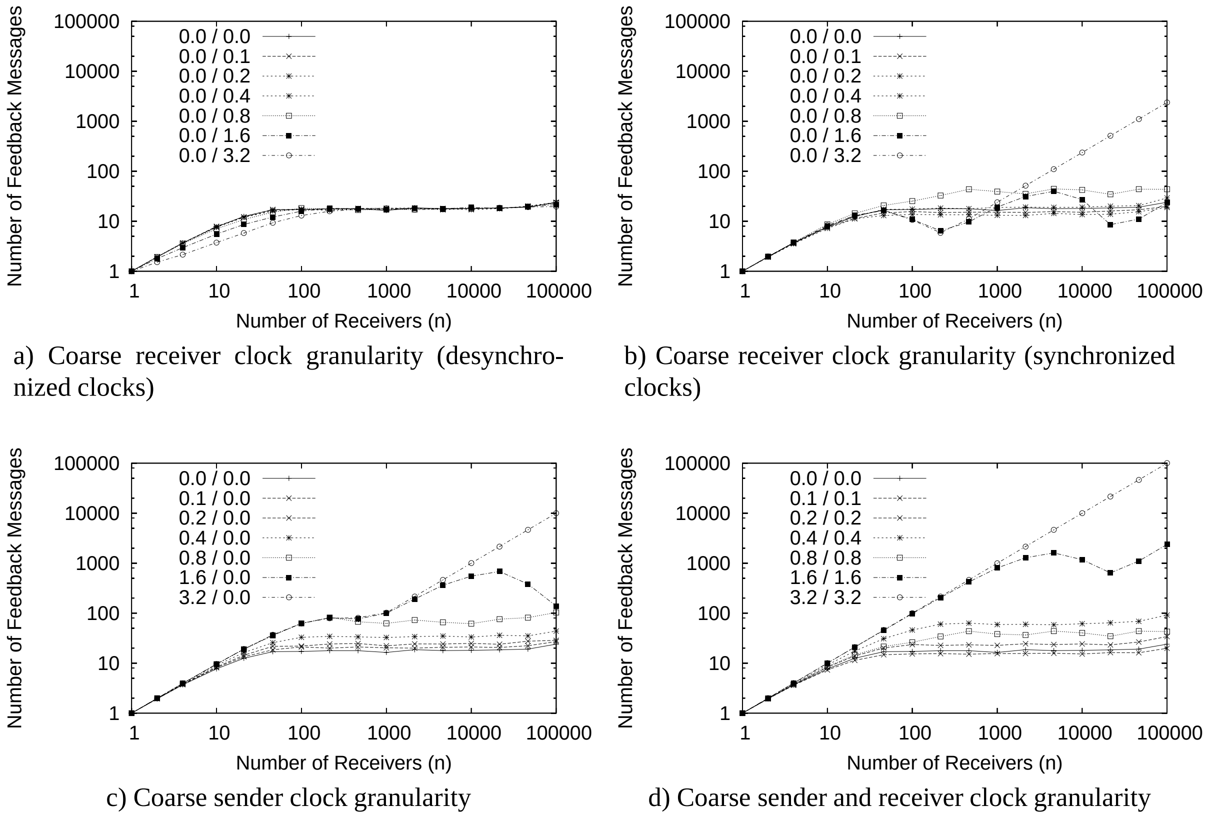
on the number of responses. Note that if feedback from the receivers is multicast, the sender does not play a role in the feedback suppression process and consequently the sender's clock granularity can be neglected.

Figure 6.9 shows the dependence of the average number of responses on the clock granularity of the sender and the receivers. Curves are plotted for a feedback delay of $T = 4\tau$ and clock granularities of 0.0τ (i.e., accurate clocks), 0.1τ, 0.4τ, 0.8τ, 1.6τ, and 3.2τ. If receiver clocks have a coarse granularity but are not synchronized, feedback may be delayed but there is no implosion of feedback messages (Figure 6.9a). Otherwise, the average number of feedback messages increases with a coarser clock granularity but only for a clock granularities close to the feedback delay T can we see a substantial increase in the number of responses. While all curves a fairly alike for fine-grained clocks with only a moderate increase in responses, there are striking differences in the runs of the curves for the very large clock granularity values of 1.6τ and 3.2τ. For coarse (synchronized) receiver clocks, a clock granularity of 1.6τ even reduces the number of responses for some receiver set sizes, and only for 3.2τ there is a significant increase in the number of responses (Figure 6.9b). The increase in the number of responses when the sender does not send suppression messages immediately but only at certain time intervals with a duration close to the feedback delay, the increase in the number of responses is more pronounced (Figure 6.9c). As is to be expected, the number of feedback messages is even higher than in the previous two cases when sender as well as receivers may delay messages because of coarse clock granularities (Figure 6.9d). Here, for a clock granularity of 3.2τ there is virtually no suppression possible and all receivers of the group respond to feedback requests.

These differences in the number of responses for large clock granularity values do not stem from an insufficient number of simulations (the results were averaged over 1000 runs each) but are a consequence of the distribution of response times. The probability mass functions of the response times for the experiments with a clock granularity of 1.6τ and for receiver set sizes between 1000 and 100,000 are depicted in Figure 6.10. When the sender has this coarse clock granularity, suppression is first possible after $1.6\tau + 0.5\tau$ (to account for the one-way delay from the receivers to the sender). All receivers within this interval cannot be suppressed, but if one or more receivers fall into this interval they will suppress feedback in later intervals (Figure 6.10a). As a consequence, there is a large drop in the probability mass function after 2.1τ. If no feedback was given within the first interval, no suppression is possible for the second interval, and we can again see the build-up in feedback responses towards $3.2\tau + 0.5\tau$ due to the exponential distribution of feedback timers. Later receivers will almost always be suppressed unless the number of receivers is very small, and for that reason no receivers have timer values within the first two intervals. The higher the number of receivers, the more likely it is that some of them have timers within the early intervals. This explains the drop in the number of responses around 20,000 receivers in Figure 6.9c.

For coarse receiver clocks, we can see distinct spikes in the distribution instead of a smooth mass function as receivers can only respond at certain times (Figure 6.10b, bottom). The higher the number of receivers, the more pronounced is the first spike. For $n \geq 46415$, there is always a timer that falls into the first interval, and receivers in later intervals are always suppressed. With this fact we can explain why the number of feedback messages *decreases* for receiver set sizes of 10,000 and above. It becomes more and more likely that (a few) receivers fall into the first interval, suppressing the large bulk of receivers in later intervals. If no timers fall into the first interval, the number of receivers with timers in the second interval will be much larger, and fewer receivers will be suppressed.

The results for the simulations where sender and receivers have coarse clock granularities can be explained simply by combining the above distributions. Receivers will respond only at certain points in time, and the additional delay at the sender causes the distribution of response times to be shifted compared to the distribution discussed in the previous paragraph, resulting in much fewer responses being suppressed.



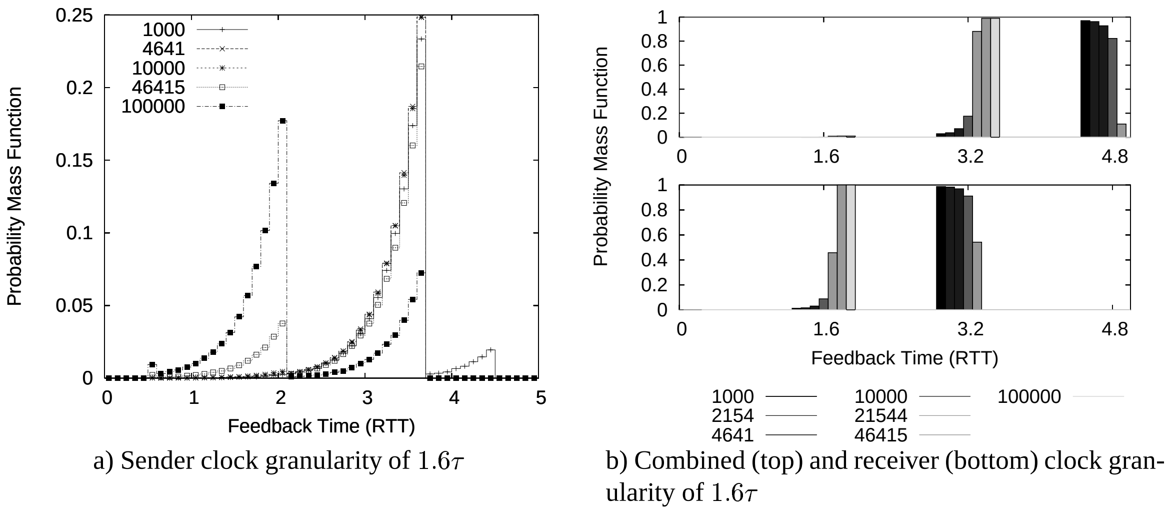
a) Coarse receiver clock granularity (desynchronized clocks)

b) Coarse receiver clock granularity (synchronized clocks)

c) Coarse sender clock granularity

d) Coarse sender and receiver clock granularity

Figure 6.9: Double logarithmical plot of the impact of receiver and sender clock granularity on the number of responses (accurate clocks have a granularity of 0.0)



a) Sender clock granularity of 1.6τ

b) Combined (top) and receiver (bottom) clock granularity of 1.6τ

Figure 6.10: Distribution of response times

We conclude that as long as receiver clocks are desynchronized or clock granularities are a sufficiently small fraction of the feedback delay, a coarse clock granularity does not endanger feedback suppression. Only with clock granularities close to the feedback delay can we observe a significant increase in the number of feedback messages.

6.8 Value-Based Feedback

The mechanisms presented so far assume that there is no preference as to which receivers send feedback. All feedback messages are the same. As we will see, for many applications this is not sufficient. Those applications require the feedback to reflect an extreme value for some parameter within the group [WF01]. Multicast congestion control, for example, needs to get feedback from the receiver(s) experiencing the worst network conditions. Other examples are the polling of a large number of sensors for extreme values, online auctions where one is interested in the highest bids, and the detection of resource availability in very large distributed systems.

In the remainder of this chapter, we propose several algorithms that favor feedback from receivers with certain characteristics while preserving the feedback implosion avoidance of the original feedback mechanism. Our algorithms can therefore be used to report extrema from very large multicast groups.

6.8.1 Extremum Detection

Let us now consider the case where not only an arbitrary response from the group is required but an extremum value for some parameter from within the group. Depending on the purpose the required extremum can be either a maximum or a minimum. Without loss of generality we will formulate all algorithms as maximum detection algorithms.

An obvious approach to introduce a feedback preference scheme is to extend the normal exponential feedback mechanism with the following algorithm:

Algorithm 4 *(Basic Extremum Detection):*
Let $v_1 > v_2 > \cdots > v_k > 0$ *be the set of response values of the receivers.*

Upon receipt of a feedback request each receiver sets a feedback timer according to Algorithm 3. When a receiver with value v *is notified that another receiver already gave feedback with* $v' \geq v$*,* *it cancels its timer. Otherwise, when the feedback timer expires (i.e., for all previous notifications $v' < v$ or no notifications were received at all), the receiver sends a feedback message with value v.*

With this mechanism the sender will always obtain feedback from the receiver with the largest response value within one feedback round.

Let us now analyze the algorithm in detail: Following Equation (6.7) we use n for the actual number of potential responders and denote the expected number of feedback messages in Algorithm 3 with $R(n) := E[M]$. Let p_i be the fraction of responders with value v_i. For $k = 1$ the problem reduces to Algorithm 3 and we expect $R(n)$ feedback messages. For $k = 2$ we can reduce the problem to the previous case by assuming that every v_1 responder responds with both a v_1 and a v_2 message. Hereby, we can treat both groups independently from each other while preserving the fact that v_1 responders also stop further (unnecessary) responses from v_2 responders. Summing up both expected values we have $R(p_1 n) + R(n)$ messages. However, p_1 of the v_2 messages were sent by v_1 responders and are thus duplicates. Subtracting these duplicates we obtain $R(p_1 n) + p_2 R(n)$ for the expected number of responses.

This argument can be extended to the general case

$$\begin{aligned} E[M] &= R(p_1 n) + \frac{p_2}{p_1 + p_2} R(p_1 n + p_2 n) \\ &\quad + \frac{p_3}{p_1 + p_2 + p_3} R(p_1 n + p_2 n + p_3 n) \\ &\quad + \cdots + p_k R(n) \\ &= \sum_{i=1}^{k} \frac{p_i}{P_i} R(P_i n) \qquad (6.17) \end{aligned}$$

where $P_i := p_1 + p_2 + ... + p_i$ and thus $P_k = 1$. According to Section 6.3.3, $R(n)$ remains approximately constant over wide ranges of n. Assuming $R(n) \simeq R$, $p_i \simeq \frac{1}{k}$, and $k \gg 1$ we have

$$\begin{aligned} E[M] &\simeq \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}\right) R \\ &\simeq (\ln k + C) R \qquad (6.18) \end{aligned}$$

where $C = 0.577\ldots$ denotes the Euler constant.

From this analysis we see that the number of possible feedback values has an impact on the expected number of feedback messages. For a responder set with a real-valued feedback parameter, we can expect n different feedback values and therefore $E[M] \simeq \ln(n)R$.

6.8.2 Class-Based Extremum Feedback

Although this logarithmic increase is well acceptable for a number of applications, the algorithm's properties can be further improved by the introduction of feedback classes. Within those classes no differentiation is made between different feedback values. It is not necessary to choose a fixed size for all classes. The class size can be adapted to the required granularity for certain value ranges. In case a fixed number of classes is used, the expected number of feedback messages increases only by a constant factor over normal exponential feedback. As expected, this increase is also observed in the simulation results shown in Figure 6.11. As the number of classes approaches the number of receivers, the increase in feedback messages follows more and more the logarithmic increase for real-valued feedback as stated in Equation 6.18. For all simulations in this section we use the parameters $N = 100,000$ and $T = 4\tau$ and average the results over 200 simulation runs, unless stated otherwise.

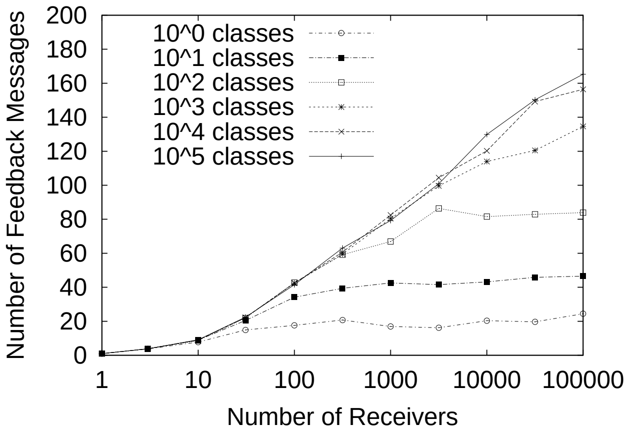


Figure 6.11: Uniformly sized classes

By adjusting the classes' positions depending on the actual value distribution, the number of classes required to cover the range of possible feedback values can be reduced without increasing the intervals' actual size. Thereby, the granularity of the feedback suppression (i.e., to what extent less optimal values can suppress better values) remains unchanged while the number of feedback messages is reduced.

Figure 6.12 gives a schematic overview of this mechanism. The first diagram shows the classless version of the feedback algorithm. Here, each time a feedback message v_i is sent, the range of suppressed value increases to $[0; v_i]$. A total of four feedback messages is sent in this example. The second diagram shows the same distribution of feedback for the case of static classes. We assume equally sized classes of size δ and $v_1 \in [0; \delta]$ for this example. After receipt of the first feedback message v_1, the entire range $[0; \delta]$ of the lowest feedback class is suppressed. Only when a value outside this class is to be reported another message is sent, resulting in three feedback messages in total. The third diagram shows the case of dynamically adjusted classes. Upon receipt of the first feedback message v_1 the suppression limit is immediately raised to $v_1 + \delta$ and thus the value range $[0; v_1 + \delta]$ is now being suppressed. Through this mechanism feedback is reduced to only two messages.

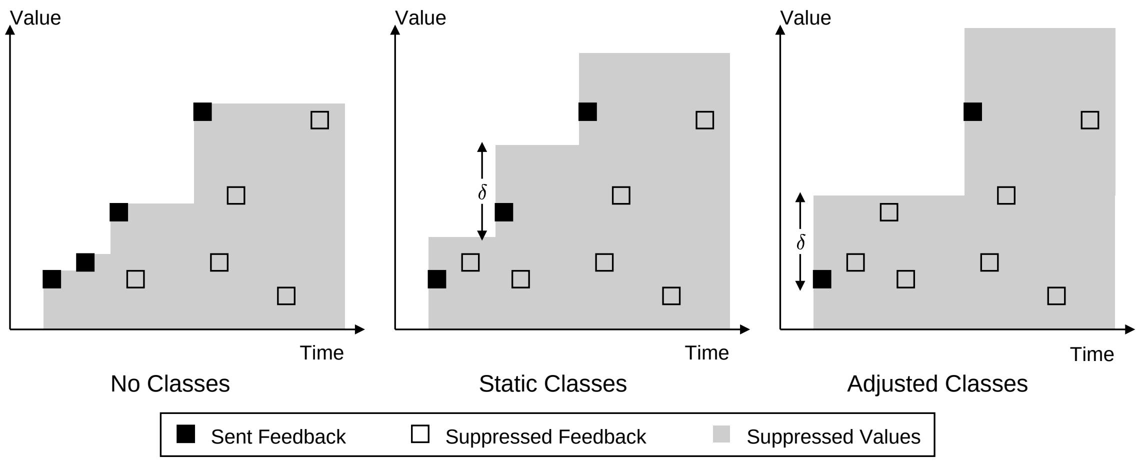


Figure 6.12: Class-based suppression with variable class position

With the above considerations, an elegant way to introduce feedback classes is the modification of Algorithm 4 to suppress feedback not only upon receipt of values strictly larger than the own value v but also upon receiving values $v' \geq (1 - q)v$. This results in an adaptive feedback granularity dependent on the absolute value of the optimum.

Algorithm 5 *(Adaptive Class-Based Extremum Detection):*
Let q be a tolerance factor with $q \in [0; 1]$. Modify Algorithm 4 such that a responder with value v cancels its timer if another responder has already sent feedback for value v' with $v' \geq (1-q)v$.

For $q = 0$ the algorithm is equivalent to Algorithm 4 where only feedback messages with smaller value are suppressed, and for $q = 1$ we obtain Algorithm 3 where all feedback after the first response is suppressed.

Assuming the values v_i to be evenly distributed between rv_{max} and v_{max} $(0 < r < 1)$ we have approximately k feedback classes[4], where $k < \frac{\ln r}{\ln 1-q}$. For a value range $0 < v_i < 1$ we can assume $k < \frac{-\ln n}{\ln 1-q}$, setting r inversely proportional to the number of receivers since the receiver set is too small to cover the whole range of possible values.

Approximating further with

$$p_i = \frac{(1-q)^{i-1} - (1-q)^i}{1-r} = q\frac{(1-q)^{i-1}}{1-r}$$

and

$$P_i = \frac{1-(1-q)^i}{1-r}$$

we have

$$E_{max}[M] \quad < \quad qR\sum_{i=1}^{k} \frac{(1-q)^{i-1}}{1-(1-q)^i} \tag{6.19}$$

The mechanism strongly benefits from the feedback classes being wider near the maximum and so holding more values than the classes near the minimum. As a consequence, the expected number of feedback messages is much lower compared to that of the previous algorithm. Note that for small r, the number of members with $v < (1-q)^{-1}r$ can be very small. Eventually, these feedback classes will contain only a single member, and we therefore loose the desired suppression effect that leads to a sub-logarithmic increase of feedback messages. In maximum search this effect cannot be observed since already a single response in the larger feedback classes near the maximum will suppress all feedback from the potentially large number of small classes. In fact, this characteristic is not specific to maximum and minimum search but rather depends on the classes being large or small near the optimum.

To demonstrate the effect we will calculate the expected number of feedback messages for a minimum search scenario: The feedback values v_i are again evenly distributed between rv_{max} and v_{max}, but in contrast to Algorithm 5 a responder cancels its timer if a response with $v' \leq (1-q)v$ is received. The algorithm produces the minimal value of the group within a factor of q.[5]

[4]We assume the parameter range $(r, 1)$ to be fully covered by the feedback classes which is not strictly the case for this algorithm. This approximation thus overestimates the expected number of feedback messages.

[5]As far as the expected number of feedback messages is concerned, this mechanism is equivalent to a maximum search with small class sizes for classes close to the optimum.

The feedback classes are in the opposite order as compared to our previous calculation.

$$p_i = \frac{(1-q)^{k-i} - (1-q)^{k-i+1}}{1-r} = q\frac{(1-q)^{k-i}}{1-r}$$

and

$$P_i = \frac{(1-q)^{k-i} - (1-q)^{k}}{1-r}$$

Thus

$$\begin{aligned} E_{min}[M] &< qR\sum_{i=1}^{k} \frac{1}{1-(1-q)^i} \\ &\approx (1-q)\,E_{max}[M] + kqR \end{aligned} \tag{6.20}$$

Hence, for small r (large k) the sum is significantly larger than in the previous case.

Both scenarios have been simulated with various values for q. The sub-logarithmic increase of feedback messages can be seen in the plots shown in Figure 6.13. But only with maximum search where the feedback classes near the search goal are wider, the strong class-induced suppression dominates the $\ln(n)$ scale-effect.

