Performance Analysis of the Datagram Congestion Control Protocol DCCP for Real-Time Streaming Media Applications

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Samuel C. Jero

August 2013

This thesis titled

Performance Analysis of the Datagram Congestion Control Protocol DCCP for Real-Time Streaming Media Applications

by

SAMUEL C. JERO

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Shawn D. Ostermann

Associate Professor of Computer Science

Dennis Irwin

Dean, Russ College of Engineering and Technology

Abstract

JERO, SAMUEL C., M.S., August 2013, Computer Science

Performance Analysis of the Datagram Congestion Control Protocol DCCP for Real-Time Streaming Media Applications (180 pp.)

Director of Thesis: Shawn D. Ostermann

The growth of real-time, streaming media application traffic in the Internet presents a number of challenges because the real-time constraints and interactive nature of these applications render the use of TCP ineffective. These streaming media application flows are usually high bandwidth and long duration, which means they should utilize network congestion control to avoid congestion collapse and ensure fairness. The Internet Engineering Task Force (IETF) developed the Datagram Congestion Control Protocol (DCCP) to provide congestion control for these types of real-time applications. A major factor behind this effort was the desire to eliminate the duplication of effort and potential for error resulting from each application implementing its own congestion control.

In this research, we examine the difference in network performance and video quality for a typical application, a high bandwidth video telephony client, when using DCCP instead of the default transport protocol. We discuss several challenges to porting an application to DCCP and then examine the impact of DCCP on network and application performance in both testbed and Internet environments. We show that DCCP responds to changing network conditions within a few round trip times and provides better fairness to other network traffic than typical real-time, streaming media congestion control methods. Given fair bandwidth allocation, DCCP provides equivalent or better video quality, as measured by Peak Signal to Noise Ratio (PSNR) and Structural Similarity (SSIM).

Soli Deo gloria

Acknowledgments

I would like to thank my advisor, Dr. Shawn Ostermann, for his guidance in this thesis work and suggestions when things became difficult. The opportunity to work in his Internetworking Research Group the past three years, providing me with invaluable experience and financial support, has also been greatly appreciated.

Dr. Hans Kruse, my unofficial second advisor, deserves a special thanks for his advice and support. His knowledge of VoIP and media streaming was invaluable in getting started in this research.

Thanks also to my labmates, James Swaro, Zack Sims, and Joshua Schendel, for their help and support. It has been great having someone to bounce ideas off of and discuss research issues with.

A final thanks to my family and friends for their encouragement and support and for providing me an escape from my thesis in times of frustration or after untold hours of work. Our conversations about topics totally unrelated to this thesis have been incredibly valuable in enabling me to keep going and complete this work.

TABLE OF CONTENTS

List of Tables

List of Figures

1 Introduction

1.1 Real-time Streaming Media Applications

The usage of real-time, interactive, streaming media applications, such as VoIP, videoconferencing, telepresence, and remote musical collaboration, over the global Internet is growing. This growth presents a unique challenge to the stability of the Internet because of the particular characteristics of these applications, the most distinctive of which is their real-time interactivity.

Most of these applications are used for two way communication between people, either for conversations or other forms of collaboration. Research has shown that latencies of 150ms are annoying for normal conversation and latencies above 300ms become intolerable [Per99]. Musical collaboration has even tighter latency requirements [Con12]. These requirements place severe constraints on these systems in general, affecting everything from the choice of audio/video codec to the amount of data that can be buffered at the receiver. Further, these low latency requirements make recovery of lost data by retransmission ineffective in many cases. Packets in these applications usually comprise timeslices of the content. Once one timeslice has been played, the next one is needed immediately. If it is unavailable, playback pauses until the next play-able timeslice. By the time a packet is sent, its loss detected, a request for retransmission sent, and the packet resent, the time for that packet to be played will have passed in all but the lowest-latency networks.

Another important characteristic of these applications is that they use a significant amount of bandwidth. This will only increase as high definition video becomes more common. Skype, a common VoIP/videoconferencing application, recommends a connection speed of 500kbits/sec to 1.5Mbits/sec for video calls [Sky13]. Another VoIP/videoconferencing application, linphone, uses 4-8Mbits/sec when transmitting HD video. Cisco's TelePresence system requires 1-4Mbits/sec per video stream, depending on quality [Cis13]. Remote musical collaboration applications often demand even more bandwidth because they usually dispense with complex audio and video compression in order to reduce latency. The LOLA system requires between 100Mbits/sec and 500Mbits/sec depending on configuration [Con12]. Such high bandwidth connections are particularly likely to saturate network links, causing problems both for themselves and for other connections.