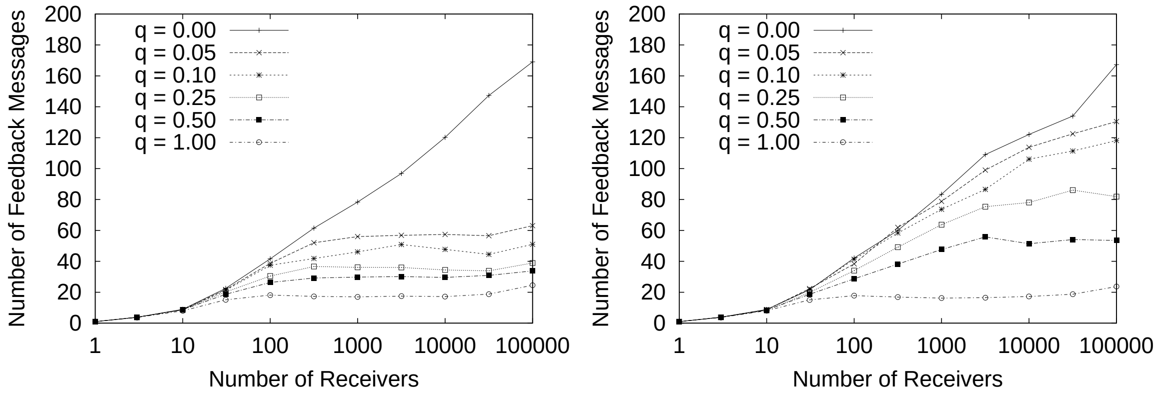


Figure 6.13: Number of feedback messages for maximum search (left) and minimum search (right) with adaptive class sizes

Numeric values for the upper limits on the expected number of feedback messages in both scenarios can be obtained from Equations (6.19) and (6.20). Some example values are shown in Table 6.1. These limits match well with the results of our simulations.

As mentioned before, Algorithm 5 guarantees a maximum deviation from the true optimum of a factor of q. It is worthwhile to note that this factor really is an upper bound on the deviation.

Table 6.1: Upper limits for the expected number of feedback messages [6]

q	0.05	0.10	0.25	0.50	1.00
Maximum search	3.64	3.00	2.19	1.59	1.00
Minimum search	7.91	7.00	5.64	3.80	1.00

Almost always the reported values will be much closer to optimal since the sender can choose the best one of all the responses given. The deviation of the best reported value from the optimum for different tolerance factors q is depicted in Figure 6.14. On average, with normal exponential suppression (i.e., $q = 100\%$) the best reported value lies within 10% of the optimum, for $q = 50\%$ the deviation drops to less than 0.15%, for $q = 10\%$ we obtain less than 0.02% deviation, etc. Thus, even for relatively high q with consequently only a moderate increase in the number of feedback messages, the best feedback values have only a marginal deviation from the optimum.

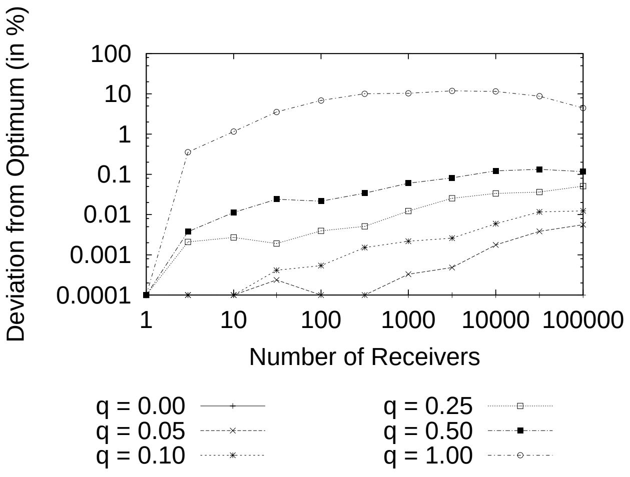


Figure 6.14: Feedback quality with different tolerance values

6.8.3 Biased Feedback

The previously described algorithms yield good results for various cases of extremum detection. However, they will not affect the expected value of the first feedback message but only improve the expected values of subsequent messages. In certain cases, the algorithms can be further improved by biasing the feedback timers. Increasing the probability that $t_1 < t_2$ if $v_1 > v_2$ results in better feedback behavior[7] but we must carefully avoid a feedback implosion for cases

[6]The numbers shown are factors of R and calculated with $r = 10^{-2}$

[7]Note that we are now back to maximum detection.

where many large values are present in the responder group. Without loss of generality we assume $v \in [0, 1]$ for the remainder of this section.

If the probability distribution of the values v is known, the number of responses can be minimized using the following algorithm:

Algorithm 6 *(Deterministic Feedback Suppression):*
Let $P(v) = P(v' < v)$ be the probability distribution function of the values v within the group of responders. We follow Algorithm 3, but instead of drawing a random number we set the feedback time directly to

$$t = T \max(0; 1 + \log_N(1 - P(v)))$$

Clearly, duplicate feedback responses are now only due to network latency effects since the responder with the maximum feedback value is guaranteed to have the earliest response time. However, the feedback latency is strongly coupled to the actual set of feedback values. Moreover, if the probability distribution of this specific set does not match the distribution used in the algorithm's calculation, feedback implosion is inevitable. For this reason, Algorithm 6 should only be used if the distribution of feedback values is well known for each individual set of values.

The latter condition is crucial. In general, it does not hold for values from causally connected responders. Consider for example the loss rate for multicast receivers: If congestion occurs near the sending site, all receivers will experience a high packet loss rate simultaneously. Since the time-average distribution does not show this coherence effect the algorithm presented above will produce feedback implosion if used to solicit responses from high-loss receivers. Due to this effect, the application of this simple mechanism is quite limited. It can be used, for example, with application level values where no coherence is generated within the network. An online auction could be an example.

A simple way to adopt the key idea of value-based feedback bias is to mix value-based response times with a random component. This mechanism can be applied in various cases where coherence effects prohibit the application of Algorithm 6. Let us study an example:

Algorithm 7 *(Feedback with Combined Bias):*
Apply Algorithm 3 but modify the feedback time to

$$\begin{aligned} t &= T \max(0; (1 - v) + v(1 + \log_N x)) \\ &= T \max(0; (1 + \log_N x^v)) \end{aligned} \tag{6.21}$$

Here, the feedback time consists of a component linearly dependent on the feedback value and a component for the exponential feedback suppression. The feedback time t is increased in proportion to decreasing feedback values v, and a smaller fraction of T is used for the actual suppression. As long as at least one responder has a sufficiently early feedback time to suppress the majority of other feedback this distribution of timer values greatly decreases the number of duplicate responses while at the same time increasing the quality of the feedback (i.e., the best reported value with respect to the actual optimum value of the receiver set). Furthermore, in contrast to pure extremum detection algorithms, this mechanism improves the expected value of the value reported in the first response as well as subsequent responses.

However, the feedback suppression characteristics of the above mechanism still depend at least to some extent on the value distribution at the receivers. Some extreme cases such as $v = 0$ for all receivers will always result in a feedback implosion. A more conservative approach is to not combine bias and suppression but use a purely additive bias.

Algorithm 8 *(Feedback with Additive Bias):*
Apply Algorithm 3 but modify the feedback time to

$$t = T \max\left(0; \gamma(1-v) + (1-\gamma)(1+\log_N x)\right) \tag{6.22}$$

with $\gamma \in [0;1]$.

To retain the same upper bound on the maximum feedback delay, it is necessary to split up T and use a fraction of T to spread out the feedback with respect to the response values and the other fraction for the exponential timer component. As long as $(1-\gamma)T$ is sufficiently large compared to the network latency τ, an implosion as in the above example is no longer possible.

6.8.4 Feedback Bias with Different Distributions of Response Values

To better demonstrate the characteristics of these modifications, Figures 6.15 to 6.17 show how the feedback time changes with respect to response values compared to normal unbiased feedback according to Algorithm 3. A single set of random variables was used for all the simulations to allow a direct comparison of the results. For the simulations, the parameters N and n were set to $10,000$ and $2,000$ respectively.[8] In these simulations we do *not* consider maximum search

[8]Note that using $n = N = 10,000$ instead of $n = 2,000$ would *reduce* the probability of an implosion since the probability that one early responder suppresses all others increases.

but only how feedback biasing affects the distribution of feedback timers. Thus, to isolate the effect of feedback biasing, only a single feedback class was used such that the first cancellation notification suppresses *all* subsequent feedback. All simulations were carried out with $T = 4\tau$ (right-hand graphs) as well as $T = 8\tau$ (left-hand graphs) to demonstrate the impact of the feedback delay on the number of feedback responses. Each graph shows the feedback times in τ of the receiver set along the x-axis and the corresponding response values on the y-axis for each of the three feedback mechanisms *no bias* (Algorithm 3), *combined bias* (Algorithm 7), and *additive bias* (Algorithm 8) with $\gamma = 1/4$. Suppressed feedback messages are marked with a dot, feedback that is sent is marked with a cross, and the black square indicates which of these feedback messages had a value closest to the actual optimum of the receiver set.

In the graphs in Figure 6.15, the response values of the receivers are uniformly distributed. When no feedback bias is used, the first response that suppresses the other responses is random in value. In contrast, both feedback biasing methods result in the best reported feedback value being very close to the actual optimum. The number of sent feedback messages is higher with the two biasing methods since a smaller fraction of T is used for feedback suppression. Naturally, the number of feedback messages also increases when T is smaller, as depicted in the right graph.

In Figure 6.16, the same simulations were carried out for an exponential distribution of response values with a high probability of being close to the optimum. (When a reversed exponential distribution with most values far from the optimum is used, the few good values suppress all other feedback, and again a feedback implosion is always prevented.) As can be seen from the graph, feedback suppression works well even when the actual distribution of response values is no longer uniform. For a uniform as well as an exponential distribution of response values, the *combined bias* suppression method results in fewer feedback messages while maintaining the same feedback quality.

However, as mentioned before, combining bias and suppression permits a feedback implosion when the range of feedback values is smaller than anticipated. In this case, the bias results in an unnecessary delaying of feedback messages, thus reducing the time that can be used for feedback suppression. In Figure 6.17, the response values are distributed uniformly in $[0; 0.25]$ instead of $[0; 1]$. For $T = 4\tau$, the time left for feedback suppression is τ, resulting in a scenario where no suppression is possible and each receiver will send feedback. Even when $T = 8\tau$ and thus a time of 2τ can be used for the feedback suppression, the number of feedback messages is considerably larger than in simulations with an additive bias. The exact numbers for the feedback responses of the three methods are given in Table 6.2.

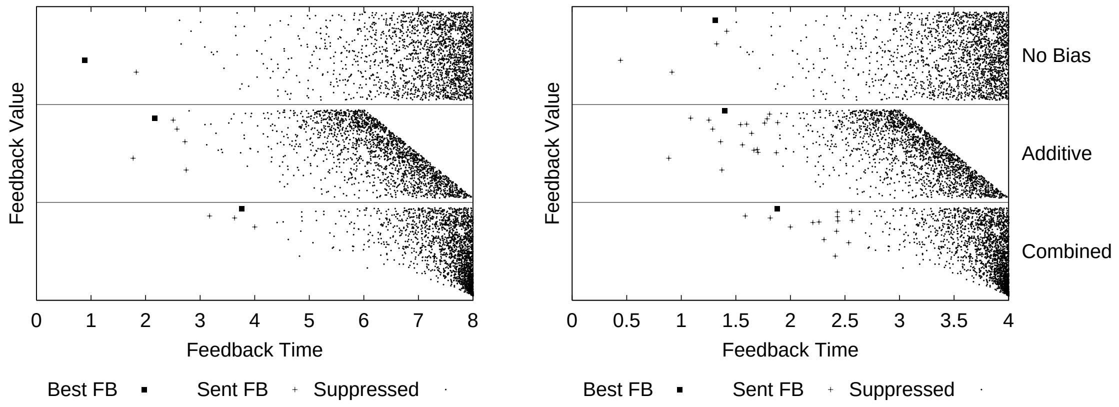


Figure 6.15: Feedback time and value (uniform distribution of values)

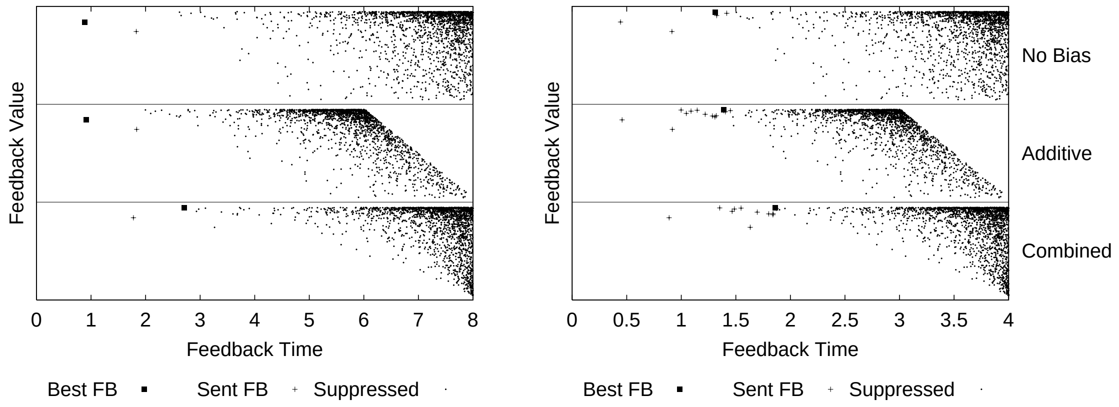


Figure 6.16: Feedback time and value (exponential distribution of values)

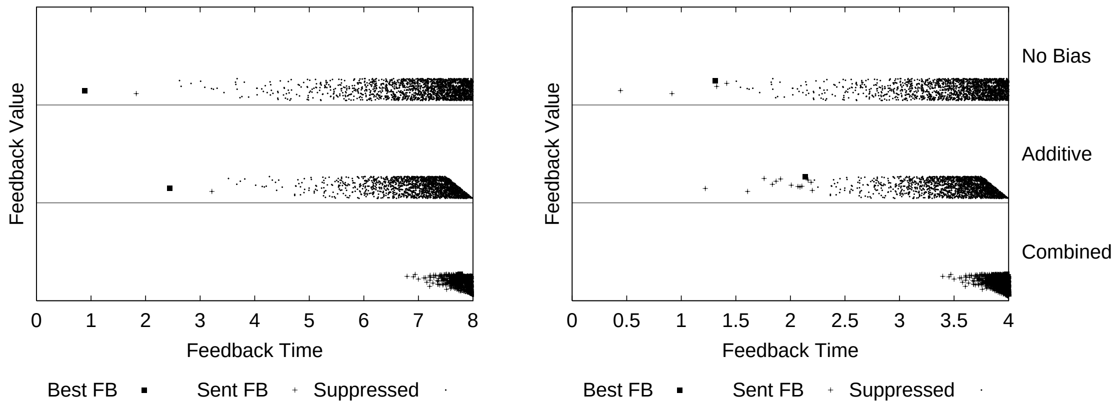


Figure 6.17: Feedback time and value (truncated uniform distribution of values)

Table 6.2: Number of responses with the different biasing methods

Feedback Time	No Bias	Additive	Combined
T=4, Uniform	5	19	15
T=4, Exponential	5	14	12
T=4, Truncated	5	14	2000
T=8, Uniform	2	6	4
T=8, Exponential	2	2	2
T=8, Truncated	2	2	334

For suppression to be effective, the amount of time reserved for the exponential distribution of the feedback timers should not be smaller than 2τ. Thus, the feedback implosion with Algorithm 7 can be prevented by bounding v such that $vT > 2\tau$ (i.e., using $v' = \max(v; 2\tau/T)$ instead of v in Equation 6.21). In the worst case, the distribution of the feedback timers is then similar to an unbiased exponential distribution with $T = 2\tau$. A higher upper bound can be used to reduce the expected number of feedback messages in the worst case. The same considerations hold for the choice of the value of γ for the additive feedback bias.

The outcome of a single experiment is not very representative since the number of feedback messages is extremely dependent on the feedback values of the early responders. As for the previously discussed feedback mechanisms, we depict the number of feedback messages for combined and additive bias with $T = 4\tau$, averaged over 200 simulations, in Figure 6.18.

Combined bias works well for a uniform or an exponential distribution of feedback values, and the number of responses is much closer to that of plain exponential feedback suppression without bias than with an additive bias. Nevertheless, we clearly see that it fails to prevent a feedback implosion in case of a truncated uniform distribution, while additive bias is only a small constant factor worse than feedback suppression without any bias.

The main advantage of the feedback bias is that the expected response value for early responses is improved. This not only reduces the time until close-to-optimal feedback is received (with unbiased feedback and class-based suppression, close to optimal feedback is likely to arrive at the end of a feedback round) but also reduces the number of responses with less optimal feedback.

Figure 6.19a shows how the feedback quality improves compared to normal exponential feedback suppression when biasing the feedback timer. Between 100 and 10000 receivers, the deviation of the best response from the optimum is reduced from about 14% to 6% for additive bias and to less than 2% for combined bias.

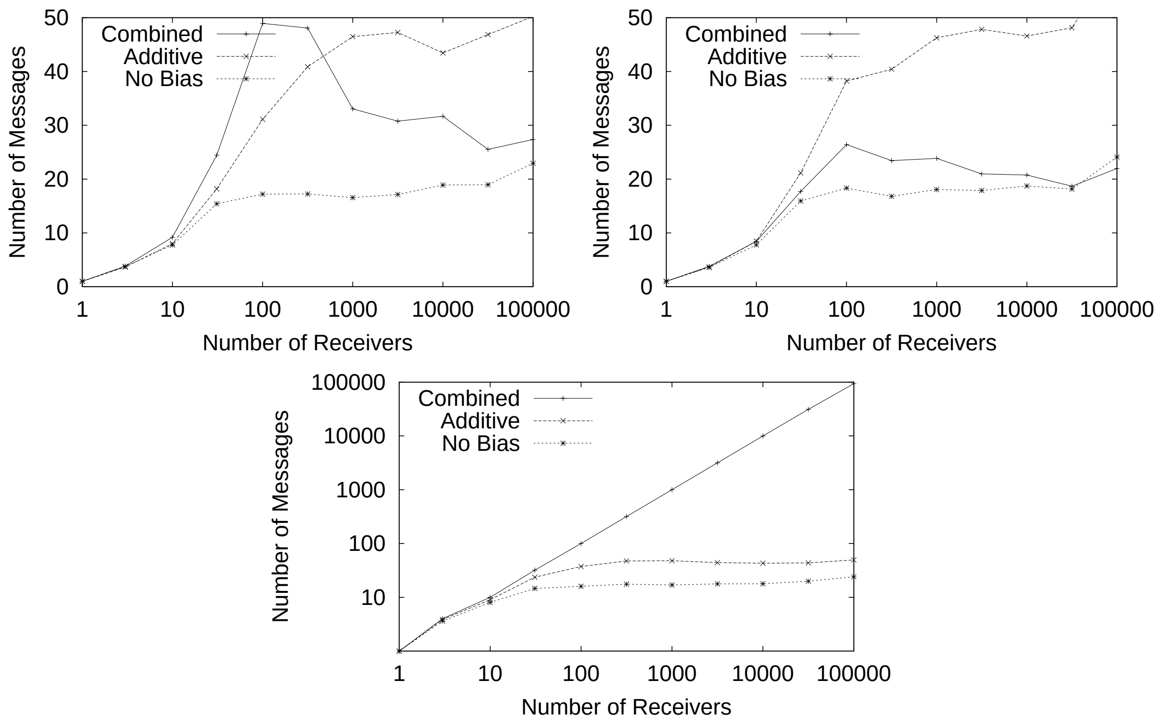


Figure 6.18: Number of responses with feedback biasing (uniform, exponential, and truncated uniform distribution of feedback values)

While a similar increase in feedback quality can be achieved by using feedback classes (at the expense of an increased number of feedback messages), only with a feedback bias is it possible to improve the quality of the *first* feedback message. In case a close to optimal value is needed very quickly, using either Algorithm 7 or Algorithm 8 can be beneficial. Figure 6.19b depicts the average deviation of the value of the first feedback message from the optimum. Here, the increase in quality is much more obvious than in the previous case. With all unbiased feedback mechanisms, the first reported value is random, and thus the average deviation is 50% (for large enough n) whereas the combined and the additive biased feedback mechanisms achieve average deviation values around 10% and 30% respectively.

Lastly, the expected delay until the first feedback message is received is of concern. While all mechanisms adhere to the upper bound of T, feedback can be expected earlier in most cases. In Figure 6.20 we show the average feedback delay for biased and unbiased feedback mechanisms. For all algorithms the feedback delay decreases logarithmically for an increasing number of receivers. The exact run of the feedback curve depends on the amount of time used for suppression. For this reason, unbiased feedback delay drops faster than biased feedback, since a bias can only delay feedback messages compared to unbias feedback. In case the number of receivers

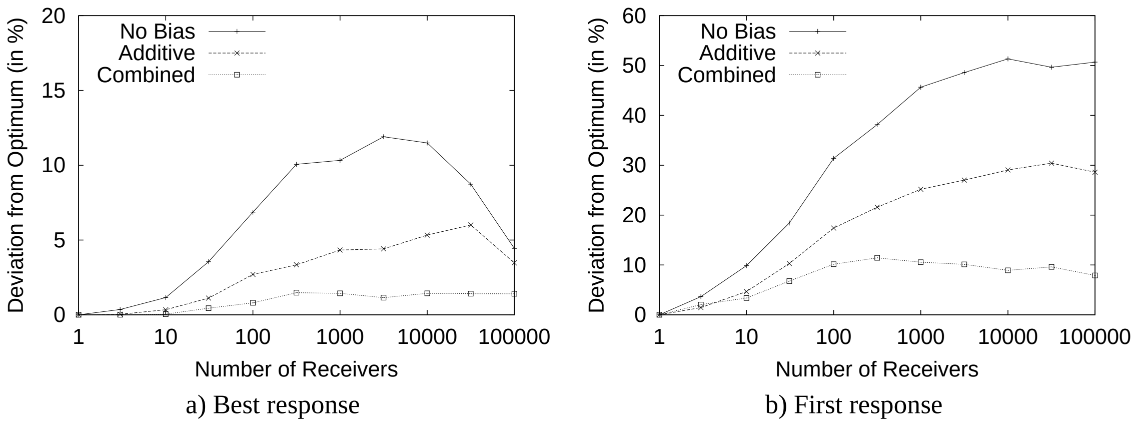


Figure 6.19: Feedback quality (deviation from optimum)

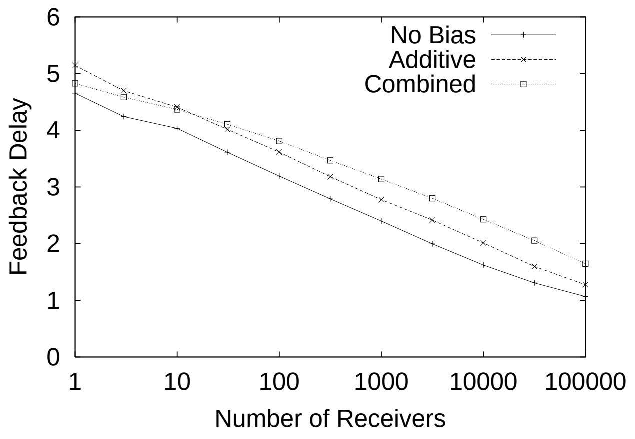


Figure 6.20: Average feedback delay

is estimated correctly (i.e., $n = N$), the feedback delay for unbiased feedback drops to τ, the minimum delay possible for such a feedback system. Biased feedback delay is slightly higher with approximately 1.5τ.

6.8.5 Combining Feedback Bias and Class-Based Suppression

When feedback bias and class-based suppression are combined, we obtain the same bound on the deviation of the best response from the real optimum value as with class-based suppression and the same improvement for the expected value of the first response as with the feedback biasing methods.

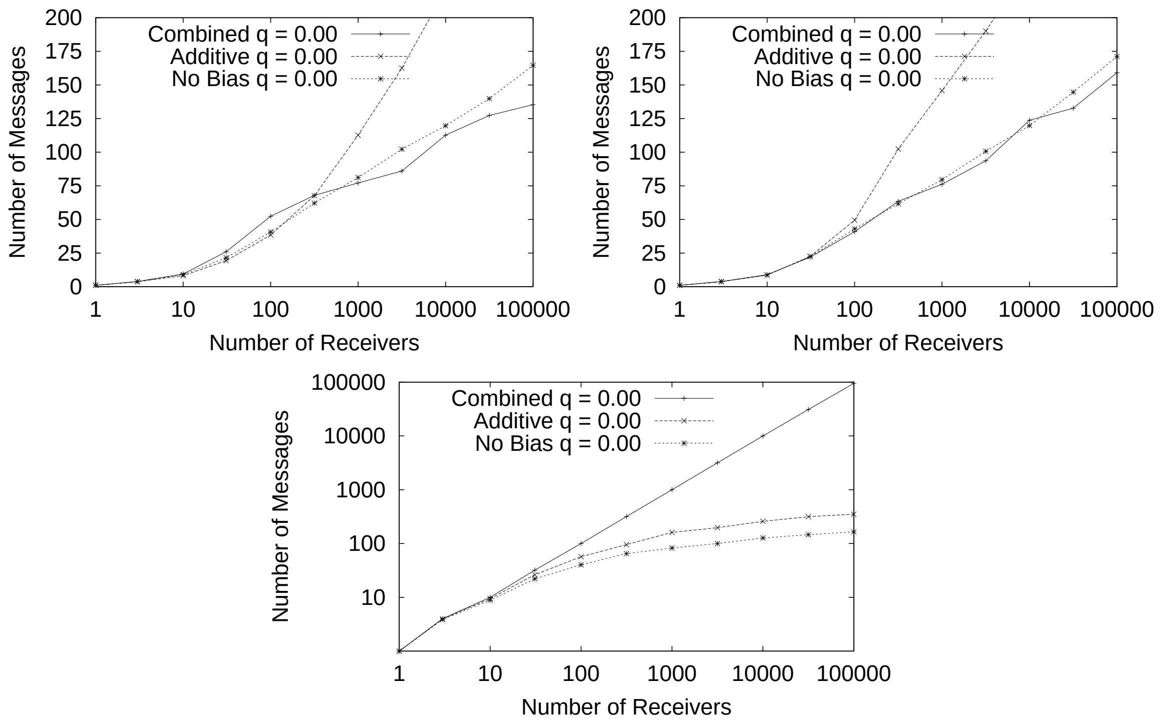


Figure 6.21: Feedback biasing and class-based extremum detection with $q = 0.0$ (uniform, exponential, and truncated uniform distribution of feedback values)

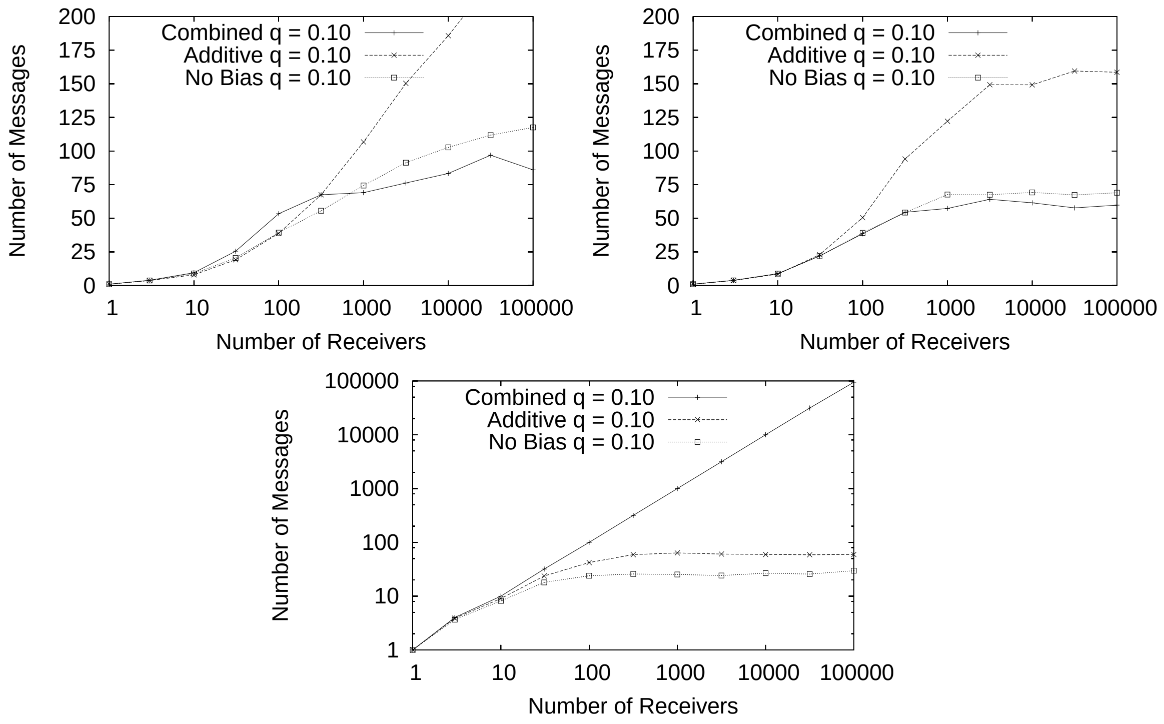


Figure 6.22: Feedback biasing and class-based extremum detection with $q = 0.1$ (uniform, exponential, and truncated uniform distribution of feedback values)

To see which impact this combined mechanism has on the number of responses, we show simulation results of the mechanism for $q = 0.0$ and $q = 0.1$ in Figures 6.21 and 6.22 respectively. For the class-based suppression, we choose minimum search with small class sizes close the the optimal value, as this scenario is much more challenging than maximum search. We use the same distributions (uniform, exponential, and truncated uniform) as in the previous simulations. From the graphs we can see that the difference in the number of responses between combined bias and no bias is small (except for the truncated uniform distribution, where the combined bias allows a feedback implosion).

In contrast, the smaller time interval allocated for suppression with the additive bias results in a higher number of feedback messages. Generally, with additive bias the response times will vary less than without bias, which results in a higher number of feedback timers being within the short time interval after the first response, where no suppression is possible. On average, the response that would be the first one if no bias were used will be delayed a bit, and its timeout value will be closer to the large bulk of later responses. Therefore, additive bias does improve the expected feedback value of the first response, but this improvement will not reduce the total number of responses seen in combination with class-based suppression.

In this chapter, we have analyzed several feedback algorithms and have identified exponential feedback suppression as being the most suitable algorithm for our purpose. Together with the modifications to allow the detection of extremal values within the group of responders, we can now use these mechanisms for scalable, value-based receiver feedback upon which the congestion control mechanism presented in the next chapter will be based.

Chapter 7

TCP-Friendly Multicast Congestion Control

7.1 Introduction

It is widely accepted that one of several factors inhibiting the usage of IP multicast is the lack of good, deployable, well-tested multicast congestion control mechanisms. To quote [MRBP98]:

> *The success of the Internet relies on the fact that best-effort traffic responds to congestion on a link by reducing the load presented to the network. Congestion collapse in today's Internet is prevented only by the congestion control mechanisms in TCP.*

We believe that for multicast to be successful, it is crucial that multicast congestion control mechanisms be deployed that can co-exist with TCP in the FIFO queues of the current Internet. However, the design of good multicast congestion control protocols is by far more difficult than the design of unicast protocols. Multicast congestion control schemes ideally should scale to large receiver sets and be able to cope with heterogeneous network conditions at the receivers.

This chapter describes TCP-Friendly Multicast Congestion Control (TFMCC), which belongs to the class of single-rate congestion control schemes. Such schemes inevitably do not scale as well as layered schemes. However, they are much simpler, match the requirements of some applications well, and we will demonstrate that they can scale to applications with many thousands of receivers. These schemes also suffer from degradation in the face of highly congested links to a few receivers – how to deal with such situations is a policy decision, but we expect that most applications using a single-rate scheme will have application-specific thresholds below which a receiver is compelled to leave the multicast group.

TFMCC [WH01a, WH01b] extends the basic mechanisms of TFRC described in Chapter 3 into the multicast domain. The primary differences in the design of TFRC and TFMCC are that in the multicast case the receivers measure their RTT to the sender and perform the calculation of the acceptable rate. This rate is then fed back to the sender, the challenge being to do this in a manner which ensures that feedback from the receiver with the lowest calculated rate reaches the sender whilst avoiding feedback implosions. Moreover, we need to make sure than any additional delay imposed to avoid feedback implosion does not adversely affect the fairness towards competing protocols.

7.2 Related Work

In this section we will briefly discuss earlier work on multicast congestion control protocols. For a more detailed overview and a protocol evaluation we refer the interested reader to [WDM01].

is a hybrid protocol that combines aspects of window-based and rate-based congestion control. TEAR receivers calculate a fair receive rate which is sent back to the sender, who then adjusts the sending rate. To this end, the receivers maintain a congestion window that is modified similarly to TCP's congestion window. Since TCP's congestion window is located at the sender, a TEAR receiver has to try to determine from the arriving packets when TCP would increase or decrease the congestion window size. Additive increase and window reductions caused by triple duplicate ACKs are easy to emulate. However, due to the lack of acknowledgements, timeout events that would lead to a new slow-start in TCP can be estimated only roughly.

In contrast to TCP, the TEAR protocol does not directly use the congestion window to determine the amount of data to send but calculates the corresponding TCP sending rate. This rate is roughly a congestion window worth of data per round trip time. To avoid TCP's sawtooth-like rate shape, TEAR averages the rate over an *epoch*, which is defined as the time between consecutive rate reduction events. To prevent further unnecessary rate changes caused by noise in the loss patterns, a smooth rate is determined by using a weighted average over a certain number of epochs for the final rate. This value is then reported to the sender, which adjusts the sending rate accordingly. Since the rate is determined at the receivers and TEAR refrains from acknowledging packets, it can be used for multicast as well as for unicast communication. So far, only a unicast version of TEAR exists, but the mechanism can be made multicast-capable by implementing an appropriate feedback suppression scheme to communicate the calculated rate to the sender as well as scalable RTT measurements. Due to the close modeling of TCP's short-term behavior, TEAR shows TCP-friendly behavior while avoiding TCP's frequent rate changes.

There are two main problems that have to be solved in order to use window-based congestion control for multicast. First, care has to be taken as to how the sending rate is decreased in case of network congestion. In large multicast sessions receivers may experience uncorrelated loss, and it is therefore likely that most of the transmitted packets are lost on the path to at least one receiver. If the sender responded to each of these losses by decreasing the congestion window, the transmission would likely stall after some time. This problem is known as the *loss path multiplicity problem* [BTK99]. Whenever rate adjustment decisions are based not on congestion information from a specific receiver but on the overall congestion information present in the whole distribution tree, protocol performance can suffer considerably if the protocol has not been designed correctly. The second problem is how to free slots in the congestion window. Clearly it is not possible for the sender to receive acknowledgments for each packet from each receiver, as this would cause an acknowledgment implosion.

The Random Listening Algorithm (RLA) proposed by Wang and Schwartz [WS98] extends TCP SACK by introducing some enhancements for multicast. For each receiver, the multicast sender stores the smoothed round-trip time and the measured congestion probability. A loss is detected by the sender via identification of discontinuous acknowledgements or via timeout. Based on these loss indications, the number of receivers n with a high congestion probability is tracked. If congestion is detected, the window is halved in the following two cases: (1) if the previous window cut was made too long ago (the authors propose an interval of twice the moving average of the window size times the smoothed round-trip time of the corresponding receiver) or (2) if a generated uniform random number π is less than or equal to $1/n$. When a packet has been acknowledged by all receivers, the congestion window $cwnd$ is incremented by $1/cwnd$, identical to TCP. A TCP-like retransmission scheme with fast-recovery is also included in RLA. With the above mechanisms, RLA avoids the loss path multiplicity problem, while achieving statistical long-term fairness. In [WS98] it is demonstrated that RLA is fair to TCP according to the definition of bounded fairness (see Section 2.3).

Multicast TCP (MTCP) [RBR99] is a reliable multicast protocol that uses window-based congestion control to achieve TCP-friendliness. MTCP groups the session participants into a logical tree structure where the root of the tree is the sender of the data. A parent in the logical tree structure stores a received packet until receipt is acknowledged by all of its children. Upon receiving a packet, a child (which may be a parent for other participants) transmits an acknowledgment to its parent using unicast. To control congestion, MTCP requires that each parent maintain two values: a congestion window and a transit window. The size of the congestion window is managed similarly to that of TCP, including slow-start and congestion avoidance. The main differences to TCP are (1) that the congestion window is only incremented when ACKs from all children have been received and (2) that a packet is immediately (re)transmitted to a child if it indicates via a NACK that it has not yet received the packet. The size of the congestion window is halved when any child reports three consecutive NACKs, or set to one when a timeout occurs because a child has not acknowledged a packet at all. The transit window keeps track of the amount of data that the children of a parent node have not yet acknowledged. In MTCP, the loss path multiplicity problem is avoided by means of aggregation at the intermediate nodes. Each node forwards the information about the bottleneck link of its children to its parent. Therefore the sender will receive information about the overall bottleneck link rather than about uncorrelated packet loss. The main drawback of MTCP is its complexity and the required setup of a tree structure where each node has to perform package storage, repair, and congestion monitoring functionality.

In pragmatic general multicast congestion control (pgmcc) [Riz00] the bottleneck receiver with the worst network connection is selected as a group representative. Once such a receiver, called the *acker*, is selected, a TCP-style window-based congestion control algorithm is run between the sender and the acker. Minor modifications compared to TCP concern the separation of congestion control and reliability to be able to use PGMCC for reliable as well as unreliable data transport and the handling of out of order packets and RTT changes when a different receiver is selected as the acker.

The most challenging aspect of pgmcc is how to select the group representative, since the selection process for the acker mainly determines the fairness of the protocol. It is based on the simple version of the TCP throughput model in Equation (2.1). Each receiver tracks the RTT and the smoothed loss rate and feeds these values into the model. The results are communicated to the sender using randomized feedback timers to avoid an implosion. If available, PGMCC also makes use of network elements to aggregate feedback. As evidenced by the simulations in [Riz00], PGMCC competes fairly with TCP for many different network conditions. The basic congestion control mechanism is simple, and its dynamics are well understood from the analysis of TCP congestion control. This close mimicking of TCP's window behavior produces rate variations that resemble TCP's sawtooth-like rate.

As discussed in the introduction, it is also possible to design multi-rate congestion control mechanisms where the receivers individually adapt the receive rate according to the congestion on their path to the sender. The data is distributed over multiple multicast session, and receivers join the appropriate number of sessions to adjust their overall receive rate with respect to the current network conditions.

One of the first working examples of layered multicast transmission in the Internet was Receiver-driven Layered Multicast (RLM) for the transmission of video, developed by McCanne, Jacobson and Vetterli [MJV96]. Their work did not focus on TCP-friendliness but on how to provide each receiver with the best possible video quality with respect to the bandwidth available on the path between the sender and that receiver. In RLM, the sender splits the video into several layers. A receiver starts receiving by subscribing to the first layer. When the receiver does not experience congestion in the form of packet loss for a certain period of time, it subscribes to the next layer. This is called a join experiment. When a receiver experiences packet loss, it unsubscribes from the highest layer it is currently receiving.

The use of RLM to control congestion is problematic since RLM's mechanism of adding or dropping a single layer based on the detection of packet loss is not TCP-friendly and can also result in a very unfair distribution of bandwidth among concurrent RLM sessions. Furthermore, leaving a multicast group may take a significant amount of time, usually on the order of several seconds. Failed join experiments (i.e., a receiver who has just joined a layer immediately having to leave it again because the necessary bandwidth is not available) are therefore very costly in terms of the additional congestion they may cause. For layered schemes to be efficient, it is imperative that receivers behind the same bottleneck synchronize their join and leave decisions. Several protocols have been developed that improve the original concept of RLM.

Vicisano, Crowcroft and Rizzo address most of these problems in their work on Receiver-driven Layered Congestion Control (RLC) [VCR98]. They propose to dimension the layers so that the bandwidth consumed by each new layer increases exponentially. If layer n carries data at the rate L_n, then layer $n+1$ has a rate of $L_{n+1} = \sum_{i=0}^{n} L_i$. The time that a receiver has to wait before being allowed to join a new layer also increases exponentially with each additional layer. On the other hand, a layer is dropped immediately when congestion becomes apparent in form of packet loss. This emulates the behavior of TCP since the increase in bandwidth is proportional to the amount of time required to pass without packet loss before being allowed to join the layer. At the same time the reaction to congestion is a multiplicative decrease, since dropping one layer results in halving the overall receive rate.

To improve synchronization between receivers, they may join a layer only at so-called synchronization points (SP). SPs in higher layers are exponentially less frequent than in lower layers. Thus, a receiver that is only subscribed to a small number of layers is likely to catch up with receivers with a higher subscription level. After some time, receivers that share the same bottleneck should be joining and leaving layers synchronously. In order to decrease the likelihood that a join experiment will fail, the RLC sender creates a short burst period before an SP. During this burst period the data rate is doubled in each layer. Only if a receiver does not experience any signs of congestion during the burst it is allowed to join the next higher layer.

Despite the improvements in the congestion control mechanism over RLM, RLC still has some drawbacks. The granularity at which the rate can be adapted to the network conditions is very coarse and may cause unfair behavior; the exponential distribution of the layers only allows to double or halve the receive rate. The second problem is that the transmitted data must support layering. While this is true for video and bulk-data transmission, streams that are more interactive like those produced by shared whiteboards, cannot easily be separated into multiple layers. RLC does not take the round-trip time into account when determining the sending rate. This can lead to unfairness towards TCP since TCP is biased against connections with a high round-trip time. Furthermore, it is not guaranteed that the artificial bursts of packets introduced by RLC be acceptable for a broad range of applications that support layered transmission. A general point of controversy that applies to all layered congestion control schemes is whether it is acceptable to "abuse" network mechanisms like multicast session setup and teardown to achieve transport layer functionality like congestion control.

Byers et. al. propose Fair Layered Increase/Decrease with Dynamic Layering (FLID-DL) [BFH$^+$00]. The protocol uses a Digital Fountain [BLMR98] at the source. With Digital Fountain encoding, the sender encodes the original data and redundancy information such that receivers can decode the original data once they have received a fixed number of arbitrary but distinct packets. Since it is not necessary to ensure delivery of specific packets, the layering scheme is much more flexible than previous proposals.