The length of typical connections established by real-time, streaming media applications is also an important factor. Unlike the short connections used by email and web page requests, these applications usually send a continuous stream of data that lasts for minutes or hours. Given the high bandwidth requirements of these applications and their long duration flows, they are particularly likely to overload network links for long periods of time, causing significant problems for other users of the network.

A final important feature of these applications is their bursty data streams. These streams are bursty because they by transmit data in large chunks, or bursts, at regular intervals. Video, for example, is transmitted in frames that are spaced between 30 and 40ms apart, depending on the frame rate. Audio samples are similarly grouped together and sent in small chunks every 10-40ms [WW08]. This is in sharp contrast to web pages, emails, and file transfers where the entire transfer is available at once to be sent as fast as possible. These regular bursts of data, particularly bursts of numerous packets resulting from large video frames, can play havoc on router queues, causing overflow immediately after a burst even if the network may be able to handle the application's average throughput.

In addition, the sizes of these chunks can vary significantly, introducing a second level of burstiness. This is particularly true for video because most compression algorithms utilize several different types of frames with sizes that can vary by a factor of four or more [Int02a]. Even within a single frame type, the size can vary dramatically depending on the complexity of the video frame being encoded. The same is true for audio data [Int02c].

Interestingly, augmented reality systems and online gaming share many of these same characteristics, especially the real-time, interactive nature of the data being sent. Such data streams are also likely to be bursty and long duration. If video or texture data is being transmitted, high bandwidth is also likely. Although this work does not directly address such systems, many of the ideas discussed here may also be relevant for these areas.

As a result of the characteristics of real-time, streaming media applications mentioned above, many of the existing networking protocols perform inadequately for these applications. To understand why this is the case, one needs to understand the concept of network congestion.

1.2 Congestion Collapse and Fairness

Computer networks are subject to a phenomenon known as network congestion that occurs when computer networks become over-subscribed, causing router queues to fill up and packets to be dropped. Adding additional traffic to the network at this point does not result in an increase in overall throughput and may, in fact, result in a decrease.

If this condition is not corrected, congestion collapse, where the network spends most of its time sending data that will later be discarded, can occur. This results in drastic throughput reductions in the affected network. When this occurred on the Internet in 1986, throughput was reduced from 32Kbits/sec to 40bits/sec [Jac88], a decrease of three orders of magnitude.

Jacobson [Jac88] observed this issue and introduced several new algorithms into TCP to recognize congestion and slow down appropriately. These algorithms continue to form the basis of TCP's congestion control scheme to this day. We will discuss these algorithms, in their modern form, in chapter 2.

These improvements to TCP eliminated the congestion problems on the Internet at the time. However, the presence of non-congestion controlled traffic (i.e. traffic that is not responsive to indications of congestion) on the Internet continues to be a matter of concern [Flo00, FF99, FHK06b]. Were such traffic flows to make up a significant fraction of traffic on the Internet, congestion collapse could easily occur.

Another closely related issue is fairness. This is the idea that applications should share the available network bandwidth fairly. No application should be able to monopolize the network to the detriment of others. Because TCP is the dominant protocol in today's Internet, fairness with TCP is particularly important [FF99]. In general, the Internet community has understood reasonable fairness between two flows as being able to maintain throughput rates that are within a factor of two of each other [WH06, FHPW08].

Unfairness between congestion controlled and non-congestion controlled traffic is a major issue. When congestion controlled applications compete with non-congestion controlled applications, the non-congestion controlled applications often come to dominate because the congestion controlled applications will slow down in response to congestion, effectively ceding bandwidth to the non-congestion controlled flows [Flo00].

Real-time, streaming media flows are both high bandwidth and long duration, and, therefore, it is particularly important for them to be responsive to congestion and reasonably fair with other traffic. The high bandwidth demands of these applications make them particularly likely to overload the network causing congestion, possibly to the point of congestion collapse, or to steal bandwidth from congestion controlled flows causing significant unfairness. While this behavior might be tolerable for very short data flows, typical real-time, streaming media application flows last for minutes or hours.