FLID-DL introduces the concept of Dynamic Layering to reduce the join and leave latencies associated with adding or dropping a layer. With Dynamic Layering, the bandwidth consumed by a layer decreases over time. Thus, a receiver has to periodically join additional layers to *maintain* its receive rate. The receive rate is reduced simply by not joining additional layers, whereas rate increase requires joining multiple layers. To reduce the total number of layers required by the mechanism, layers are reused after a quiet period during which no data is transmitted over the layers for a certain amount of time. This scheme provides an elegant solution to the potentially long leave latencies, provided that the quiet period is sufficient for normal leave operations to take effect. Dynamic Layering is complemented by a Fair Layered Increase/Decrease scheme that results in a receive rate that is fair to a TCP flow with a fixed round-trip time experiencing the same loss rate. FLID retains RLC's concepts of sender-initiated synchronization points to coordinate receivers but refrains from packet bursts to probe for available bandwidth. Increase signals are distributed over time such that the long-term response to packet loss is the same as the one given in Equation (2.3). The authors discuss a deterministic approach with increase signals at fixed points of time as well as a probabilistic approach, where increase signals for the different layers are given with certain probabilities.

The FLID-DL protocol is a considerable improvement over RLC. It does not suffer from long leave latencies and is more flexible with regard to the bandwidth distribution on the layers. However, like RLC, FLID-DL does not take into account the round-trip time and thus exhibits unfair behavior towards TCP under certain network conditions. Its use results in major overhead for the underlying multicast routing protocol as join and leave decisions may occur much more frequently than with other protocols. Furthermore, it requires the use of significantly more multicast groups than conventional layered multicast schemes.

7.3 The TFMCC Protocol

Building an equation-based multicast congestion control mechanism requires that the following problems be solved:

- Each receiver must measure the loss event rate. Thus a filter for the packet loss history needs to be chosen that is a good stable measure of the current network conditions, but is sufficiently responsive when conditions change.
- Each receiver must measure or estimate the RTT to the sender. Devising a way to do this without causing excessive network traffic is a key challenge.
- Each receiver uses the model for long-term TCP throughput given in Equation (2.3) to calculate an acceptable sending rate from the sender to itself. This rate needs to be reported back to the sender. A feedback scheme must be devised so that feedback from the receiver calculating the slowest transmission rate always reaches the sender, but feedback implosions do not occur when network conditions change.

- A filtering algorithm needs to be devised for the sender to determine which feedback it should take into account to adjust the transmission rate.

Clearly, all these parts are closely coupled. For example, altering the feedback suppression mechanisms will impact how the sender deals with this feedback. Many of our design choices are heavily influenced by TFRC, as these mechanisms are fairly well understood and tested. In this chapter we will expend most of our efforts focusing on those parts of TFMCC that differ from TFRC.

In the following sections we will elaborate on how the necessary parameters for the control equation are computed and how to deal with potentially large receiver sets. We do not go into the details of the measurement mechanism for the loss event rate since it is the same as in TFRC presented in Section 3.3.2.

7.3.1 Adjusting the Sending Rate

The sender will continuously receive feedback from the receivers. If a receiver sends feedback that indicates a rate that is lower than the sender's current rate, the sender will immediately reduce its rate to that given in the feedback message. However, this leaves us with a problem – how do we increase the transmission rate? We cannot afford to increase the transmission rate in the absence of feedback, as the feedback path from the slowest receiver may be congested or lossy. As a solution we introduce the concept of the *current limiting receiver* (CLR). The CLR is the receiver that the sender believes to currently have the lowest expected throughput of the group. In this respect, the CLR is comparable to the *representative* (or "acker") used in congestion control schemes such as PGMCC. The CLR is permitted to send immediate feedback without any form of suppression, so that the sender can use the CLR's feedback to increase the transmission rate.

The CLR will change if another receiver sends feedback indicating that a lower transmission rate is required. It will also change if the CLR leaves the multicast group – this is normally signaled by the CLR, but an additional timeout mechanism serves as a backup in case the CLR crashes or becomes unreachable.

The way loss measurement is performed limits the possible rate increase to roughly 0.3 packets per RTT, as discussed in Section 3.3.4. However, if the CLR leaves the group, the new CLR may have a significantly higher calculated rate. We cannot afford to increase directly to this rate, as the loss rate currently measured may not be a predictor of the loss rate at the new transmission rate. Instead we then impose a rate increase limit of one packet per RTT, which is the same as TCP's additive increase constant, so that the rate gradually increases to the new CLR's rate. In contrast, as mentioned above, a decrease of the sending rate is always done immediately to the rate indicated by the CLR.

In the absence of any feedback from any of the receivers it is also necessary to reduce the sending rate since the probable cause of missing feedback is congestion. For every 10 consecutive RTTs without feedback, the sending rate is cut in half. The rate is at most reduced to one packet every 64 seconds. Note that when receivers stop receiving data packets, they will stop sending feedback. This eventually causes the sending rate to be reduced in the case of network failure. If the network subsequently recovers, a linear increase to the calculated rate of the CLR will occur at one packet/RTT per RTT.

How the TFMCC sender schedules the sending of packets, given that the clock of an operating system only has a limited granularity, is described in Appendix B.1.

7.3.2 Round-trip Time Measurements

A key challenge of TFMCC is for each receiver to be able to measure its RTT to the sender without causing excessive traffic at the sender. In practice the problem is primarily one of getting an initial RTT measurement as, with the use of timestamps in the data packets, a receiver can see changes in the delay on the forward path simply from the packet's arrival time.

RTT Estimate at Initialization Time

Ideally we would like a receiver to be able to initialize its RTT measurement without having to exchange any feedback packets with the sender. This is possible if the sender and the receivers have synchronized clocks, which might be achieved using GPS receivers. Less accurately, it can also be done using clocks synchronized with NTP [MTH97].

In either case, the data packets are timestamped by the sender, and the receiver can then compute the one-way delay. The RTT is estimated to be twice the one-way delay $d_{S \to R}$. In the case of NTP, the errors that accumulate between the stratum-1 server and the local host must be taken into account. An NTP server knows the RTT and dispersion to the stratum-1 server to which it is synchronized. The sum of these gives the worst-case error ϵ in synchronization. To be conservative:

$$t_{RTT} = 2(d_{S \to R} + \epsilon_{sender} + \epsilon_{receiver})$$

In practice NTP provides an average timer accuracy of 20-30 ms [MTH97], and in most cases this gives us an estimate of RTT that is accurate at least to the nearest 100 ms. Although not perfect, this is still useful as a first estimate.

In many cases though, no reliable form of clock synchronization is available. Each receiver must then initialize its RTT estimate to a value that should be larger than the highest RTT of any of the receivers. We assume that for most networks a value of 500 ms is appropriate [All00]. This initial value is used until a real measurement is completed.

RTT Measurement

A receiver gets to measure the instantaneous RTT t_{RTT}^{inst} by sending timestamped feedback to the sender, which then echoes the timestamp and receiver ID in the header of a data packet.[1] If more feedback messages arrive than data packets are sent, we prioritize the sender's report echoes in the following order:

1. a receiver whose report causes it to be selected as the new CLR
2. receivers that have not yet measured their RTT
3. non-CLR receivers with previous RTT measurements
4. the existing CLR.

Ties are broken in favor of the receiver with the lowest reported rate. Normally the number of data packets is larger than the number of feedback packets, so the CLR's last report is echoed in any remaining data packets.

To prevent a single spurious RTT value from having an excessive effect on the sending rate we smooth the values using an exponentially weighted moving average (EWMA)

$$t_{RTT} = \beta \cdot t_{RTT}^{inst} + (1 - \beta) \cdot t_{RTT}$$

[1]To be able to infer an accurate RTT from the timestamps it is necessary to also take into account the offset between receipt of a timestamp and echoing it back.

For the CLR we set $\beta_{CLR} = 0.05$. Given that other receivers will not get very frequent RTT measurements and thus old measurements are likely to be outdated, a higher value of $\beta_{non-CLR} = 0.5$ is used for them.

One-way Delay RTT Adjustments

Due to the infrequent RTT measurements, it would also be possible for large increases in RTT to go unnoticed if the receiver is not the CLR. To avoid this we adjust the RTT estimate between actual measurements. Since data packets carry a send timestamp t_{data}, a receiver that gets an RTT measurement at time t_{now} can also compute the one-way delay from sender to receiver (including clock skew) as

$$d_{S \rightarrow R} \quad = \quad t_{now} - t_{data}$$

and the one-way from receiver to sender as

$$d_{R \rightarrow S} \quad = \quad t_{RTT}^{inst} - d_{S \rightarrow R}$$

Due to clock skew, these values are not directly meaningful, but $d_{R \rightarrow S}$ can be used to modify the RTT estimate between real RTT measurements. When in a later data packet the one-way delay from sender to receiver is determined as $d'_{S \rightarrow R}$, it is possible to compute an up-to-date RTT estimate

$$t_{RTT}^{inst}{}' \quad = \quad d_{R \rightarrow S} + d'_{S \rightarrow R}$$

The updated RTT estimate takes into account any delay changes on the path from sender to receiver but ignores delay changes on the return path. Clock skew between sender and receiver cancels out, provided that clock drift between real RTT measurements is negligible. The modified RTT estimates are smoothed with an EWMA just like normal RTT measurements, albeit with a smaller decay factor for the EWMA since the one-way delay adjustments are possible with each new data packet. One-way delay adjustments are used as an indicator that the RTT may have changed significantly and thus a real RTT measurement is necessary. If the receiver is then selected as the CLR, it measures its RTT with the next packet, and all interim one-way delay adjustments are discarded. For this reason it proved to be unnecessary to filter out flawed one-way delay estimates.

Sender-side RTT Measurements

While a preconfigured initial RTT value can be used at the receiver for loss aggregation and rate computation in case a real measurement is not yet available, it should not be used to set the sending rate. Using a high initial RTT would result in a very low sending rate, followed by a high sending rate when the CLR gets the first RTT measurement, then a CLR change to a receiver with no previous RTT measurement, and so on. Such rate oscillations should be avoided. On the other hand, if the sender only accepted a receiver with a valid RTT as CLR, receivers with a very high loss rate might never receive their feedback echo, and so never become CLR.

For these reasons, TFMCC supports additional sender-based RTT measurements.[2] A receiver report also echoes the timestamp of the last data packet, and so the sender and receivers are both able to measure the RTT. The sender only uses the RTT when it has to react to a receiver report without a valid RTT to be able to adjust the calculated rate in the receiver report. Since the initial RTT value is known to the sender, it can simply multiply the reported rate with the ratio of initial RTT to instantaneous RTT.

Determining the Maximum RTT

The sender-side RTT measurements are also used to compute the maximum RTT t_{RTT}^{max} to all receivers. If the instantaneous RTT measured at the sender is larger than the current maximum RTT, the maximum RTT is increased to this value. Otherwise, if no feedback indicating a higher instantaneous RTT than the maximum RTT is received during a feedback round, the maximum RTT is reduced exponentially to

$$t_{RTT}^{max} = 0.9 \cdot t_{RTT}^{max}$$

which results in a slow decrease over a number of feedback rounds. The maximum RTT is mainly used for feedback suppression among receivers with heterogeneous RTTs, as explained in the following section.

[2] A sender-side RTT measurement is only performed when the sender receives a receiver report. Through the use of feedback suppression, only a very small fraction of receivers will send reports each feedback round, preferentially receivers with a low calculated rate.

7.3.3 Receiver Feedback

As TFMCC is designed to be used with receiver sets of several thousand receivers, it is critical to ensure that the sender gets feedback from the receivers experiencing the worst network conditions without being overwhelmed by feedback from all the other receivers. Congestion may occur at any point in the distribution tree, from the sender's access link through to a single receiver's tail circuit. Thus any mechanism must be able to cope when conditions change from a single receiver being lightly congested to all the receivers being equally heavily congested, and other similarly pathological cases. At the same time we would like the feedback delay to be relatively small in the steady state. The latter can be achieved through the concept of a CLR which can send feedback immediately.

However, a CLR is of no help during a change in network conditions that affect receivers other than the CLR. Thus, we will ignore the influence of the CLR on the feedback process in this section, but we note that the CLR generates relatively little feedback traffic and strictly improves the protocol's responsiveness to congestion while having no impact on the feedback sent by other receivers. The CLR gives feedback independently of all the other receivers, and only non-CLR receivers participate in the feedback suppression process.

For TFMCC, we use a feedback control mechanism based on exponentially weighted random timers, as presented in Section 6.3.3. The receivers set a timer and only send feedback when the timer expires. The dynamics of such a mechanism depend on the way the timers are initialized, and on how one receiver's feedback suppresses another's. Each time the sender receives a feedback message, it updates a so-called *suppression rate* and the maximum RTT in the TFMCC data packet header. This way, the remaining receivers are notified that from now on only feedback reporting an even lower rate than the current suppression rate or with a higher RTT than the current maximum RTT may be sent. Other feedback will be canceled.

The basic exponentially weighted random timer mechanism initializes a feedback timer to expire after t seconds, with

$$t = T \max\left(1 + \log_N x, 0\right) \tag{7.1}$$

T is set to a multiple of the maximum RTT of the receivers; $T = b\ t_{RTT}^{max}$. The choice of b determines the expected number of feedback packets per round as well as the feedback delay. From the feedback analysis in Section 6.3.3, we know that useful values for b lie between 3 and 6.[3] Feedback is given in rounds, where the beginning and end of a feedback round are

[3]For the simulations we use a default value of 4.

indicated by the sender. After a time span of T, the feedback round ends if non-CLR feedback was received during that time. Otherwise, the feedback round ends as soon as the first non-CLR feedback message arrives at the sender but at most after $2T$. The feedback round counter is incremented by one, and the suppression rate is reset to the highest representable value. When a new round begins, receivers cancel outstanding feedback for the old round.

The feedback mechanism is relatively insensitive to overestimation of the receiver set size N, but underestimation may result in a feedback implosion. In our simulations we use $N = 10,000$ which seems reasonable given our scaling goals.

Canceling Feedback

When a receiver sees echoed feedback from another receiver, it must decide whether or not to cancel its feedback timer. For this purpose, we use adaptive class-based extremum detection (i.e., Algorithm 5 described in Section 6.8.1). At the beginning of a feedback round, the suppression rate is set to infinity, and all receivers are eligible to give feedback. Each time the sender receives a feedback message indicating a rate lower than the current suppression rate, the suppression rate is reduced to the lower value. If the receiver's calculated rate is R_{TCP} and the suppression rate from previous feedback is R_{supp}, the timer is canceled if $R_{supp} - R_{TCP} < q\ R_{supp}$ (i.e., $R_{TCP} > (1-q)\ R_{supp}$). As shown in Section 6.8.1, the expected number of feedback messages increases logarithmically with n for $q = 0$. For values of $0 < q \leq 1$, this number becomes approximately constant in the limit for large n. For TFMCC, we use $q = 0.1$.

As the round progresses, more and more high-rate receivers are suppressed until in the end only the receiver with the lowest rate may give feedback. The same applies for receivers with a higher RTT than the current maximum RTT reported by the sender. Such receivers are eligible to give feedback even if their calculated rate is higher than the suppression rate, to notify the sender of an increased maximum RTT.

The feedback suppression process is complicated by the fact that the calculated rates of the receivers will change during a feedback round. If the calculated rates decrease rapidly for all receivers, feedback suppression can no longer prevent a feedback implosion since earlier feedback will always report a higher rate than current feedback. To make the feedback suppression mechanism robust in the face of changing rates, it is necessary to introduce R^0_{supp}, the calculated rate of a receiver at the beginning of a feedback round. A receiver needs to suppress its feedback not only when the suppression rate is less than the receiver's current calculated rate but also in the case that the suppression rate falls below R^0_{supp}.

Biasing Feedback

Optionally, it is possible to bias the receivers' feedback timers in favor of receivers with lower rates to ensure that receivers with low expected rates are more likely to respond than receivers with high rates even at the beginning of a round. Since a receiver knows the sending rate but not the calculated rates of other receivers, a good estimate of the importance of its feedback is the ratio r of its calculated rate R_{TCP} to the current sending rate R_{send}. TFMCC can make use of an additive bias (see Section 6.8.3) as follows:

$$t' = \gamma r T + (1-\gamma)T \cdot (1 + \log_N x) \tag{7.2}$$

where γ determines the fraction of T that should be used to spread out the feedback responses with respect to the reported rate.

While biasing may improve the quality of the feedback received, unbiased feedback results in the sender having a better notion of the distribution of calculated rates at receivers other than the worst. Furthermore, unbiased feedback results in a lower number of expected feedback messages as discussed in Section 6.8.5. In most cases, the above feedback cancellation method together with unbiased feedback timers will suffice.

Unfavorable Feedback Conditions

At very low sending rates and high loss rates (which usually go together), it is still possible to get a feedback implosion. The feedback echo from the sender that suppresses other feedback is sent with the next data packet. Thus, when the delay before the next data packet is sent is close to the feedback delay, it will arrive too late for suppression to work.

This problem can be prevented by increasing the feedback delay T in proportion to the time interval between data packets when the sending rate R_{send} is low:

$$T = b \max\left(t^{max}_{RTT}, (c+1)\frac{s}{R_{send}}\right)$$

c being the number of consecutive data packets that can be lost without running the risk of implosion, and s the packet size. We recommend using values of c between 2 and 4. Alternatively, it is possible to send a separate feedback cancellation message (without payload) to ensure that feedback is canceled on time.

As discussed in Section 6.6.3, loss of feedback messages does not have an impact on the suppression characteristics of the feedback mechanism.

When the maximum RTT changes significantly during one feedback round, it is necessary to reschedule the feedback timer in proportion to the change:

$$t = t \cdot t_{RTT}^{max} / t_{RTT}^{max}{}'$$

where t_{RTT}^{max} is the new maximum RTT and $t_{RTT}^{max}{}'$ is the previous maximum RTT.[4] The same considerations hold in the situation when the last data packets were received more than a time interval of t_{RTT}^{max} ago. In this case, it is necessary to add the difference of the inter-packet gap and the maximum RTT to the feedback time to prevent a feedback implosion (e.g., when the sender crashed).

$$t = t + \max(t_{now} - tr_i - t_{RTT}^{max}, 0)$$

where tr_i is the time when the last data packet arrived at the receiver.

7.3.4 Storing the Previous CLR

As an option, the sender can keep information about the previous CLR after switching to a new CLR. In case the switch-over is only temporary, it is possible to immediately switch back to the old CLR without the need of further feedback. Possible causes for transient switching of the CLR include short-term congestion or inaccurate one-way delay RTT adjustments. Here, the new expected rate may quickly increases above the expected rate of the previous CLR.

Storing this additional information will always result in more conservative TFMCC behavior. In particular, when network conditions for the new CLR as well as the old CLR improve simultaneously, TFMCC will switch back to the old CLR before increasing the sending rate. Since this results in a delayed reaction to improved network conditions, the information about the old CLR should be timed out after a short amount of time (on the order of a few RTTs).

[4] Normally, only some receivers obtain new RTT measurements over the course of a feedback round and these measurements are smoothed, such that abrupt changes in the measured RTT are rare.

7.3.5 TFMCC Startup

At initialization of the sender, the maximum RTT is set to a value that should be larger than the highest RTT to any of the receivers. It should not be smaller than 500 milliseconds for operation in the public Internet. The sending rate is initialized to 1 packet per maximum RTT.

The receiver is initialized when it receives the first data packet. The RTT is set to the maximum RTT value contained in the data packet, unless the receiver has other means to initialize the maximum RTT (e.g., using NTP as discussed before). This initial value is used as the receiver's RTT until the first real RTT measurement is made. The loss event rate is initialized to 0.

Slow-Start

TFMCC uses a slow-start mechanism to more quickly approach its fair bandwidth share at the start of a session. During slow-start, the sending rate increases exponentially, whereas normal congestion control allows only a linear increase. An exponential increase can easily lead to heavy congestion, so great care has to be taken to design a safe increase mechanism. A simple measure to this end is to limit the increase to a multiple d of the minimum rate R_{recv}^{min} received by any of the receivers. Since a receiver can never receive at a rate higher than its link bandwidth, this effectively limits the overshoot to d times that bandwidth. The target sending rate is calculated as

$$R_{send}^{target} = d\, R_{recv}^{min}$$

and the current sending rate is gradually adjusted to the target rate over the course of an RTT. In our implementation we use a value of $d = 2$. Slow-start is terminated as soon as any one of the receivers experiences its first packet loss.

Feedback biasing should not be used during slow-start as most of the receivers will experience the same receive rate, and thus biasing will have an adverse effect on the feedback process. A report from a receiver that experiences the first loss event can only be suppressed by other reports also indicating packet loss, but not by reports from receivers that did not yet experience loss. Therefore, slow-start will be terminated no later than one feedback delay after the loss was detected.

In practice, TFMCC will seldom reach the theoretical maximum of a doubling of the sending rate per RTT for two reasons:

- The target sending rate is increased only when feedback from a new feedback round is received. Thus, doubling is not possible every RTT but every feedback round.
- Measuring the receive rate over several RTTs and gradually increasing R_{send} to R_{send}^{target} gives a minimum receive rate at the end of a feedback interval that is lower than the sending rate during that interval. Thus, setting R_{send}^{target} to twice the minimum receive rate does not double the current sending rate.

As is desirable for a multicast protocol, TFMCC slow-start behaves more conservatively than comparable unicast slow-start mechanisms.

Initializing the Loss History

When a receiver registers its first loss event, the number of packets received thus far usually does not reflect the current loss rate. For example, when the sending rate is constrained by a lower-rate CLR, a receiver may not experience packet loss for a long period of time. Instead of the number of packets received before the first loss event, the sending rate at which the first packet loss is experienced can be used as an indicator of the bottleneck bandwidth. Slow-start results in an overshoot to a maximum of at most twice the bottleneck bandwidth. Thus, a more meaningful initial loss interval l_0 can be obtained by using the inverse of Equation (2.3) with half the sending rate when the first loss event occurred.

The mechanism can be facilitated by using the inverse of a simplified TCP Equation (2.1), which is easier to compute than the inverse of Equation (2.3) and results in a slightly more conservative estimate:

$$\begin{aligned} R_{TCP} &= \frac{cs}{t_{RTT}\sqrt{p}} \\ p &= \left(\frac{cs}{R_{TCP} \cdot t_{RTT}}\right)^2 \text{, with } l_0 = 1/p \end{aligned}$$

where c is a constant usually set to $\sqrt{3/2}$.

If a receiver is still using the initial RTT when the first loss event occurs, it will underestimate the loss event rate, and the initial loss interval will be too large. When the correct RTT is determined later, the receiver will consequently overestimate the fair rate. The initial loss interval must be adjusted if it is still in the loss history when the first RTT measurement is obtained. The adjusted first loss interval l'_0 can be calculated as

$$l'_0 = l_0 \cdot \left(\frac{t_{RTT}}{t_{RTT}^{initial}} \right)^2$$

using the simplified TCP equation.

Using the Initial RTT for the Aggregation of Loss Events

Using the initial RTT for the rate computation before a valid RTT measurement is obtained is safe since a higher RTT leads to a lower calculated rate. In contrast, using the initial RTT for the aggregation of lost packets to loss events may result in a more aggressive protocol behavior. The larger the RTT, the more lost packets may be aggregated to a single loss event which reduces the loss event rate and in turn results in a higher sending rate.

Nevertheless, it is still possible to use the initial RTT for both purposes since the former effect outweighs the latter. The size of the loss intervals can only increase in proportion to the ratio of the initial RTT to the true RTT. Using Equation (7.3), an initial RTT that is too high by a factor of c will allow for a loss rate that is too low by a factor of c^2 resulting in the same throughput. The rate calculated at the receiver will therefore still be conservative. Numerical analysis indicates that this also holds for the complex TCP model (2.3) when loss event rates are less than approximately 10%.

If there are many receivers with a high loss rate, throughput will be very low (see Section 7.5). If there are few such receivers, these receivers can measure their RTT soon after startup. For these reasons, it is safe to use a high initial RTT to aggregate losses to loss events as well as to compute the rate.

7.4 Providing Congestion Information to the Application

As a single-rate congestion control scheme, TFMCC adapts the sending rate to the receiver experiencing the worst network conditions (also referred to as the “crying baby” syndrome). For heterogeneous receiver sets, this may result in a sending rate well below the average acceptable rate of the group. Depending on the application, it may be desirable to exclude some receivers with network conditions significantly worse than those of the other receivers from the group.

Only the application itself can decide when it is appropriate to exclude receivers and which receivers to exclude, but the congestion control mechanism can improve the decision making process by providing information about the distribution of acceptable rates among the receivers.

In its simplest form the protocol can provide the second lowest calculated rate to the application. This way, the application knows what rate increase to expect when removing the CLR from the session. The basic mechanism can be extended to a list of n worst receivers to allow the application to remove several receivers at once. Furthermore, an estimate of the total number of receivers may be of use to the application (e.g., removing 5 out of 6 receivers to improve the rate for the remaining receiver is usually not desirable). In [FT99, LN00], methods are presented to estimate the size of the receiver set from the time interval until the first feedback message is received. While these mechanism were designed for basic exponentially distributed feedback, they can readily be used for TFMCC with the previously described feedback cancellation mechanism and unbiased feedback timers. Here, the expected value of the feedback time of the first responder remains unchanged. If biased feedback timers are used, a more complex estimation process is necessary. When the number of receivers is known to the application, it can trade off the loss of utility caused by the removal of certain receivers against the potential performance improvement for the remaining receivers.

7.5 Protocol Behavior with Very Large Receiver Sets

The loss path multiplicity problem is a well-known characteristic of multicast congestion control mechanisms that react to single loss indications from receivers on different network paths. It prevents the scaling of those mechanisms to large receiver sets. In [BTK99], the authors propose as a possible solution to track the most congested path and to take only loss indications from that path into account. Since the reports of a TFMCC receiver contain the expected rate based on the loss event rate and RTT on the single path from the sender to that receiver, the protocol implicitly avoids the loss path multiplicity problem. Yet TFMCC (and all other single-rate congestion control schemes) may be confined to a rate below the fair rate if, rather than there being a single most congested path, there is a path that changes over time. The faster a multicast congestion control protocol responds to transient congestion, the more pronounced is the effect of tracking the minimum of stochastic variations in the calculated rate at the different receivers. For example, if loss to several receivers independently varies fairly quickly between 0% and 10% with the average being 5%, a congestion control protocol may always track the worst receiver, giving a loss estimate that is twice what it should be.

A worst-case scenario in this respect is a high number of receivers with independent loss and a calculated rate in the range of the lowest-rate receiver. If n receivers experience independent packet loss with the same loss probability, the loss intervals will have an exponential distribution. The expected value of the minimum of n exponentially distributed random variables is proportional to $1/n$. Thus, if TFMCC based its rate calculations on a single loss interval, the average sending rate would scale proportionally to $1/\sqrt{n}$ (in the case of moderate loss rates, otherwise even worse). The rate calculation in TFMCC is based on a weighted average of m loss intervals. Since the average of exponentially distributed random variables is gamma distributed, the expected loss rate in TFMCC is inversely proportional to the expected value for the minimum of n gamma distributed random variables.[5]

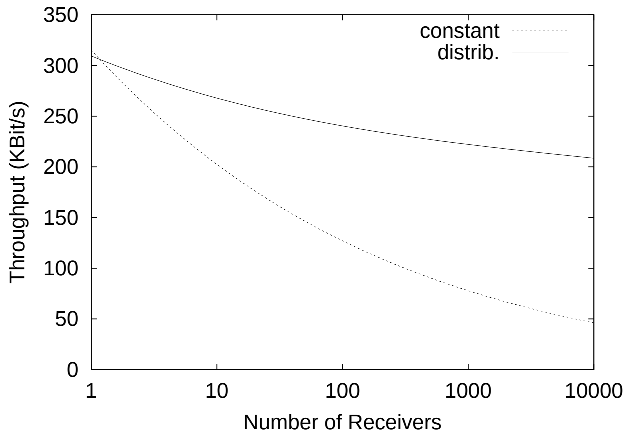


Figure 7.1: Scaling

This effect is shown in Figure 7.1 for different numbers of receivers n with a constant loss probability. For uncorrelated loss at a rate of 10% and a RTT of 50 ms, the fair rate for the TFMCC transmission is around 300 kBit/s. This sending rate is reached when the receiver set consists of only a single receiver but it quickly drops to a value of only a fraction of the fair rate for larger n. For example, for 10,000 receivers, only 1/6 of the fair rate is achieved.

Fortunately, such a loss distribution is extremely unlikely in real networks. Multicast data is transmitted along the paths of the distribution tree of the underlying multicast routing protocol. A lossy link high up in the tree may affect a large number of receivers, but the losses are correlated

[5]For first order statistics of the gamma distribution, no simple closed form expressions exists. Details about the distribution of the minimum of gamma distributed random variables can be found in [Gup60].

and so the above effect does not occur. When some of those receivers have additional lossy links, the loss rates are no longer correlated, rather the values are spread out over a larger interval, thus decreasing the number of receivers with similar loss rates. To demonstrate this effect, we choose a distribution of loss rates that is closer to actual loss distributions in multicast trees in that there are only a limited number of high loss receivers while the majority of receivers will have moderate loss rates.[6] Here, a small number of receivers (proportional to $a \log(n)$, where a is a constant) is in the high-loss range of 5-10%, some more are in the range of 2%-5%, and the vast majority have loss rates between 0.5% and 2%. Under such network conditions the throughput degradation with 10,000 receivers is merely 30%. Thus, the throughput degradation plays a significant role only when the vast majority of packet loss occurs on the last hop to the receivers and those losses amount to the same loss rates.

It is impossible to distinguish between a “stochastic” decrease in the sending rate and a “real” decrease caused by an increased congestion level (otherwise it would be possible to estimate the effect and adjust the sending rate accordingly). The degradation effect can be alleviated by increasing the number of loss intervals used for the loss history, albeit at the expense of reduced responsiveness.

7.6 Protocol Simulations

We implemented TFMCC in the *ns*-2 network simulator [BBE$^+$99] to investigate its behavior under controlled conditions. In this chapter, we can only report a fraction of the simulations that were carried out. In all simulations below, drop-tail queues were used at the routers to ensure acceptable behavior in the current Internet. Generally, both fairness towards TCP and intra-protocol fairness improve when active queuing (as for example RED) is used instead.

7.6.1 Fairness

Fairness towards competing TCP flows was analyzed using the common single-bottleneck topology shown in Figure 7.2, where a number of sending nodes are connected to the same number of receiving nodes through a common bottleneck (indicated by a thick line).

[6]By no means do we claim that the chosen distribution exactly reflects network conditions in multicast distribution trees.

The right graph of Figure 7.2 shows the throughput of a TFMCC flow and two sample TCP flows (out of 15) from a typical example of such simulations. The average throughput of TFMCC closely matches the average TCP throughput but TFMCC achieves a smoother rate. Similar results can be obtained for many other combinations of flows. In general, the higher the level of statistical multiplexing, the better the fairness among competing flows. Only in scenarios where the number of TFMCC flows greatly exceeds the number of TCP flows is TFMCC more aggressive than TCP. The reason for this lies in the spacing of the data packets and buffer requirements: TFMCC spaces out data packets, while TCP sends them back-to-back if it can send multiple packets, making TCP more sensitive to nearly-full queues typical of drop-tail queue management.

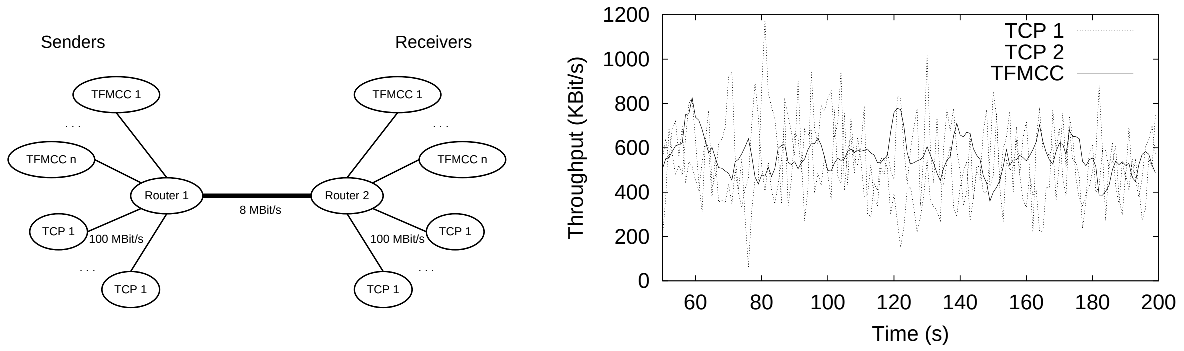


Figure 7.2: One TFMCC flow and 15 TCP flows over a single 8 MBit/s bottleneck

If instead of one bottleneck the topology has separate bottlenecks on the last hops to the receivers, then we observe the throughput degradation predicted in Section 7.5. When the scenario above is modified such that TFMCC competes with single TCP flows on sixteen identical 1 MBit/s tail circuits, then TFMCC achieves only 70% of TCP's throughput (see Figure 7.3).

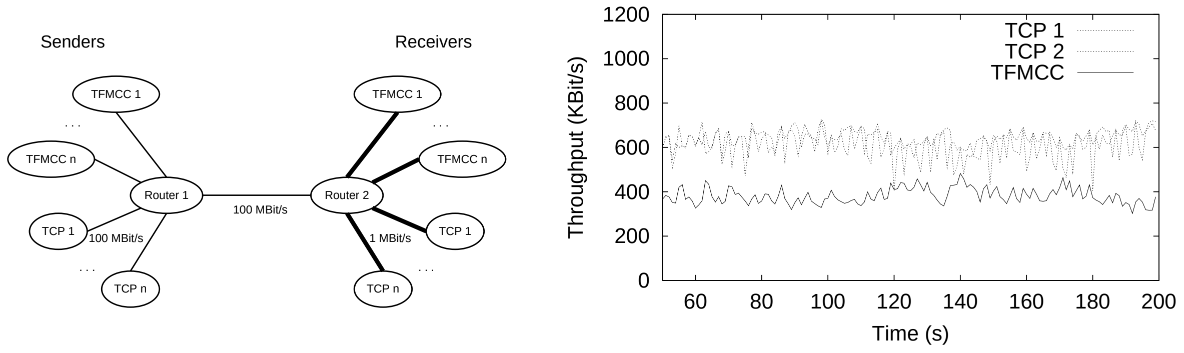


Figure 7.3: 1 TFMCC flow and 16 TCP flows (individual bottlenecks)

7.6.2 Responsiveness to Changes in the Loss Rate

An important concern in the design of congestion control protocols is their responsiveness to changes in network conditions. Furthermore, when receivers join and leave the session it is important that TFMCC react sufficiently fast should a change of CLR be required. This behavior is investigated using a star topology with four links having an RTT of 60 ms and loss rates of 0.1%, 0.5%, 2.5%, and 12.5% respectively. At the beginning of the simulation the receiver set consists only of the receiver with the lowest loss rate. Other receivers join the session after 100 seconds at 50 second intervals in the order of their loss rates (lower-loss-rate receivers join first). After 250 seconds, receivers leave the transmission in reverse order, again with 50 second intervals in between. To verify that TFMCC throughput is similar to TCP throughput, an additional TCP connection to each receiver is set up for the duration of the whole experiment.

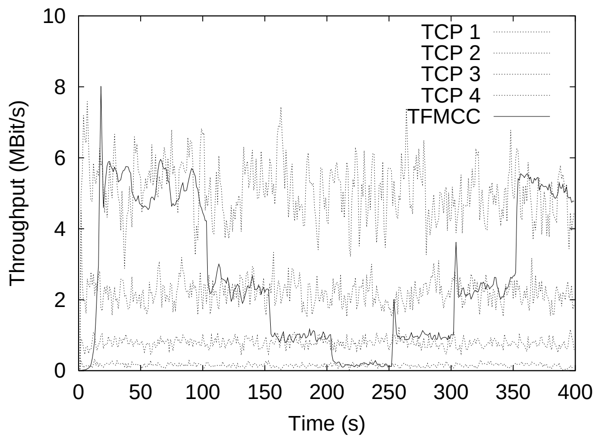


Figure 7.4: Responsiveness to changes in the loss rate

As show in Figure 7.4, TFMCC matches closely the TCP throughput at all four loss levels. Adaption of the sending rate when a new higher-loss receiver joins is fast. The receiver needs 500-1000 ms after the join to get enough packets to compute a meaningful loss rate. The major part of the delay is caused by the exponential timer for the feedback suppression, which increases the overall delay before a new CLR is chosen to roughly one to three seconds.[7] The experiment demonstrates TFMCC's very good reactivity to changes in the congestion level.

The delay before TFMCC assumes that a rate-limiting receiver left the group and the sending rate can be increased is configurable. Currently, an absence of feedback from the CLR for 10

[7]Note that this high delay is caused by the use of the initial RTT in the feedback suppression mechanism. Once all receivers have a valid RTT estimate, the delay caused by feedback suppression is much shorter.

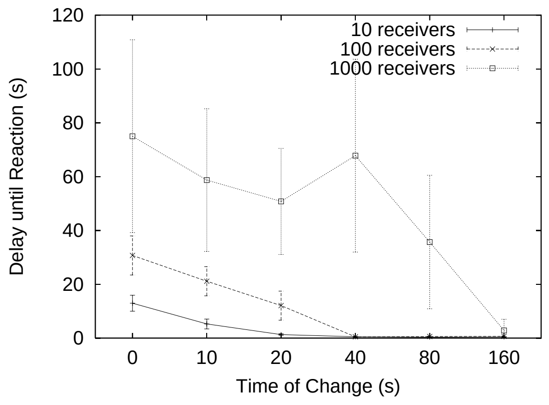


Figure 7.5: Responsiveness to RTT changes

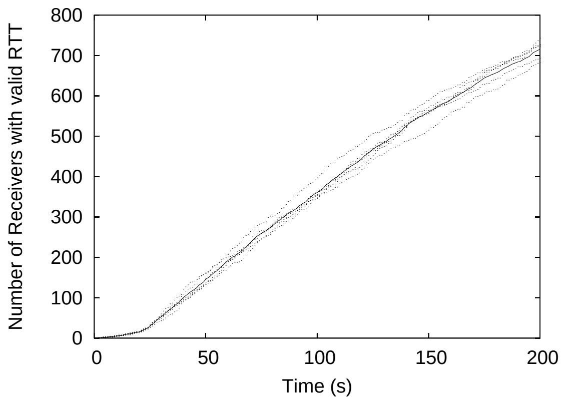


Figure 7.6: Rate of initial RTT measurements

times the feedback delay is used as an indication that this receiver left the group. In case explicit leave messages are used with the TFMCC protocol the delay can be reduced to one RTT.

The same simulation setting can be used to investigate responsiveness to changes in the RTT. The results (not shown here) are similar to those above, since all four receivers have measured their RTT by the time the RTT changes, and the one-way RTT adjustments immediately indicate this change. With larger receiver sets, the amount of time that expires until a high RTT receiver is found may be greater. This effect is investigated in the next section.

7.6.3 Responsiveness to Changes in the RTT and Rate of Initial RTT Measurements

In scenarios with 40, 200 and 1000 receivers respectively, we investigate how long it takes until a high RTT receiver is found among receivers with a low RTT when all receiver experience independent loss with the same loss probability. The x-axis of the graph in Figure 7.5 denotes the point of time when the RTT is increased during the experiment, and the y-axis shows the amount of time after which this change in RTT is reacted upon by choosing the correct CLR. The later the increase in RTT, the greater the number of receivers already having valid RTT estimates, and the expected time until the high-RTT receiver is selected as CLR decreases.

The time interval until the correct CLR is selected may seem large but we have chosen a relatively artificial scenario with independent identical loss rates for all receivers. Before the first RTT measurement, the receiver with a high RTT is indistinguishable from the other receivers. Therefore, this time interval exclusively depends on the rate of RTT measurements.

The number of receivers that measure their RTT in each feedback round depends on the number of feedback messages and thus on the parameters used for feedback suppression. Figure 7.6 shows how the number of receivers with a valid RTT estimate evolves over time for a large receiver set and a high initial RTT value. The link RTTs for the 1000 receivers vary between 60 ms and 140 ms, and the initial RTT value is set to 500 ms. A single bottleneck is used to produce highly correlated loss for all receivers. This is the worst case scenario, as with varying loss estimates at the receivers it is unnecessary to measure the RTT to the low-loss receivers. Since the calculated rate of the receivers still using the initial RTT is below the current sending rate, at least one receiver will get its first RTT measurement per feedback round until all receivers have measured their RTTs.

At the beginning of the simulation, the number of receivers obtaining initial RTT measurements is close to the expected number of feedback messages per feedback round. Over time, as more and more receivers have a valid RTT, the number of receivers that want to give feedback decreases, and the rate of initial RTT measurements gradually drops to one new measurement per feedback round. Again, a delay of 200 seconds until 700 of the 1000 receivers have measured their RTTs seems rather large but one should keep in mind that this results from having the same congestion level for all receivers. If some receivers experience higher loss rates, those receivers will measure their RTTs first, and TFMCC can adapt to their calculated rate. Under more realistic network conditions RTT and loss rate are likely to be positively correlated and it will not be necessary to measure the RTT to all receivers.

7.6.4 Slow-Start

The highest sending rate achieved during slow-start is largely determined by the level of statistical multiplexing. To demonstrate this effect, we run TFMCC simulations without competing traffic, with one competing TCP flow, and with 16 competing TCP flows. The fair rate for TFMCC in all three simulations is 1 MBit/s. On an otherwise empty link, TFMCC will reach roughly twice the bottleneck bandwidth before leaving slow-start, as depicted in Figure 7.7. When TFMCC competes with a single TCP flow, slow-start is terminated at a rate below the fair rate of the TFMCC flow, and this rate is relatively independent of the number of TFMCC receivers. Already in the case of two competing TCP flows, and even more so when the level of statistical multiplexing is higher, the slow-start rate decreases considerably when the number of receivers increases. Most of the increase to the fair rate takes place after slow-start in normal congestion control mode. Since TCP slow-start is more aggressive than TFMCC slow-start, the

TCP flows are already using the full capacity of the link when TFMCC starts to increase the rate, and it will likely experience a packet drop before slow-start is able to reach the fair share of the bandwidth.

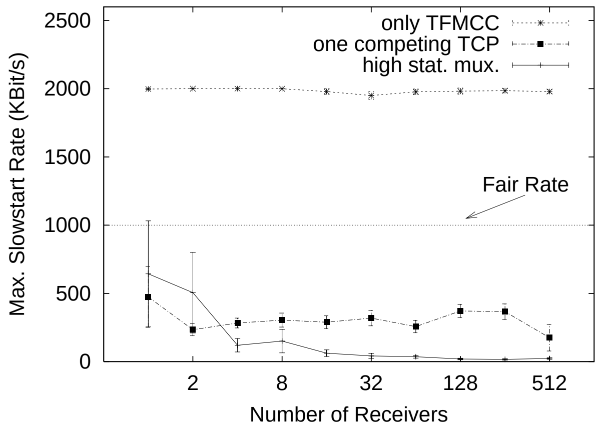


Figure 7.7: Maximum slow-start rate

We do not include an extra graph of the exact increase behavior of TFMCC compared to TCP, since this can be seen, for example, in Figures 7.8 and 7.9. TFMCC and TCP are started at the same time. TCP's increase to the fair rate is very rapid, while it takes TFMCC roughly 20 seconds to reach that level of bandwidth.