In addition, the bursty behavior of these flows can lead to more undesirable behaviors. Bursty flows can induce packet loss at routers by temporarily filling the router queue with a burst of packets. Other network traffic arriving immediately following such a burst will then experience loss and, if congestion controlled, will slow down. In reality, however, the network is not congested, and the burst of packets will drain from the router queue in a few milliseconds, leaving unutilized network bandwidth.

Designing and implementing congestion control algorithms is difficult, and there is a long history of bad designs and incorrect implementations [Flo00, FHK06b]. For that reason, a networking protocol for use with real-time, streaming media applications would ideally provide built-in congestion control so that the application programer is not required to design their own scheme.

We now consider several common networking protocols and examine their suitability for real-time, streaming media applications.

1.3 The Transmission Control Protocol

The Transmission Control Protocol (TCP) was defined in 1981 as RFC 793 [Pos81] and has become by far the most popular networking protocol on the Internet today, underlying the web, email, instant messaging, and file transfer, among other applications. TCP offers applications reliable, in-order transmission of a byte-stream from one endpoint to the other.

In order to accomplish these goals, TCP numbers each byte of data and includes a sequence number on each packet indicating the position of the first byte of this packet in the data stream. Also present in each packet is an acknowledgement number that indicates the next byte of data expected from the other side [Pos81]. These sequence numbers allow TCP to determine what data has been lost in order to be able to retransmit it. They also enable the data to be presented to the receiving application in the same order it was sent, even in the presence of network reordering.

TCP provides bidirectional connections between communicating applications and initializes these connections using a three-way handshake at the start of the connection. This serves to initialize the sequence numbers that will be used by each side [Pos81]. Once the connection is established, TCP sends data as allowed by its congestion control and the receiver. This takes the form of a sliding window containing the data that TCP is allowed to send. Acknowledgements from the receiver move this window forward as sent data is acknowledged. The size of this window is controlled by TCP's congestion control and the receiver and determines the throughput of the connection [APB09]. A final handshake of FIN packets terminates a connection [Pos81].

TCP has two methods for retransmitting lost data. The first is the retransmission timeout (RTO). The RTO timer is reset every time TCP sends a packet or receives an acknowledgment. If this timer expires, TCP assumes that the packet immediately above the cumulative acknowledgement has been lost and retransmits it. This timer is set for the current round trip time plus four times the round trip time variance, or a minimum of one second [PACS11].

Because TCP's acknowledgement number is cumulative, the sliding window will not advance beyond a missing packet until the data has been retransmitted and received correctly. This makes retransmissions very expensive because, by the time the retransmission timer expires, TCP will have sent its whole window of data and have been idle for at least a round trip time. Before any new data can be sent, an additional round trip time will pass while the lost packet is retransmitted. This situation is known as head of line blocking.

To reduce the cost of retransmissions, another mechanism known as fast retransmit was introduced. If a sending TCP receives three duplicate acknowledgments in a row, it assumes that the indicated data is missing and retransmits that packet without waiting for the transmission timer to expire [APB09].

TCP is not very well suited for real-time, streaming media applications because most of these applications would prefer that lost data not be retransmitted at all. These applications operate under very tight latency constraints and are not likely to have buffered a full round trip's worth of data. By the time the triple duplicate acknowledgement occurs, the data is retransmitted, and the retransmission arrives at the receiver, the time for that data to be displayed will most likely have passed [FHK06b].

Head of line blocking is an even worse issue for these applications. Having to wait a full second until the retransmission timer expires and then wait another round trip time before new data can begin flowing is completely unacceptable.

As a result, real-time, streaming media applications do not use TCP, even though it provides convenient, well-tested congestion control.

1.4 The User Datagram Protocol

The User Datagram Protocol (UDP) was defined in 1980 as RFC 768 [Pos80]. It is heavily used for DNS [Moc87] and other applications that need to send single, short messages to other hosts and receive short responses in reply, and which can handle retransmitting these messages themselves.

UDP offers applications a simple method for transmitting datagrams to another endpoint. It does not guarantee the reliability of these datagrams nor that they arrive in the order in which they were sent [Pos80].