7.6.5 Late-join of a Low-rate Receiver

In the previous experiments we investigated congestion control with moderate loss rates, expected to be prevalent in the application domains for which TFMCC is well suited. Under some circumstances, the loss rate at a receiver can initially be much higher. Consider an example where TFMCC operates at a fair rate of several MBit/s and a receiver with a very low-bandwidth connection joins. Immediately after joining, this receiver may experience loss rates close to 100%. While such conditions are difficult to avoid, TFMCC should ensure that they exist only for a limited amount of time and quickly choose the new receiver as CLR.

The initial setup for this simulation is an eight-member TFMCC session competing with seven TCP connections on an 8 MBit/s link, giving a fair rate of 1 MBit/s. During the simulation, a new receiver joins the session behind a separate 200 kBit/s bottleneck from the sender from time 50 to 100 seconds.

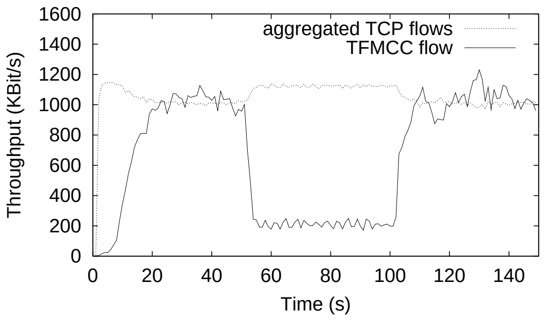


Figure 7.8: Late-join of a low-rate receiver

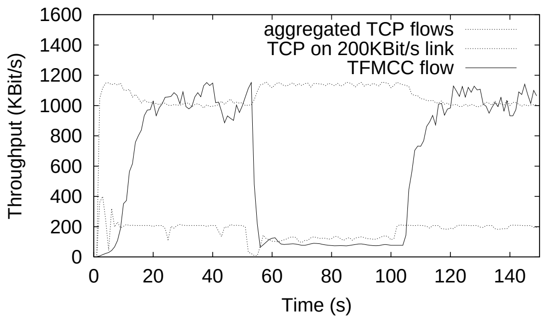


Figure 7.9: Additional TCP flow on the slow link

TFMCC does not have any problems coping with this scenario, choosing the joining receiver as CLR within a very few seconds. Although the loss rate for the joining receiver is initially very high, the TFMCC rate does not drop to zero. As soon as the buffer of the 200 kBit/s connection is full, the receiver experiences the first loss event, and the loss history is initialized as specified in Section 7.3.5. When the first loss occurs, the receiver gets data at a rate of exactly the bottleneck bandwidth. Thus, the loss rate will be initialized to a value below the 80% value and from there adapt to the appropriate loss event rate such that the available bandwidth of 200 kBit/s is used.

When an additional TCP flow is set up using the 200 kBit/s link for the duration of the experiment, this flow inevitably experiences a timeout when the new receiver joins the multicast group and the link is flooded with packets, as shown in Figure 7.9. However, shortly afterwards, TFMCC adapts to the available capacity, and TCP recovers with bandwidth shared fairly between TFMCC and TCP.

We conclude that TFMCC shows good performance and fairness, even under unfavorable network conditions.

7.7 *libtfmcc*: an Implementation of the TFMCC Protocol

The TFMCC protocol has been implemented as a library in accordance with the specifications of the corresponding Internet draft [WH01c]. The library *libtfmcc* provides TFMCC sender and receiver functionality together with an interface to the application for passing data packets to and from the library and setting TFMCC-specific parameters. The source code for the TFMCC library is publicly available.[8]

The main components of the library are as follows:

- `Tfmcc_Sender` is an implementation of the sender side of the TFMCC protocol.
- `Tfmcc_Sink` is the corresponding implementation of TFMCC receiver functionality.
- `Tfmcc_Application_Sender` provides an interface for applications using the TFMCC protocol for the transmission of data.
- `Tfmcc_Application_Sink` is the corresponding interface for data reception.

In addition to the aforementioned classes, the library uses the class `Udp` as a simple interface to UDP sockets and `Data` as a helper class representing data packets. A more detailed description of the architecture of the TFMCC library can be found in Appendix B.2 and in [Höt02]. The header formats for data packets and control packets used by the library are described in Appendix B.4 and B.3.

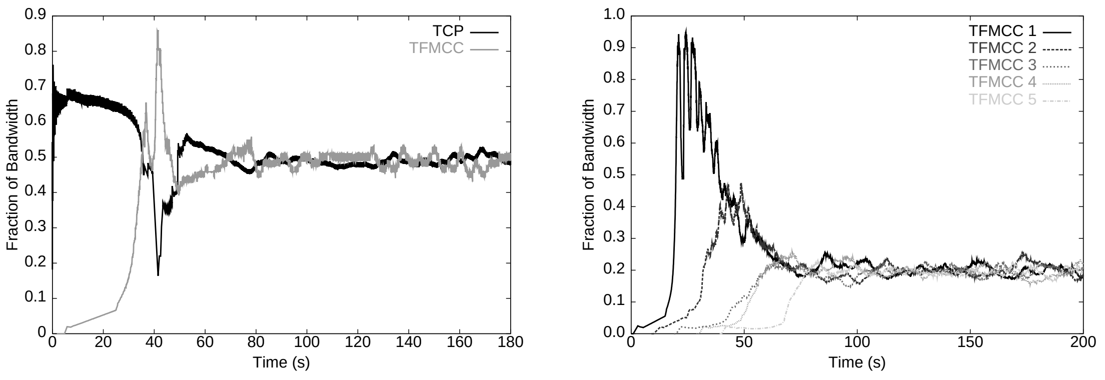
7.7.1 Network Experiments with the TFMCC Library

Similar to the experiments with the ns-2 implementation of the TFMCC protocol, we performed a number of tests with the implementation of the TFMCC library. In the Internet, it is very difficult to study protocol behavior under well-defined conditions. As for the TFRC experiments reported in Section 3.4.2, we therefore use *Dummynet* [Riz98] to simulate arbitrary network conditions. Dummynet is implemented as a filter in the protocol stack of the operating system. Packets that are handed from one network layer to another can be intercepted and passed through pipes, which allow to add bandwidth limitations, delay, and packet drops.

For the first simulation we use one TFMCC sender and one TFMCC receiver, connected via a FreeBSD router with Dummynet enabled. The throughput is limited to 1 MBit/s, and an additional link delay of 80 ms is added to the buffer delay. Throughput is measured at the receiver using *tcpdump* [MJ93] (i.e., after the bottleneck).

As in the ns-2 simulations, TFMCC shows high inter- and intra-protocol fairness. When TFMCC competes with a TCP connection, both settle at the fair rate of half the available capacity after

[8] *libtfmcc* was developed under the RedHat Linux 7.1 operating system and was further tested under FreeBSD 4.4.



a) Fairness to TCP b) Intra-protocol fairness

Figure 7.10: TFMCC throughput over time

TFMCC slow-start terminates. Similar behavior can be experienced when running multiple concurrent TFMCC connections. In the second experiment, TFMCC connections are started one after the other with an interval of 10 seconds in between. Again, the TFMCC flows share the bandwidth fairly, once slow-start is over. Also, the performance during slow-start itself is acceptable, causing no excessive packet drop rates or shutting out other flows.

7.7.2 MPEG Streaming Using the TFMCC Library

To gain a first insight into the suitability of TFMCC for media streaming, we combined the TFMCC library with an existing video codec. We used a very simple rate adaptation policy, where the encoding rate was set directly to TFMCC rate. The video data was packetized and written into the TFMCC payload without additional information. For an interoperable video tool, packetization in accordance with the corresponding RTP MPEG1/MPEG2 payload format [HFGC98] would be the correct choice.

As such, the resulting tool suffices for a first evaluation but obviously, significant improvements are possible with a more sophisticated rate adaptation mechanism, an optimized buffering strategy at the sender, look-ahead prediction of the encoding rate given the current quantization parameters, and RTP conformance. Yet, it can be seen as a first step towards a TFMCC-based video streaming solution.

The implementation of the MPEG-1 streaming system consists of a sender and a receiver part:

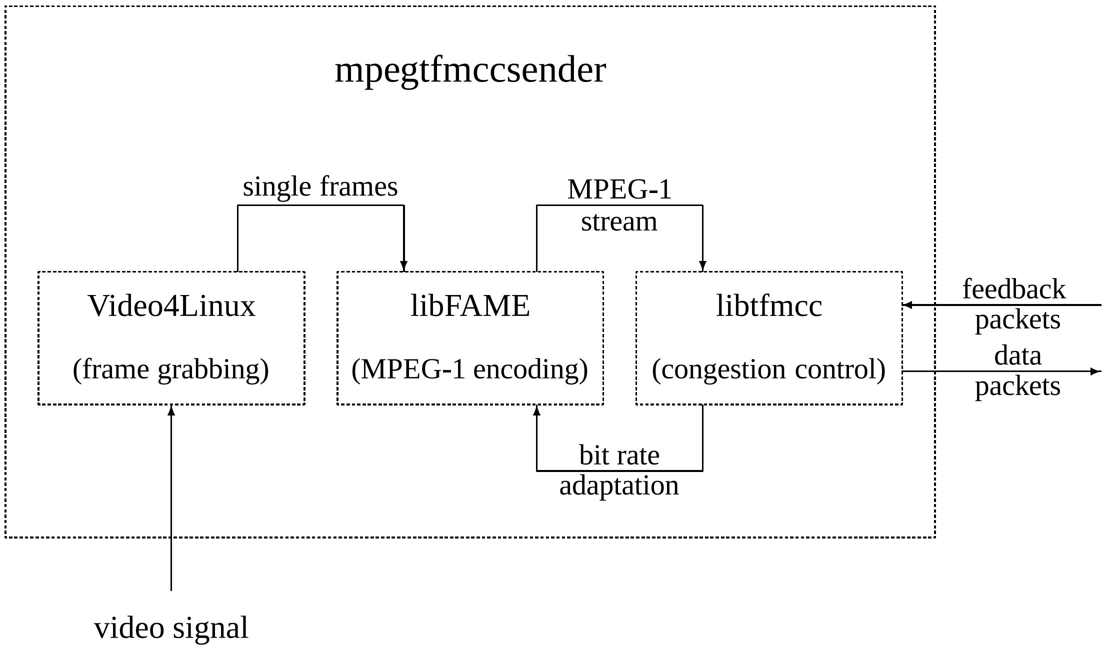


Figure 7.11: Architecture of the sender

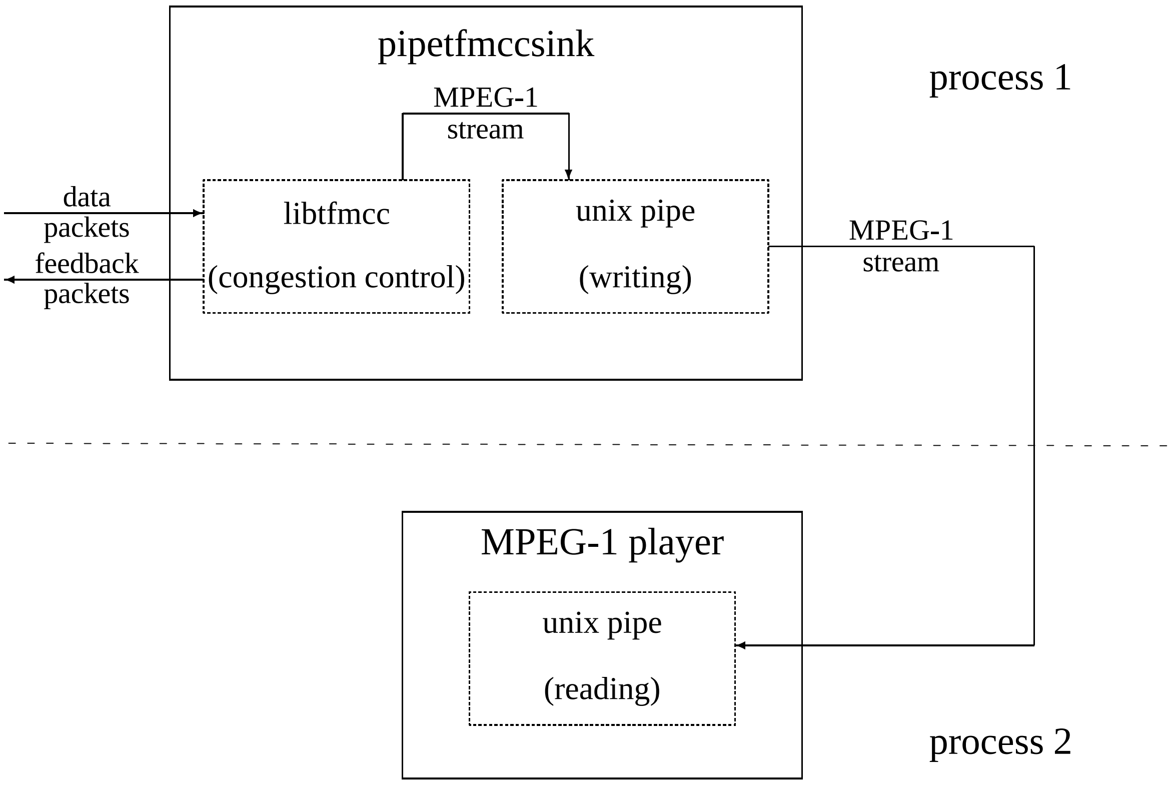


Figure 7.12: Architecture of the receiver

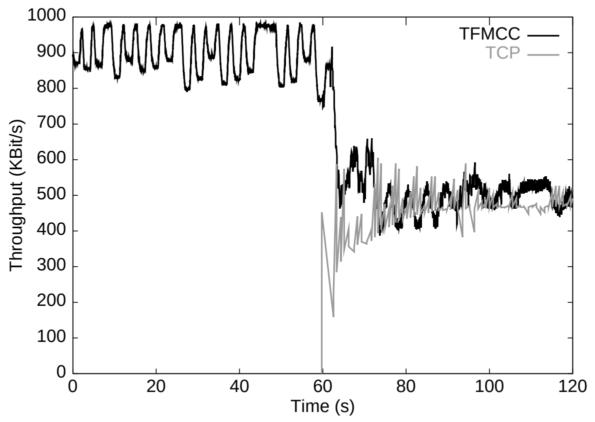


Figure 7.13: TFMCC throughput over time

- The sender *mpegtfmccsender* obtains a video stream from a *Video4Linux* device, which provides a general API for Linux to access video hardware. The video stream is then encoded as an MPEG-1 stream, packetized, and sent to the multicast group (see Figure 7.11). For the encoding, the open source C library *libFAME*[9] was used. It allows the encoding of MPEG-1 video streams in real-time. The TFMCC sender continuously gives feedback to the encoder to ensure that the current encoding rate conforms to the TCP-friendly rate.
- The receiver *pipetfmccsink* forwards the video stream to an external MPEG-1 decoder using a unix pipe (see Figure 7.12).

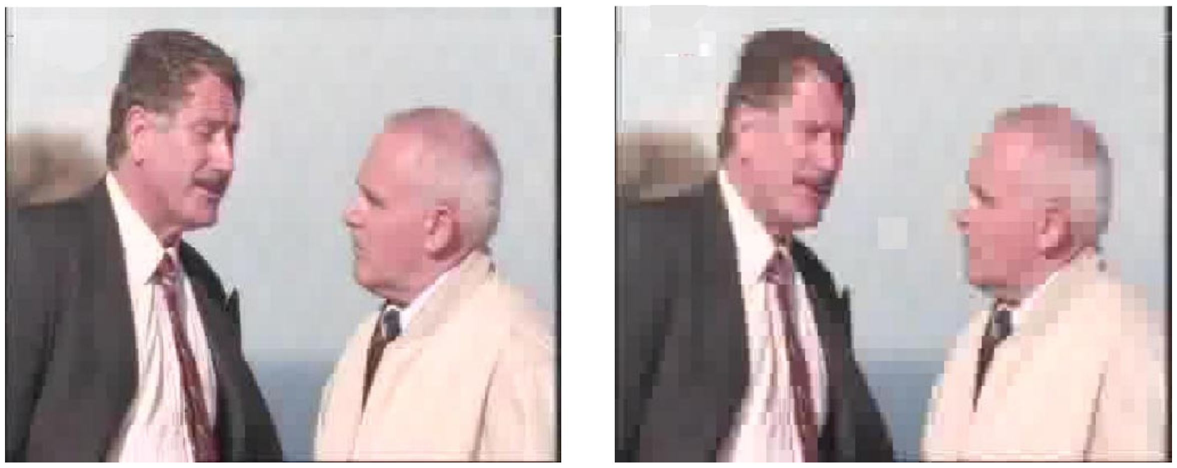
For congestion control, sender and receiver make use of the aforementioned TFMCC library.

7.7.3 MPEG Test Results

Obviously, the minimum bit rate that *libFAME* can produce depends on the chosen resolution and frame rate of the video. With a resolution of 352x288 pixels and a frame rate of 25 frames per second, the minimum bit rate is approximately 500 kBit/s. At lower bit rates, the perceived video quality decreases rapidly. Again, a real video transmission tool would have to implement spatial and temporal scaling to cover a wider range of possible bit rates.

Figure 7.13 shows how the bit rate available for video streaming changes when a competing TCP flow is started after 60 seconds. While a fair distribution of bandwidth is achieved after a few seconds, naturally the video quality decreases. At time $t = 30$s, the video stream can make use of the full capacity of 1 MBit/s. Figure 7.14a shows a video frame coded at that point of time

[9]http://fame.sourceforge.net



a) Video quality at 1 MBit/s b) Video quality at 0.5 MBit/s

Figure 7.14: Variable bit rate video coding

with the given TCP-friendly rate. At $t = 90$s, after the TFMCC sender has adjusted the sending rate, encoding the video with 500 kBit/s corresponds to the video quality shown in Figure 7.14b. In particular, the sudden change in quality at time $t = 60$s decreases the perceived video quality significantly. Such abrupt changes can be avoided to some degree by a more sophisticated buffer strategy at sender and receiver, but optimizing video transmission over IP networks is out of the scope of this dissertation. Interested readers are referred to [Kuh01] instead.

Chapter 8

Conclusions

We believe that the emergence of congestion control mechanisms for relatively smooth congestion control for unicast and multicast traffic can play a key role in preventing the degradation of traffic flows in the public Internet, by providing a viable alternative for multimedia flows that would otherwise be tempted to avoid congestion control altogether [FF99]. The design of good congestion control mechanisms is a hard problem, even more so for multicast environments where scalability issues are much more of a concern than for unicast.

8.1 Contributions

Equation-based congestion control provides a very good platform for the development of control mechanisms that cater to applications for which TCP or other forms of AIMD mechanisms are not well suited. An important example are applications running over UDP. In this dissertation, we have studied a number of different applications and analyzed how equation-based congestion control can be adapted to meet their requirements.

Starting out from unicast equation-based congestion control we investigated, how such a mechanism can be used in the context of non-adaptable flows. This type of flow carries data at a rate determined by the application. It cannot be adapted to the level of congestion in the network in any way other than by suspending the entire flow. Existing congestion control mechanisms that adjust the sending rate are thus not viable for non-adaptable flows.

Instead of striving for TCP-friendliness for each single network flow, out *probabilistic congestion control scheme* suspends individual flows in a way that the aggregation of all non-adaptable flows on a given link behaves in a TCP-friendly manner. The decision about suspending a given flow is made by means of random experiments. This type of congestion control can complement conventional congestion control in the regime where rate adaption is not possible.

In a series of simulations we have shown that PCC displays a TCP-friendly behavior under a wide range of network conditions. We have further identified conditions under which PCC throughput does not correspond to the TCP-friendly rate, but either undershoots or overshoots this rate. To some extent, these effects on the average PCC sending rate cancel each other out but nevertheless, they may degrade PCC performance.

The second large application area for which equation based congestion control is well suited is multicast congestion control. Before going into the details of protocol design, we investigated a very important component of all multicast protocols that rely on feedback from receivers, namely *feedback suppression*. The main complexity in the design of a multicast congestion control protocol lies in such a feedback mechanism. With receiver sets of several thousand receivers, it is critical to ensure that the sender gets timely feedback from the receivers experiencing the worst network conditions without being overwhelmed by feedback from all the other receivers.

We analyzed three feedback algorithms with respect to their suppression characteristics in the face of an inaccurate estimation of the actual group size. Only one of these algorithms, feedback suppression with exponentially distributed timers, is able to provide sufficiently stable expected values across a large range of group sizes.

We further improved upon the concept of exponential feedback suppression in case *feedback of an extremum value* of the group is needed. We discussed two orthogonal methods to improve the quality of the feedback given. If no information is available about the distribution of the values at the receivers, a safe method to obtain better feedback is to modify the suppression mechanism to allow the sending of high valued feedback even after the first feedback was given. We specified exact bounds for the expected increase in feedback messages for a given improvement in feedback quality. If more information about the distribution of feedback values is available or certain worst-case distributions are very unlikely, it is furthermore possible to bias the feedback timer. The better the feedback value, the earlier the feedback is sent, thus suppressing later feedback with less optimal values. The modified suppression mechanism and the feedback biasing can be used in combination to further improve the feedback process. The feedback mechanisms were developed with multicast congestion control in mind but can also be used for a number of other applications where feedback from a large multicast group is required.