Because UDP lacks the retransmission mechanisms of TCP and their associated latency issues, streaming media applications tend to use UDP as their network transport [CCZ03, CGL^{+}01]. However, UDP lacks a mechanism for congestion control, which means that such applications are left to implement their own congestion control algorithms as best they can—or simply ignore congestion control altogether. Many applications use the RTCP protocol, discussed in detail in section 2.1.2, to obtain information on lost packets and the current round trip time, but this still requires the application to implement a congestion control algorithm based on this information. Obviously, this is far from an optimal solution.

1.5 The Datagram Congestion Control Protocol

Recognizing the absence of a protocol to provide congestion control for unreliable traffic, the IETF developed the Datagram Congestion Control Protocol (DCCP) in 2006 as RFC 4340 [FHK06a]. This protocol allows an application to send an unreliable stream of datagrams to an endpoint in a congestion controlled manner. It also offers a selection of congestion control mechanisms.

DCCP numbers each datagram using a 48-bit sequence number. Most datagrams also include an acknowledgement number informing the other side what data has been received. Unlike TCP, this acknowledgement number is not cumulative; it simply indicates the highest sequence number received without implying anything about the reception of packets with lower sequence numbers [FHK06a]. Additional DCCP options can be used if a more detailed understanding of packet reception is desired. Also unlike TCP, every packet increments the sequence number, even pure acknowledgements. This enables the detection of reverse path congestion [FHK06a].

DCCP forms bi-directional connections between hosts using a three-way handshake similar to TCP's. The handshake enables initial synchronization of sequence numbers as well as the selection of a congestion control algorithm and other options. Once the connection is established, DCCP transmits datagrams as allowed by its congestion control. At least once a round trip time, acknowledgements are received and processed by the congestion control algorithm to provide information about current network conditions.

DCCP does not retransmit lost data nor does it guarantee in-order delivery of datagrams. Acknowledgements are used purely for congestion control. To terminate a DCCP connection, a handshake of CLOSE and RESET packets is performed [FHK06a].

DCCP includes a highly extensible option mechanism. Options can take up the entire packet and are specified as type-length-value fields. Available options include an ack vector option, which is similar to a TCP SACK block, timestamp, timestamp echo, and elapsed time options, which can be used for improved round trip time measurement [FHK06a]. All of these options are designed to gain a more detailed understanding of what is occurring in the network.

DCCP was also designed to take advantage of Explicit Congestion Notification (ECN) [RFB01] by default [FHK06a]. ECN enables routers to set a bit in the IP header indicating that they are congested. Receivers then echo this information back to the sender, providing an indication of congestion before any data is actually dropped [RFB01].

In order to cater to a wide variety of applications, DCCP offers pluggable congestion control modules, or CCIDs (Congestion Control IDs). Two are standardized at the moment, with a few other experimental CCIDs in development. CCID 2 offers congestion control that is very similar to TCP's congestion control algorithm while CCID 3 is an implementation of TCP-Friendly Rate Control (TFRC) [FHK06a]. TRFC is designed for applications that desire reduced fluctuations in sending rate compared to TCP [FHPW08]. We examine these two algorithms in more detail in Chapter 2.

DCCP seems uniquely well suited to the needs of real-time, streaming media applications since it provides congestion control without reliability. This frees the application programmer from having to design and debug a congestion control algorithm and eliminates the retransmissions and head of line blocking that come with reliability in TCP. In addition, DCCP's pluggable congestion control system enables the selection of a congestion control scheme that best suits these real-time, streaming applications or the development of a new scheme if no effective scheme currently exists.

1.6 Research Aims

Since DCCP appears to be particularly well suited for real-time, streaming media applications, we would expect it to provide distinct performance advantages over application level congestion control schemes. However, an examination of the small body of literature on DCCP shows a focus on DCCP's fairness with TCP [GDW06, LL08, TKI^{+}05] and performance in satellite [NHG10, SBJL08, SLB07] or wireless networks [CMY09, LLA^{+}04, NAT06]. Very little attention has been given to how real-time, streaming media applications perform when using DCCP.

As a result, this thesis evaluates the performance of DCCP for real-time, streaming media applications. We utilize a real VoIP/videoconferencing application, linphone [Lin13], which has been modified to support DCCP. This enables examination of the interactions between the video codec and DCCP's congestion control as well as analysis of what is required to migrate such an application to DCCP. Further, using a real application ensures that the intricacies of real application behavior are considered.