The main contribution of this dissertation is the *TCP-Friendly Multicast Congestion Control protocol* (TFMCC), an equation-based single-rate congestion control mechanism intended to scale to groups of several thousand receivers. In general, TFMCC has a low variation of throughput, which makes it suitable for applications such as streaming media where a relatively smooth sending rate is of importance. The penalty of having smooth throughput while competing fairly for bandwidth is a reduced responsiveness to changes in available bandwidth. Thus TFMCC should be used when the application has a requirement for smooth throughput, in particular, avoiding halving of the sending rate in response to a single packet drop.

The challenges in the design of TFMCC lie in scalable round-trip time measurements, appropriate feedback suppression, and in ensuring that feedback delays in the control loop do not adversely affect fairness towards competing flows. The key component of end-to-end multicast congestion control schemes is the feedback control mechanism which largely determines the overall protocol behavior. For this task, TFMCC makes use of the previously discussed extremum feedback mechanism. To further reduce the delay in the congestion control loop, we have introduced the concept of the *current limiting receiver* (CLR). The CLR is the receiver that the sender believes currently has the lowest expected throughput of the group. The CLR is permitted to send immediate feedback without any form of suppression once per round-trip time, hence reducing the delay before TFMCC reacts to changes in the network conditions on the path to the CLR to a single round-trip time. The CLR will change if another receiver sends feedback indicating that a lower than the current transmission rate is required.

With a single-rate congestion control protocol that adapts to the slowest receiver, the sending rate may be very low in a large group of heterogeneous receivers. In order to enable the application to remove receivers that unduly impair protocol performance, it is necessary to provide the application with information about the heterogeneity of the group. This information is (partially) available to the congestion control mechanism through the receiver feedback process and can be made available to the application. With an approximate distribution of TCP-friendly rates of the receivers, the application can then calculate the expected improvement in the sending rate when removing certain receivers from the group and take appropriate action.

An important part of any research is to identify the limitations of a new design. TFMCC's main weakness is in the startup phase – it can take a long time for sufficiently many receivers to measure their RTT (assuming we cannot use NTP to provide approximate default values). In addition, with large receiver sets, TCP-style slow-start is not really an appropriate mechanism, and a linear increase can take some time to reach the correct operating point. However, these limitations are not specific to TFMCC but hold for single-rate multicast congestion control in general if the mechanisms are designed to be TCP-compatible. The implication is therefore that single-rate multicast congestion control mechanisms like TFMCC are only really well-suited to relatively long-lived data streams. Fortunately it also appears that most current multicast applications such as stock-price tickers or video streaming involve just such long-lived data-streams.

We have extensively *evaluated TFMCC through analysis and simulation*, and believe we have a good understanding of its behavior in a wide range of network conditions. Under the sort of conditions TFMCC is likely to experience in the real-world it will behave well. However we have also examined certain pathological cases; in these cases the failure mode is for TFMCC to achieve a lower than desired transmission rate. Given that all protocols have bounds to their good behavior, this is the failure mode we would desire, as it ensures the safety of the Internet. To demonstrate that TFMCC is a suitable congestion control mechanism for streaming media, it was implemented as a library and integrated into a multicast video transmission tool for MPEG.

Summing up, we conclude that performing multicast congestion control while remaining TCP-friendly is difficult, in particular because TCP's transmission rate depends on the RTT, and measuring RTT in a scalable manner is a hard problem. Given the limitations of end-to-end protocols, we believe that TFMCC represents a significant improvement over previous work in this area.

8.2 Areas of Future Research

As is always the case with an evolving research area, several unresolved issues remain. One particular problem is the lack of standard methods to compare congestion control protocols. Thus, an evaluation is often based on hints given in the corresponding papers and quite a bit of guesswork. A test environment (the *ns*-2 network simulator comes to mind) with a standardized suite of test scenarios that investigate different important aspects such as fairness and scalability, combined with measures to directly compare the protocol performance, would be very handy. While such a testbed is not sufficient to explore all details of a specific protocol, it would provide a reasonable basis for more objective protocol comparisons.

While TCP-friendliness is a useful fairness criterion in today's Internet, it is well possible that future network architectures (in which TCP may no longer be the predominant transport protocol) will allow or require different definitions of fairness. Also, fairness definitions for multicast are still subject to research. We presented one possible definition and also briefly addressed a different form where multicast flows are allowed to consume a higher percentage of bandwidth than are unicast flows, but these are by no means the only possible fairness definitions. For a detailed discussion of fairness issues the reader is referred to [Den03].

Many current congestion control protocols are still in the development phase, and little attention is paid to the fact that not all receivers share the same goal as the sender. It has been shown that conformant TCP senders can easily be tricked into sending at a higher rate by modifying the TCP receiver [SCWA99]. Only single-rate multicast protocols with large receiver sets are usually immune since a single receiver that claims to be able to receive at a higher rate than it actually is will simply not contribute to the congestion control process. Before the large-scale deployment of new protocols it is necessary to also investigate the aspect of malicious receivers.

Some reliable multicast protocols build an application-level tree for acknowledgment aggregation. We have devised a hybrid rate-/window-based variant of TFMCC that uses implicit RTT measurement combined with suppression within the aggregation nodes. This variant does not need to perform explicit RTT measurements or end-to-end feedback suppression. Whilst at first glance this would seem to be a big improvement over the variant in Chapter 7, in truth it moves the complex initialization problem from RTT measurement to scalable tree construction, which shares many of the problems posed by RTT measurement. Still, this seems to be a promising additional line of research.

Also, further improvements to the feedback mechanisms used with TFMCC are possible. An important step will be the combination of knowledge about the value distribution within the responder group with implosion avoidance features. Several mechanisms to estimate the size of the receiver set from the feedback time and the number of feedback messages with exponential feedback timers have been proposed [LN00, FT99]. By combining such estimation methods with extremum feedback, it should be possible to estimate the distribution of response values at the receivers in case this distribution is not known. For continuous feedback, this knowledge can then be used to generate feedback mechanisms based on Algorithm 6, where instead of drawing a random number the response values themselves are used to compute a timeout value. In scenarios where the distribution of response values is not uniform, we expect that such an approach will outperform the biasing mechanisms presented in section 6.8.3, which do not take the distribution into account.

Taking these considerations one step further, in some cases the maximum change of the relevant state during one feedback round is bounded. For example, in the case of TFMCC, the measurements to determine round-trip time and loss event rate are subject to smoothing, thus limiting the maximum rate increase and decrease per round-trip time. In case some information about the previous distribution of feedback values is available (e.g., from the previous feedback round), it is possible to infer the worst case distribution of the current feedback round. This allows to further improve the feedback algorithm by tailoring it to the specific distribution.

To fully understand the dynamics of a combination of TFRC and PCC congestion control, such a hybrid protocol has to be implemented and analyzed in a simulation and real-world environment, as has been done with the separate protocols. While in theory PCC can also be used to complement TFMCC when the sending rate falls below the lowest value that is still useful for the application, we have not yet extended PCC to multicast. Nevertheless, with the experience gained from extending TFRC to TFMCC, such an extension should be relatively straightforward.

A further area of research is the improvement of the models for TCP traffic that are used for some of the rate-based congestion control mechanisms. Current TCP formulae are based on several assumptions that are often not met in real-world environments, although it has to be noted that the current formulae usually give fairly accurate TCP rate predictions even when these assumptions do not hold. We are currently investigating a method to perform a more accurate estimate of the fair TCP rate if the loss event rate is measured at a sending rate that differs from the TCP-friendly rate (which is usually the case with PCC).

A similar effect of inaccurate loss rate measurements can be observed when varying the packet size of a congestion controlled flow. TFRC's and TFMCC's loss measurement mechanisms assume that the size of the data packets is either constant or varies to some degree around a mean value but is independent of the sending rate. Applications for which these assumptions are valid vary the sending rate by adjusting the number of packets sent per time interval. For some applications however, these assumptions do not hold. When packets have to be sent at specific points in time or with a fixed packet frequency (which is the case for example for streaming audio) and the sending rate is modified by adjusting the packet size, modifications to the congestion control mechanism are necessary.

When packets are sent that are smaller than the MTU, throughput degrades linearly with the packet size. This is the correct behavior in an environment where the packet rate is the limited resource. In an environment where the bottleneck is bandwidth-limited and flows may use packets of different sizes, resources will not be shared fairly among flows but the share of resources will depend on the respective packet size.

In particular, it is necessary to determine:

- a rate that is fair to competing flows with different packet sizes with respect to the limited resource, and
- how to calculate this rate, given that a different number of packets per time interval over which the loss event rate is measured will introduce a bias in the measured loss event rate.

A less than linear degradation in throughput is only justifiable when the packet size has an impact on the load of the network (i.e., a smaller packet is less costly to send than a large one). The relative cost of smaller sized packets will usually lie somewhere between equal costs for all packets and a linear relation between cost and packet size. Modifications to the loss event rate measurement mechanism must ensure that a flow sending small packets does not achieve a higher throughput than a flow sending larger packets. At the same time, a flow sending small packets should not achieve a much smaller throughput than what is justified given the actual resource used by that flow (in terms of bandwidth *and* per packet overhead).

Finally, the basic equation-based rate controller in TFMCC would also appear to be suitable for use in receiver-driven layered multicast, especially if combined with dynamic layering [BFH+00] to eliminate problems with unpredictable multicast leave latency. We believe that TFMCC's rate control mechanism can be integrated in such a framework in a relatively straightforward manner.

Bibliography

[All00] M. Allman. A web server's view of the transport layer. *ACM Computer Communication Review*, 30(5), October 2000.

[Alm00] K. Almeroth. The evolution of multicast: From the MBone to inter-domain multicast to Internet2 deployment. *IEEE Network Special Issue on Multicasting*, 2000.

[ANS02] A. Adams, J. Nicholas, and W. Siadak. Protocol independent multicast - dense mode (PIM-DM): Protocol specification (revised), February 2002. Internet Draft draft-ietf-pim-dm-new-v2-01.txt, work in progress.

[APS99] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, Internet Engineering Task Force, April 1999. Obsoletes RFC 2001.

[BB01] D. Bansal and H. Balakrishnan. Binomial congestion control algorithms. In *Proc. IEEE INFOCOM 2001 proceedings*, volume 2, pages 631–640, Anchorage, Alaska, April 2001.

[BBC$^+$98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, IETF Network Working Group, 1998.

[BBE$^+$99] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999. revised September 1999, to appear in IEEE Computer.

[BEF$^+$00] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[BFH$^+$00] J. Byers, M. Frumin, G. Horn, M. Luby, M. Mitzenmacher, A. Roetter, and W. Shaver. FLID-DL: Congestion control for layered multicast. In *Proc. Second International Workshop on Networked Group Communication (NGC 2000)*, pages 71–81, Palo Alto, CA, USA, November 2000.

[BG99] A. Basu and J. Golestani. Architectural issues for multicast congestion control. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 1999.

[BLMR98] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. *Proceedings of ACM Sigcomm '98*, September 1998.

[BTK99] S. Bhattacharyya, D. Towsley, and J. Kurose. The loss path multiplicity problem in multicast congestion control. In *Proc. of IEEE Infocom*, volume 2, pages 856 – 863, New York, USA, March 1999.

[BTW94] J.-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. *Proceedings of the ACM SIGCOMM*, pages 58 – 67, September 1994.

[BZB$^+$97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) – version 1 functional specification. RFC 2205, IETF Network Working Group, 1997.

[CJ89] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Journal of Computer Networks and ISDN Systems*, 17(1):1–14, June 1989.

[CM83] J.-M. Chang and N. Maxemchuk. A broadcast protocol for broadcast networks. In *Proceedings of GLOBECOM*, page 19.2, December 1983.

[CM84] J.-M. Chang and N. Maxemchuk. Reliable broadcast protocols. In *ACM Transactions on Computer Systems*, volume 2(3), pages 251–273, August 1984.

[CST00] B. Cain, T. Speakman, and D. Towsley. Generic Router Assist GRA Building Block Motivation and Architecture, March 2000. INTERNET DRAFT draft-ietf-rmt-gra-arch-01.txt, Work in Progress.

[CT90] D. D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. ACM SIGCOMM*, September 1990.

[DA01] M. J. Donahoo and S. R. Ainapure. Scalable multicast representative member selection. In *IEEE INFOCOM*, March 2001.

[Dam01] J. P. Damm. Probabilistic congestion control for non-adaptable flows. Master's thesis, University of Mannheim, April 2001.

[DC90] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems (TOCS)*, 8(2):85–110, 1990.

[DDI$^+$99] E. Duros, W. Dabbous, H. Izumiyama, N. Fujii, and Y. Zhang. A link layer tunneling mechanism for unidirectional links. *draft-ietf-udlr-lltunnel-02.txt*, June 1999.

[DEF$^+$96] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. gung Liu, and L. Wei. The PIM architecture for wide-area multicast routing. *IEEE/ACM Transactions on Networking*, pages 153 – 162, April 1996.

[Den03] R. Denda. *Fairness in Computer Networks*. PhD thesis, Praktische Informatik IV, University of Mannheim, 2003. in preparation.

[DG99] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. *IEEE Networks magazine*, 13(4), July/August 1999.

[DLL$^+$00] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network Special Issue on Multicasting*, January/February 2000.

[Fen97] W. Fenner. Internet group management protocol, version 2. RFC 2236, November 1997.

[FF96] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[FF99] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, August 1999.

[FHHK00] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. Protocol independent multicast - sparse mode (PIM-SM): Protocol specification, November 2000. Internet draft draft-ietf-pim-sm-v2-new-01.txt, work in progress.

[FHHK02] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. Protocol independent multicast - sparse mode (PIM-SM): Protocol specification (revised), March 2002. Internet Draft draft-ietf-pim-sm-v2-new-05.txt, work in progress.

[FHP00] S. Floyd, M. Handley, and J. Padhye. A comparison of equation-based and AIMD congestion control, February 2000. URL http://www.aciri.org/tfrc/.

[FHPW00a] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proc. ACM SIGCOMM*, pages 43 – 56, Stockholm, Sweden, August 2000.

[FHPW00b] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications: the extended version. Technical report, Technical Report TR-00-003, International Computer Science Institute (ICSI), March 2000.

[FJ93] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[FJL$^+$97] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784 – 803, December 1997.

[FT99] T. Friedman and D. Towsley. Multicast session membership size estimation. In *IEEE Infocom*, New York, NY, March 1999.

[Fuh00] T. T. Fuhrmann. Protocol independent multicast and asymmetric routing. Technical Report 1/2000, Praktische Informatik IV, University of Mannheim, February 2000.

[FW01] T. Fuhrmann and J. Widmer. On the scaling of feedback algorithms for very large multicast groups. *Computer Communications*, 24(5-6):539 – 547, March 2001.

[GK99] R. Gibbens and F. Kelly. Distributed connection acceptance control for a connectionless network. In *Proc. 16th International Teletraffic Congress*, pages 941–952, Edinburgh, Scotland, June 1999.

[Gro97] M. Grossglauser. Optimal deterministic timeouts for reliable scalable multicast. *IEEE Journal on Selected Areas in Communications*, 15(3):422–433, April 1997.

[Gup60] S. S. Gupta. Order statistics from the gamma distribution. *Technometrics*, 2:243 – 262, 1960.

[Han98] M. Handley. Session directories and scalable Internet multicast address allocation. In *Proc. ACM Sigcomm*, pages 105 – 116, Vancouver, B.C., Canada, September 1998.

[HFGC98] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar. RTP payload format for MPEG1/MPEG2 video. *RFC 2250*, January 1998.

[HJ98] M. Handley and V. Jacobson. SDP: Session description protocol. RFC 2327, April 1998.

[Hof96] M. Hofmann. A generic concept for large-scale multicast. In *Proceedings of the International Zurich Seminar on Digital Communication, LNCS*, volume 1044, pages 95–106, February 1996.

[Höt02] J. Hötzel. VBR video coding and transmission using a single-rate multicast transport protocol. Master's thesis, University of Mannheim, April 2002. in German.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session initiation protocol. RFC 2543, March 1999.

[Jac88] V. Jacobsen. Congestion avoidance and control. *Proceedings of ACM Sigcomm*, 1988.

[JE96] S. Jacobs and A. Eleftheriadis. Providing video services over networks without quality of service guarantees. *World Wide Web Consortium Workshop on Real-Time Multimedia and the Web*, October 1996.

[JW89] J. Y. Juang and B. W. Wah. A unified minimum-search method for resolving contentions in multiaccess networks with ternary feedback. *Information Sciences*, 48(3):253–287, 1989. Elsevier Science Pub. Co., Inc., New York, NY.

[Kel01] T. Kelly. An ECN probe-based connection acceptance control. *Computer Communication Review*, 31(3), July 2001.

[KHR02] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for high bandwidth-delay product networks. In *Proc. ACM SIGCOMM*, Pittsburgh, August 2002.

[KKZ00] F. Kelly, P. Key, and S. Zachary. Distributed admission control. *IEEE Journal on Selected Areas in Communications*, 18(12):2617–2628, December 2000.

[Kuh01] C. Kuhmünch. *Videoskalierung und Integration interaktiver Elemente in Teleteaching Szenarien*. PhD thesis, University of Mannheim, May 2001. in German.

[LN00] C. Liu and J. Nonnenmacher. Broadcast audience estimation. In *IEEE Infocom*, pages 952–960, Tel Aviv, Israel, March 2000.

[MC00] S. McCreary and K. Claffy. Trends in wide area IP traffic patterns: A view from AMES internet exchange. *Proc. 13th ITC Specialist Seminar*, pages 1–11, September 2000.

[MF97] J. Mahdavi and S. Floyd. TCP-friendly unicast rate-based flow control. Note sent to the end2end-interest mailing list, January 1997.

[MHKE01] M. Mauve, V. Hilt, C. Kuhmnch, and W. Effelsberg. RTP/I - toward a common application level protocol for distributed interactive media. *IEEE Transactions on Multimedia*, 3(1), March 2001.

[MJ93] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, San Diego, CA, January 1993.

[MJV96] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proc. of ACM SIGCOMM*, pages 117 – 130, Palo Alto, CA, USA, August 1996.

[Moy94] J. Moy. Multicast extensions to OSPF. RFC 1584, March 1994.

[MRBP98] A. Mankin, A. Romanow, S. Bradner, and V. Paxson. RFC 2357: IETF criteria for evaluating reliable multicast transport and application protocols, June 1998. Obsoletes RFC1650. Status: INFORMATIONAL.

[MSMO97] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the congestion avoidance algorithm. *Computer Communications Review,* 1997.

[MTH97] D. L. Mills, A. Thyagarajan, and B. C. Huffman. Internet timekeeping around the globe. *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting,* pages 365 – 371, December 1997.

[NB99] J. Nonnenmacher and E. W. Biersack. Scalable feedback for large groups. *IEEE/ACM Transactions on Networking*, 7(3):375 – 386, June 1999.

[OKM96] T. Ott, J. Kemperman, and M. Mathis. The stationary behavior of ideal TCP congestion avoidance. 1996.

[Pad00] J. Padhye. *Towards a Comprehensive Congestion Control Framework for Continuous Media Flows in Best Effort Networks*. PhD thesis, Ph.D. thesis, University of Massachusetts at Amherst, March 2000.

[PFTK98] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. *Proceedings of ACM Sigcomm*, 1998.

[PFTK00] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2):133–145, April 2000.

[PKT99] J. Padhye, D. Kurose, and R. Towsley. A model based TCP-friendly rate control protocol. *Proc. International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 1999.

[Pos81] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, 1981.

[PSLB97] S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (rmtp). In *IEEE Journal on Selected Areas in Communications*, volume 15(3), pages 407–421, April 1997.

[RBR99] I. Rhee, N. Balaguru, and G. Rouskas. MTCP: scalable TCP-like congestion control for reliable multicast. In *Proc. of IEEE INFOCOM*, volume 3, pages 1265 – 1273, March 1999.

[RF99] K. K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to ip. *RFC 2481*, January 1999.

[RHE99] R. Rejaie, M. Handley, and D. Estrin. Rap: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. *Proc. IEEE Infocom*, March 1999.

[Riz98] L. Rizzo. Dummynet and forward error correction. In *Proc. Freenix 98*, 1998.

[Riz00] L. Rizzo. pgmcc: A TCP-friendly single-rate multicast congestion control scheme. In *Proc. ACM SIGCOMM*, pages 17 – 28, Stockholm, Sweden, August 2000.

[ROY00] I. Rhee, V. Ozdemir, and Y. Yi. TEAR: TCP emulation at receivers - flow control for multimedia streaming. Technical report, Department of Computer Science, North Carolina State University, April 2000.

[RR99] S. Ramesh and I. Rhee. Issues in TCP model-based flow control. Technical Report TR-99-15, Department of Computer Science, NCSU, 1999.

[RS98] J. Rosenberg and H. Schulzrinne. Timer reconsideration for enhanced RTP scalability. In *Proc. of IEEE INFOCOM*, 1998.

[SC01] H. Schulzrinne and S. Casner. RTP profile for audio and video conferences with minimal control, November 2001. Internet draft draft-ietf-avt-profile-new-12.txt, work in progress.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. *RFC 1889*, January 1996.