Tests are conducted both in a testbed environment and over several Internet paths spanning the majority of the United States. Both the network traffic behavior and the quality of the video stream transmitted by the application is examined and compared with results obtained using UDP for network transport.

1.7 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 gives background into streaming media and congestion control and reviews the relevant literature. Chapter 3 discusses the experimental setup and important considerations for applications using

DCCP. Chapter 4 presents and analyzes the results of the experiments, and chapter 5 presents our conclusions.

2 Background

This chapter provides background on real-time, streaming media applications, particularly the commonly utilized MPEG-4 media format and Real-time Transport Protocol (RTP). It also examines a variety of congestion control algorithms, including SACK TCP congestion control, DCCP's CCID 2 and CCID 3 algorithms, and one example of an application level congestion control scheme for real-time, streaming media applications. Finally, the literature on DCCP and video streaming using MPEG-4 and RTP is discussed.

2.1 Media Encoding and Encapsulation

Much work has gone into the development of audio and video coding formats for use in real-time, streaming media applications. Suitable formats need to be low bandwidth, low latency, high quality, and error tolerant. The development of such formats is no easy task.

The audio and video formats standardized as MPEG-4 are particularly well suited for streaming media because of the media quality achieved at lower bitrates and have become very popular in streaming media applications [GMM04, WHZ^{+}00, LM03]. They are discussed in detail in the next section both because they are generally representative formats and because they are some of the most common formats in use today. The real-time, streaming media application we utilize in this work makes use of MPEG-4 for its video streaming.

2.1.1 MPEG-4 Video

The MPEG-4 standard, officially known as ISO/IEC 14496, was developed by the Moving Pictures Experts Group of the International Standards Organization around 1999 [Int02c]. It was designed to provide a flexible method for multimedia content storage and delivery, with particular emphasis on streaming across the Internet. This multimedia content can include audio, video, animations, still images, and text, in any combination [Int02c].

Unlike the previous MPEG-1 and MPEG-2 standards, and most other formats, MPEG-4 doesn't just provide an algorithm for encoding and storing audio and/or rectangular video data. Instead, MPEG-4 defines a variety of primitive objects, like audio, video, and text, that can be composed into scenes [Int01]. These scenes can even be dynamically manipulated during a multimedia presentation; either programmatically or by viewer interaction. To complement this design, MPEG-4 also allows images and video streams to be arbitrarily shaped [Int02a].This enables the foreground and background of a video clip to be separate video streams that may use different quantization parameters or frame rates, enabling lower bitrates to produce higher quality.

The MPEG-4 standard is organized into a number of parts (30, as of this writing) each dealing with a particular component of the standard. The first three parts form the core of the standard and describe the basic structures underlying MPEG-4 streams as well as audio and video encoding [Mov13]. Many of the other parts deal with conformance testing, reference implementations, or various specialty extensions [Mov13]. We examine the first three parts in more detail below.

The first part of the specification is titled "Systems" and discusses how MPEG-4 streams are organized into object descriptors and *elementary streams* (ES) [Int01]. An MPEG-4 stream uses an initial object descriptor to gain access to an object descriptor stream, which points to all the individual multimedia elementary streams in the presentation, and a scene description stream, which contains commands to add, delete, or modify multimedia objects in the current scene [Int01]. Each audio stream, video stream, still image, or animation is contained in its own elementary stream which is added and removed from the scene using commands in the scene description stream. Additional elementary streams can be used for synchronization or content management [Int01]. This part of the standard also specifies how all these elementary streams can be multiplexed onto a single transport system.

For ease of discussion, we pass over the second part of the standard for the moment. The third part of the standard is titled “Audio” and deals with the details of audio compression in MPEG-4 [Mov13]. For general audio applications, MPEG-4 duplicates MPEG-2’s AAC compression [Int02c]. In addition, MPEG-4 introduces two new compression methods, HVXC and CLEP, for extremely low bitrate speech applications. These new methods work effectively down to 2kbits/sec [Int02c]. MPEG-4 also utilizes Reversible Variable Length Coding and Huffman Codeword Reordering to improve the robustness of audio streams to loss [Int02c].

The second part of the MPEG-4 standard, which we skipped over previously, is titled “Video” and deals with the details of video compression. Like its predecessors, MPEG-1 and MPEG-2, MPEG-4 video compression utilizes differential coding techniques for effective compression [Int02a]. This takes the form of three different types of frames. Intra-coded frames (I-frames) contain the full image texture content while predictive-coded frames (P-frames) contain motion vectors that enable reconstruction of the frame using the previous I or P frame. The third type of frame is a bidirectionally predicted frame (B-frame). These B-frames are reconstructed similarly to P-frames using motion vectors and both of the immediately adjacent I or P frames [Int02a].

The MPEG-4 standard refers to frames as *VOPs* (Video Object Planes) because they are not constrained to be rectangular, unlike traditional video frames [Int02a]. Sequences of frames, or VOPs, are divided into sets of dependent frames called *GOVs* (Group of VOPs) [Int02a]. Each of these GOVs starts with an I-frame and encompasses all frames dependent on this I-frame. One such sequence would be: IPBBPBBPBBPBBPBB.

Each VOP is divided into macroblocks, which contain the luminance and chrominance data for a 16x16 section of the image [Int02a]. Each 8x8 section of either luminance or chrominance is referred to as a block and processed separately. Since the normal YUV4:2:0 format samples chrominance at half the resolution of luminance, this results in 6 blocks in a macroblock, 4 luminance and 2 chrominance blocks [Int02a].

To encode an I-frame, or new texture data in P or B frames, each block is transformed using the Discrete Cosine Transform (DCT) and then quantized. The remaining coefficients are then encoded using a variable length code [Int02a]. In P and B frames, the motion vectors for each block are simply encoded as variable length codes [Int02a].

Because MPEG-4 was designed to operate over lossy transport layers, several error recovery and error concealment techniques were included in the standard. One of these is the insertion of a resynchronization header at periodic intervals in the bitstream. This header includes enough information to restart the block decoding process [Int02c]. Optionally, additional information can be included in these headers to recover even from a corrupt VOP header. Another key error recovery feature included in MPEG-4 is reversible variable length codes. All the variable length coded information in MPEG-4 can be decoded in both the forward and reverse directions. This enables a decoder to work backwards from a resynchronization header to recover as much of the bitstream as possible [Int02c].

Because of the size and complexity of the MPEG-4 standard, a system of profiles and levels was introduced in order to allow encoders and decoders to implement subsets of the specification in a compatible manner [Goo05]. For example, the Simple Visual Profile (SP) supports the coding of simple rectangular video in an error resilient manner and disallows the use of B-frames [Int02a]. The Advanced Simple Profile (ASP) adds support for B-frames and finer motion estimation [Int02c]. The Main Visual Profile supports interlaced or arbitrarily shaped video along with the B-frames and fine motion estimation from ASP [Int02a].

As a further complication, MPEG-4 part 10 introduces an improved video encoding method called Advanced Video Coding (AVC). This encoding scheme is identical to ITU-T Rec H.264 [Mov] but distinctly different from that in MPEG-4 part 2, which as been the focus of this section. AVC allows the decision about whether to encode data in an Intra or Predictive manner (I or P/B) to be made on a sub-frame level (the slice). In addition, AVC offers significantly improved motion compensation and adds a de-blocking filter to the decoding path [Mov].

Since Internet streaming was one of the planned uses of MPEG-4, part of the standard discusses the transportation of MPEG-4 over IP networks. The Real-time Transport Protocol (RTP) is the recommended solution, as discussed in MPEG-4 part 8 [Int02b].

2.1.2 The Real-Time Transport Protocol

The Real-time Transport Protocol (RTP) was developed by the IETF as a mechanism for transporting multimedia conferences and other real-time data streams. It was originally standardized in 1996 as RFC 1889 and later updated as RFC 3550 in 2003 [SFCJ03].

The RTP specification consists of two parts, the Real-time Transport Protocol (RTP) to carry real-time data between application endpoints and the Real-time Transport Control Protocol (RTCP) to provide participant information, session metadata, and quality-of-service information to applications [SFCJ03]. Confusingly, the combination of these protocols is commonly referred to simply as RTP.