[SCFJ01] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, November 2001. Internet draft draft-ietf-avt-rtp-new-11.txt, work in progress.

[SCG$^+$01] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano. PGM reliable transport protocol specification. RFC 3208, December 2001.

[SCWA99] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *ACM Computer Communications Review*, 29(5):71–78, October 1999.

[SEFJ97] P. Sharma, D. Estrin, S. Floyd, and V. Jacobson. Scalable timers for soft state protocols. In *Proceedings of IEEE INFOCOM*, April 1997.

[SRC84] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, pages 277–288, November 1984.

[Ste94] R. Stevens. *TCP/IP Illustrated, Volume 1. The Protocols*. Addison-Wesley Publishing Company, 1994.

[SW00] D. Sisalem and A. Wolisz. LDA+ TCP-friendly adaptation: A measurement and comparison study. *Proc. International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 2000.

[TZ99a] D. Tan and A. Zakhor. Real-time internet video using error resilient scalable compression and TCP-friendly transport protocol. *IEEE Transactions on Multimedia*, May 1999.

[TZ99b] W. Tan and A. Zakhor. Error control for video multicast using hierarchical FEC. *Proc. International Conference on Image Processing*, October 1999.

[VCR98] L. Vicisano, J. Crowcroft, and L. Rizzo. TCP-like congestion control for layered multicast data transfer. In *Proc. of IEEE INFOCOM*, volume 3, pages 996 – 1003, March 1998.

[Vog99] J. Vogel. Entwurf und Implementierung eines generischen Late Join Mechanismus für interaktive Medien. Master's thesis, Praktische Informatik IV, University of Mannheim, November 1999. (in German).

[WDM01] J. Widmer, R. Denda, and M. Mauve. A survey on TCP-friendly congestion control. *Special Issue of the IEEE Network Magazine "Control of Best Effort Traffic"*, 15(3):28–37, May/June 2001.

[WF01] J. Widmer and T. Fuhrmann. Extremum feedback for very large multicast groups. In *Proc. Third International Workshop on Networked Group Communication (NGC)*, London, GB, November 2001.

[WH01a] J. Widmer and M. Handley. Extending equation-based congestion control to multicast applications. In *Proc. ACM SIGCOMM*, pages 275 – 286, San Diego, CA, August 2001.

[WH01b] J. Widmer and M. Handley. Extending equation-based congestion control to multicast applications. Technical Report TR 13-2001, Praktische Informatik IV, University of Mannheim, Germany, May 2001.

[WH01c] J. Widmer and M. Handley. TCP-friendly multicast congestion control (TFMCC): Protocol specification, November 2001. INTERNET DRAFT draft-ietf-rmt-bb-tfmcc-00.txt, Work in Progress.

[Wid00] J. Widmer. Equation-based congestion control. Master's thesis, ACIRI / University of Mannheim, February 2000.

[WJ85] B. W. Wah and J. Y. Juang. Resource scheduling for local computer systems with a multiaccess network. *IEEE Trans. on Computers*, C-34(12):1144–1157, December 1985.

[WMD02] J. Widmer, M. Mauve, and J. P. Damm. Probabilistic congestion control for non-adaptable flows. In *Proc. 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Miami, FL, May 2002.

[WMK94] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems, Lecture Notes in Computer Science 938*. Springer-Verlag, 1994.

[WPD88] D. Waitzman, C. Partridge, and S. Deering. Distance vector multicast routing protocol. RFC 1075, November 1988.

[WS98] H. Wang and M. Schwartz. Achieving bounded fairness for multicast and tcp traffic in the internet. *Proc. of ACM SIGCOMM*, 1998.

[YL00] Y. R. Yang and S. S. Lam. General aimd congestion control. In *Proc. IEEE ICNP 2000*, Osaka, Japan, November 2000.

Appendix A

Analysis of Exponential Feedback Suppression

A.1 Proof of Equation (6.6)

To derive the expected values for the earliest response, we use the probability distribution for the least value of x chosen by the group of potential responders. As derived above the probability that $x_{min} = \min\{x_1, \ldots, x_n\} \in [x, x + dx]$ is $n(1-x)^{n-1}dx$.

The resulting value for the expected feedback latency is:

$$\begin{aligned} E[D] &= T \int_{N^{-1}}^{1} n(1-x)^{n-1}(1 + \log_N x)dx \\ &= \frac{T}{\ln N} \int_{1/N}^{1} \frac{(1-x)^n}{x} dx \end{aligned}$$

We will show that

$$\int_{1/N}^{1} \frac{(1-x)^n}{x} \, dx < \ln N - \ln n - C + \frac{n}{N}$$

and therefore

$$E[D] \simeq T(1 - \log_N n)$$

We first prove

$$-\sum_{k=1}^{n} \binom{n}{k} \frac{(-1)^k}{k} = \sum_{k=1}^{n} \frac{1}{k} \tag{A.1}$$

using $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, $\binom{n-1}{k-1}\frac{1}{k} = \frac{1}{n}\binom{n}{k}$, and complete induction. Clearly Equation (A.1) holds for $n = 1$. Then

$$\begin{aligned}
&-\sum_{k=1}^{n} \binom{n}{k} \frac{(-1)^k}{k} \\
&= -\sum_{k=1}^{n-1} \binom{n-1}{k} \frac{(-1)^k}{k} - \sum_{k=1}^{n-1} \binom{n-1}{k-1} \frac{(-1)^k}{k} - \frac{(-1)^n}{n} \\
&= \sum_{k=1}^{n-1} \frac{1}{k} - \frac{1}{n} \sum_{k=1}^{n-1} \binom{n}{k} (-1)^k - \frac{(-1)^n}{n} \\
&= \sum_{k=1}^{n-1} \frac{1}{k} + \frac{1}{n} - \frac{1}{n} \sum_{k=0}^{n} \binom{n}{k} (-1)^k \\
&= \sum_{k=1}^{n} \frac{1}{k}
\end{aligned}$$

This result is approximated from below by $\ln n + C$ where $C = 0.577\ldots$ is the Euler constant.

Thus, we have

$$\begin{aligned}
&\int_{1/N}^{1} \frac{(1-x)^n}{x}\, dx \\
&= -\int_{1/N}^{1} \sum_{k=0}^{n} \binom{n}{k} (-x)^{k-1}\, dx \\
&= \left[\ln x + \sum_{k=1}^{n} \binom{n}{k} \frac{(-x)^k}{k} \right]_{1/N}^{1} \\
&= \ln N + \sum_{k=1}^{n} \binom{n}{k} \frac{(-1)^k}{k} - \sum_{k=1}^{n} \binom{n}{k} \frac{(-1/N)^k}{k} \\
&< \ln N - \ln n - C + \frac{n}{N} - \sum_{k=2}^{n} \binom{n}{k} \frac{(-1/N)^k}{k} \\
&< \ln N - \ln n - C + \frac{n}{N}
\end{aligned}$$

A.2 Proof of Equation (6.7)

Assume that x is the smallest value chosen in the group. If $x \leq N^{-1}$ a response will be immediately sent. However, due to the network's latency τ duplicate responses will be received from all members that chose their value x_i in the interval $[x, N^{\tau/T-1})$.

If $x > N^{-1}$ the earliest response will be sent at a time $t > 0$. Duplicate responses will then be received from all members that chose their value x_i in the interval $[N^{t/T-1}, N^{\frac{t+\tau}{T}-1})$. Using $x = N^{t/T-1}$ this interval can be written as $[x, xN^{\tau/T})$.

Under the condition that all responses after the first response are distributed equally in the interval $[x, 1)$ we find the following expected values for duplicate responses in these two cases

$$(n-1)\frac{N^{\tau/T-1} - x}{1-x}$$

and

$$(n-1)\frac{xN^{\tau/T} - x}{1-x}$$

If the earliest response is sent after $t = T - \tau$ no suppression can take place any more. Clearly, the probability for this case is $(1 - N^{-\tau/T})^n$. Altogether we find

$$\begin{aligned}
E[M] &= \int_0^{1/N} (n-1)\frac{N^{\tau/T-1} - x}{1-x} \cdot n(1-x)^{n-1}\, dx && \text{(A.2)}\\
&\quad + \int_{1/N}^{N^{-\tau/T}} (n-1)x\frac{N^{\tau/T} - 1}{1-x} \cdot n(1-x)^{n-1}\, dx && \text{(A.3)}\\
&\quad +(n-1)(1 - N^{-\tau/T})^n && \text{(A.4)}\\
&= N^{\tau/T}\left(\frac{n}{N} + \left(1 - \frac{1}{N}\right)^n - \left(1 - \frac{1}{N^{\tau/T}}\right)^n\right) && \text{(A.5)}
\end{aligned}$$

Appendix B

TFMCC

B.1 Scheduling of Packet Transmissions

As TFMCC is rate-based, and as operating systems typically cannot schedule events precisely, it is necessary to be opportunistic about sending data packets so that the correct average rate is maintained despite the coarse-grain or irregular scheduling of the operating system. Thus a typical sending loop will calculate the correct inter- packet interval, t_{ipi}, as follows:

$$t_{ipi} = s/R_{send}$$

When a sender first starts sending at time t_0, it calculates t_{ipi}, and calculates a nominal send time $t_1 = t_0 + t_{ipi}$ for packet 1. When the application becomes idle, it checks the current time, t_{now}, and then requests re-scheduling after $(t_{ipi} - (t_{now} - t_0))$ seconds. When the application is re-scheduled, it checks the current time, t_{now}, again. If $(t_now > t_1 - \delta)$ then packet 1 is sent (see below for δ).

Now a new t_{ipi} may be calculated, and used to calculate a nominal send time t_2 for packet 2: $t2 = t_1 + t_{ipi}$. The process then repeats, with each successive packet's send time being calculated from the nominal send time of the previous packet.

In some cases, when the nominal send time, t_i, of the next packet is calculated, it may already be the case that $t_{now} > t_i - \delta$. In such a case the packet should be sent immediately. Thus if the operating system has coarse timer granularity and the transmit rate is high, then TFMCC may send short bursts of several packets separated by intervals of the OS timer granularity.

The parameter δ is to allow a degree of flexibility in the send time of a packet. If the operating system has a scheduling timer granularity of t_{gran} seconds, then delta would typically be set to:

$$\delta = \min(t_{ipi}/2, t_{gran}/2)$$

t_{gran} is 10 milliseconds on many Unix systems. If t_{gran} is not known, a value of 10 milliseconds can be safely assumed.

B.2 Architecture of *libtfmcc*

Here, we will give an overview of the class structure of the TFMCC library. Further details can be found in [Höt02].

B.2.1 UDP Sockets

The sender as well as the receivers have to create an instance of the `Udp` class in order to transmit TFMCC packets.

```
class Udp {
public:
  Udp(string bind_addr = "0.0.0.0",
      string to_addr = "0.0.0.0", int port = 12345);
  bool set_ttl(int ttl);
  bool addmembership(string group);
  bool dropmembership(string group);
  ...
};
```

The constructor `Udp` creates a UDP socket and the parameter `bind_addr` specifies which network interface is to be used for the transmission of datagrams.[1] For the TFMCC sender, `to_addr` specifies the multicast group to which to send the data packets whereas for the TFMCC receiver it specifies the address to send the receiver reports to (i.e., the IP address of the sender). The methods `addmembership` and `dropmembership` are used by the receiver to join and

[1]The IP address 0.0.0.0 is not assigned to a network interface but can be used to receive data packets via all network interfaces.

leave the multicast group. The method `set_ttl` can be used to set the TTL scope of the UDP datagrams.

B.2.2 Datagrams

libtfmcc uses the helper class `Data` for the internal management of data packets:

```
class Data {
public:
  Data(char* source,  size_t size);
  ~Data();
  unsigned char* data;
  size_t size;
  ...
};
```

The constructor `Data` allocates `size` bytes of memory and the array `source` is copied to the newly allocated memory. This copy can now be accessed via `data`. The memory is deallocated by the constructor `~Data`.

B.2.3 Sender Functionality

TFMCC sender functionality is provided by the class `Tfmcc_Sender`:

```
class Tfmcc_Sender {
public:
  Tfmcc_Sender(Udp& udp, int p_size);
  void one_loop(Tfmcc_Application_Sender* tas);
  void main_loop(Tfmcc_Application_Sender* tas);
  ...
};
```

The constructor `Tfmcc_Sender` requires a UDP socket `Udp` for the transmission of data packets. Furthermore, the packet size `p_size` needs to be specified by the application.[2] Single data

[2]*Path-MTU discovery*, a mechanism to automatically determine the maximum transmission unit of an Internet path, is not available for multicast transport.

packets can be sent via the method `one_loop`. The method `main_loop` hands the program control over to the class `Tfmcc_Sender` which continuously calls `one_loop` to transmit a stream of datagrams. `main_loop` does not terminate even when the last receiver quits the multicast session as new receivers may join the multicast session any time.

Application Interface

An application that wants to send data via the TFMCC protocol needs to implement the abstract class `Tfmcc_Application_Sender`:

```
class Tfmcc_Application_Sender {
  Tfmcc_Application_Sender();
  virtual void set_rate(int rate) = 0;
  virtual Data read() = 0;
  ...
};
```

A `Tfmcc_Sender` instance calls the method `read` in regular intervals. An application has to implement this method to provide data packets of the type `Data`. Whenever the sending rate changes, an object of the class `Tfmcc_Sender` calls the `set_rate` method. A TFMCC application implementing this method should limit its sending rate to `rate` bytes/s, otherwise data packets have to be discarded.

B.2.4 Receiver Functionality

The `Tfmcc_Sink` class implements the receivers side of the TFMCC protocol and has to be instantiated by an application that wants to receive data via TFMCC:

```
class Tfmcc_Sink {
public:
  Tfmcc_Sink(Udp& udp);
  void one_loop(Tfmcc_Application_Sink& application);
  void main_loop(Tfmcc_Application_Sink& application);
  bool receiver_leave;
  ...
}
```

The constructor `Tfmcc_Sink` requires the parameter `Udp`, the UDP socket over which to receive the datagrams. Similar to the sender side, single data packets can be received with the method `one_loop` and `main_loop` hands over control to the class `Tfmcc_Sink` which in turn calls `one_loop` whenever new packets arrive. Even when no more packets are received `Tfmcc_Sink::main_loop` does not terminate since a new sender could start sending on this multicast address any time. If a receiver decides to leave the multicast session, it sets `receiver_leave` to `true` and calls `one_loop` once more. This allows the sender to immediately switch to a new CLR in case the receiver was the CLR.

Application Interface

The abstract class `Tfmcc_Application_Sink` needs to be implemented by an application that wants to receive data via the TFMCC protocol.

```
class Tfmcc_Application_Sink {
public:
  Tfmcc_Application_Sink();
  virtual void write(Data data) = 0;
  ...
};
```

An object of the class `Tfmcc_Sink` regularly calls the method `write`. This method hands data packets over to the application (without buffering). If necessary, buffering has to be implemented by the application.

B.3 Structure of the TFMCC Data Packet Header

A TFMCC data packet header (see Figure B.1) contains the following fields:

sequence_number (32 bit integer): This number is incremented by one for each data packet transmitted. The field must be sufficiently large that it does not wrap causing two different packets with the same sequence number to be in the receiver's recent packet history at the same time.

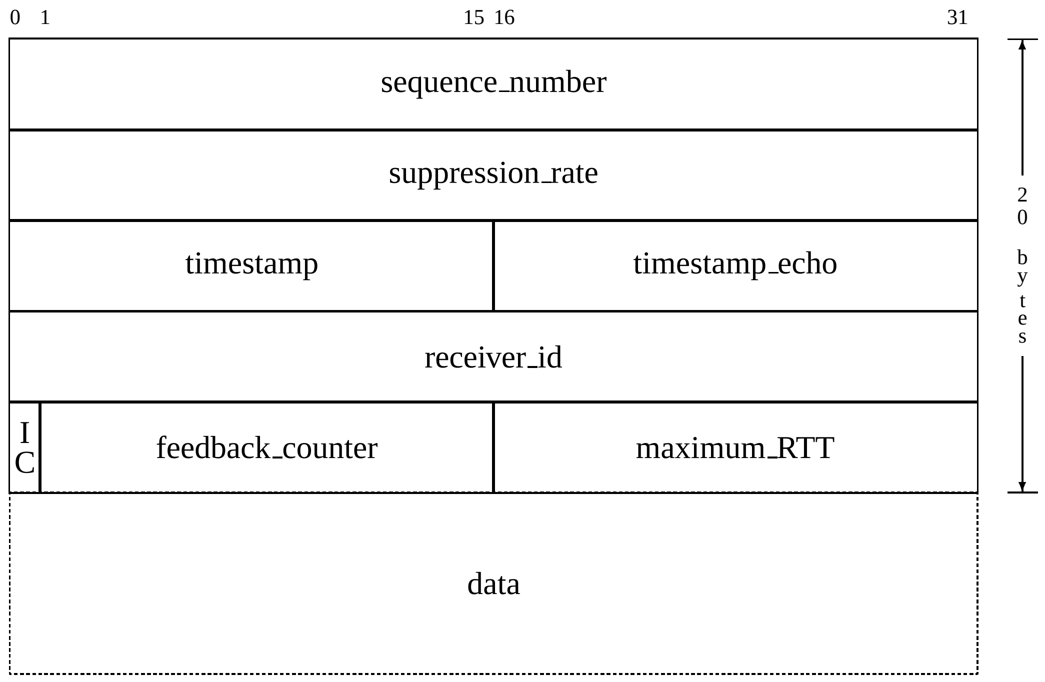


Figure B.1: Structure of the TFMCC data packet header

suppression_rate (32 bit integer): Only receivers with a calculated rate lower than the suppression rate (measured in bytes/s) are eligible to give feedback, unless their RTT is higher than the maximum RTT described below in which case they are also eligible to give feedback.

timestamp (16 bit integer): A timestamp (in milliseconds) indicating when the packet is sent. The timestamp is used by the receiver for the one-way delay adjustments and is also echoed in the receiver's feedback packets for the sender-based calculation of the maximum RTT.

timestamp_echo (16 bit integer): The sender copies the timestamp of the last report from receiver with ID `receiver_id` into `timestamp_echo`. The timestamp echo is used for the receiver's RTT calculation.

receiver_id (32 bit integer): The field stores ID of the receiver whose timestamp is included in the timestamp echo field.

IC (1 bit): The `IC` flag is set in case the receiver with ID `receiver_id` is the CLR.

feedback_counter (15 bit integer): The number of the current feedback round is transmitted to indicate the beginning of a new round to the receivers. At the beginning of a new round receiver reports for older rounds are discarded.

maximum_RTT (16 bit integer): The field contains the largest RTT (in milliseconds) known to the sender. A receiver uses this value for the RTT in case it has not yet measured its own RTT.

B.4 Structure of the TFMCC Control Packet Header

A TFMCC control packet header (see Figure B.2) contains the following fields:

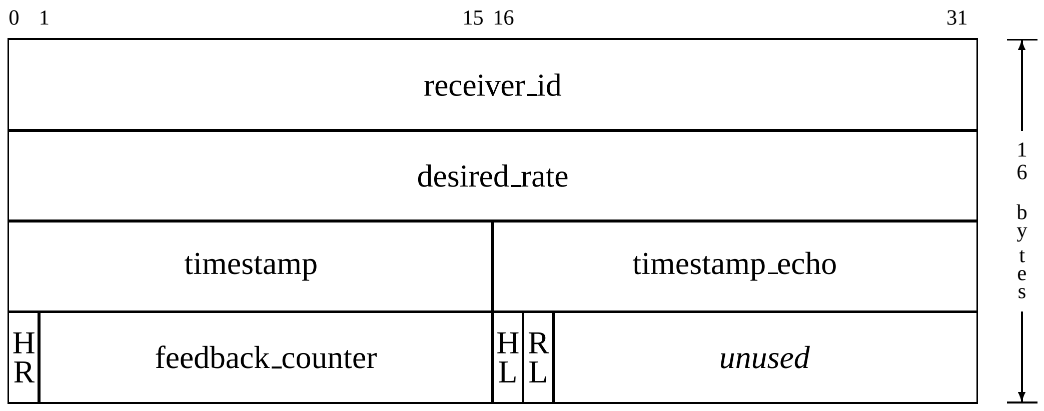


Figure B.2: Structure of the TFMCC control packet header

receiver_id (32 bit integer): The field contains the ID of the receiver sending the control packet. Generation of unique receiver IDs is beyond the scope of the library. However, for most purposes a combination of IP address and port will suffice.

desired_rate (32 bit integer): The desired rate (in bytes/s) specifies the TCP-friendly rate calculated by the receiver.

timestamp (16 bit integer): A timestamp (in milliseconds) indicating when the control packet is sent. It may be echoed by the sender to allow the receiver to measure its current RTT.

timestamp_echo (16 bit integer): The field contains a copy of the timestamp of the last data packet from the sender.

HR: (1 bit): Receivers without any previous RTT measurements have to set the HR flag. The sender preferentially echoes timestamps of control packets with this flag set to allow all receivers to measure their RTT at least once.

feedback_counter (15 bit integer): The feedback counter indicates the feedback round valid when the control packet is sent.

HL (1 bit): A receiver sets the `HL` flag after the first packet loss. As soon as the sender receives the first control packet with this flag set, slow-start is terminated.

RL (1 bit): If a receiver wishes to leave the multicast session, it can set the `RL` flag to prevent it from being selected as CLR. If the receiver is the CLR, it should send at least one control packet with the `RL` flag set before leaving the group to allow the sender to select a new CLR.