RTP does not provide ports for multiplexing multiple sessions on a single address nor checksums for data-integrity validation. For these reasons, it is typically run on top of UDP or another transport layer protocol [SFCJ03]. Typically, the RTP data stream uses an even numbered port of the transport layer protocol and the corresponding RTCP stream uses the next higher (odd numbered) port [SFCJ03].

Each RTP packet consists of a 12 byte header followed by a media payload. The header contains a payload type identifier specifying the particular type and encoding of the transported media. Each of the payload type identifiers is defined in a separate RFC along with details on exactly how the encoded data is encapsulated [SFCJ03]. This standardization allows different applications to interoperate correctly and enables applications to readily support multiple media types. The RTP header also provides a packet sequence number, for loss and re-ordering detection, and a high-precision 32 bit RTP timestamp indicating the sampling instant of the media payload [SFCJ03]. Finally, RTP includes a source identifier (SSRC), identifying the source of this particular payload in multi-party or multi-media presentations, and a marker bit to mark important locations in the packet stream, typically the end of a frame or a new period of speech [SFCJ03, KNF+00].

While RTP provides encapsulation of media data, RTCP provides control information to the application to inform the sending and display of this media data. This control information includes periodic reports from media senders and media receivers as well as media metadata and application specific information [SFCJ03].

RTCP was designed for large, multi-party events, for example, multicasted IETF working group meetings [SFCJ03]. For this reason, RTCP maintains a rough idea of the number of senders and receivers in a session and adjusts its feedback rate to avoid consuming an unreasonable amount of bandwidth with this feedback information [SFCJ03]. An estimate of the number of senders and receivers is maintained by adding the SSRC from any received RTP or RTCP packet to a table and then periodically removing entries that haven't been heard from in several RTCP report intervals. Using this information, RTCP computes a report interval such that at most 5%

of the session bandwidth is consumed by feedback information [SFCJ03]. This bandwidth is further divided so that senders get at least 25% of this feedback bandwidth. This ensures that they can send metadata with reasonable frequency. In addition, RFC 3550 imposes a minimum interval of 5 seconds, or 2.5 seconds at the start of a session, between RTCP packets sent by an individual host [SFCJ03].

An RTCP feedback packet contains a variety of information. The simplest case is a feedback packet from a host that is only a receiver. In this case, the feedback packet simply contains a Receiver Report (RR) [SFCJ03]. An RR contains the unique, identifying SSRC value for this receiver and a set of statistics blocks about each of the senders heard from since the previous RR. Each statistic block contains the SSRC of the sender, the fraction of RTP packets lost since the last RR, the cumulative number of lost packets, the highest RTP sequence number received, and an estimate of the jitter from that source. In order to enable round trip time calculations, the NTP[1] timestamp from the last Sender Report (SR) and the delay between receiving that SR and sending this RR are also included [SFCJ03]. These RRs provide senders with a variety of important information about network conditions and received media quality.

Feedback packets from a sending host contain both a Sender Report (SR) and source description items (SDES). An SR is very similar to an RR; it starts with the unique, identifying SSRC for this sender and finishes with a set of statistics blocks about each of the senders heard from since the last SR [SFCJ03]. However, it also includes some information about the RTP stream being sent. In particular, an NTP timestamp and a matching RTP timestamp are included so that the RTP timestamps in RTP packets can be matched against wall clock time, enabling the synchronization of multiple streams with different RTP timestamp increments. Each SR also includes the total number of packets and octets of RTP data sent by this sender [SFCJ03].

[1] Network Time Protocol, see [MMBK10].

The SDES items sent with each feedback packet from a sender contain metadata about the RTP stream. The only required metadata is CNAME, a fixed identifier for all streams originating from the same host. It is recommended that it takes the form *user@host* [SFCJ03]. Other optional metadata tags include NAME, EMAIL, PHONE, LOC (location), and NOTE.

RTCP enables a media sender to gather lots of information about network conditions and the quality of media reception at the receivers. However, RTCP does not specify what senders or receivers are supposed to do with this information [SFCJ03]. This is partly because the reaction depends on the session infrastructure and environment. For example, a single poor receiver in a large multicast session should join a lower quality multicast group or drop the session while a sender in a one-to-one video conference should reduce its sending rate in response to poor reception information from the receiver. However, this leaves individual applications to develop their own algorithms for rate adjustment and congestion control.

As mentioned above, RTP requires an additional RFC to standardize the transport of a particular media type and assign a payload type identifier. For MPEG-4 audio and video, this is RFC 3016 [KNF$^+$00]. This standard does not make use of MPEG-4's object multiplexing or scene description information. A single audio or video elementary stream is simply encapsulated in RTP packets. Multiple MPEG-4 elementary streams require multiple RTP flows and scene composition information, if needed, must be supplied via additional configuration [KNF$^+$00].

An MPEG-4 video bitstream is mapped into an RTP payload without any modifications. No additional headers are added nor is any data removed. However, there are restrictions on where the bitstream is broken between packets. Specifically, MPEG-4 headers may only occur at the beginning of a packet and may not be broken between packets [KNF$^+$00]. It is recommended that only one VOP be contained in each packet.

These restrictions are designed to reduce the impact of lost packets on the decoding of other, correctly received, packets. This standard also specifies that the RTP marker bit must be set on the last packet of a VOP and that the RTP timestamp should increment at 90kHz [KNF+00].

MPEG-4 audio access units from an audio elementary stream are similarly mapped into an RTP payload. No additional headers are added nor is any data removed. Each RTP packet contains a complete access unit or a part of one [KNF+00]. As with the MPEG-4 video mapping, the RTP marker bit must be set on the last packet of an access unit and the RTP timestamp increments at 90kHz [KNF+00].

2.2 Congestion Control

A great quantity of research has been poured into understanding network congestion and developing effective congestion control algorithms to avoid congestion while maximizing network utilization. This section examines several of these algorithms, focusing on those used with RTP and DCCP. We start with possibly the most common and best studied congestion control algorithm, TCP's New Reno algorithm with SACK support.

2.2.1 SACK TCP

The development of TCP's congestion control began with Jacobson's seminal 1988 paper [Jac88] in which he proposed five new algorithms designed to avoid congestion collapses like those experienced on the Internet in 1986. These algorithms were based on a principle that Jacobson called "conservation of packets" [Jac88]; that is, for a stable TCP flow with plenty of data to transmit, a new packet is not put into the network until an old packet has left the network.

Recall from section 1.3 that TCP offers reliable, in-order transmission of a byte-stream between applications. To accomplish this, TCP forms bi-directional connections between hosts and numbers each byte of application data. These sequence numbers enable the detection of lost or re-ordered data. TCP maintains a sliding window containing the data that it is allowed to send, and this window is moved forward by acknowledgements from the receiver. TCP congestion control operates by adjusting the size of this window, which corresponds to the amount of data maintained in the network and hence the connection throughput [APB09].

Jacobson proposed a congestion control scheme controlled by two state variables. The first, called *cwnd*, is the current congestion window, that is, the amount of data that TCP's congestion control will allow to be in the network at a given moment. The second is *ssthresh*, the slow-start threshold. This variable determines what mode TCP congestion control is currently in. If *cwnd* is below *ssthresh*, then TCP is in slow-start. Otherwise, TCP is in congestion avoidance mode [Jac88, APB09].

TCP starts with *cwnd* equal to between 2 and 4 packets depending on the MTU[2] of the path[3] and *ssthresh* initialized to infinity. This places TCP in slow-start mode. In slow-start, TCP increases *cwnd* by one packet for each new acknowledgement received from the receiver. Since the TCP receiver must acknowledge at least every other packet received [Bra89], this results in exponential growth of *cwnd*. This mode is designed to rapidly reach the equilibrium rate of a connection.

When a loss occurs, TCP halves *cwnd*, sets *ssthresh* to this new value of *cwnd*, and enters congestion avoidance mode [APB09]. In this mode, TCP increases *cwnd* by one packet per round trip (i.e. one packet every *cwnd* packets). This results in a gradual, linear increase in *cwnd* [Jac88]. This mode is commonly described as additive increase, multiplicitve decrease (AIMD).

TCP's original method for detecting packet loss was the retransmission timeout (RTO). An RTO occurs if TCP sends its whole window of data and a packet still has not

[2] Maximum Transmission Unit, the size of the largest packet that could be sent on this network.

[3] see RFC 5681 [APB09], section 3.1 for details.

been acknowledged after at least a round trip time plus four times the round trip time variance, or a minimum of one second [PACS11]. TCP's response is to assume that all sent data has drained from the network and set *cwnd* to one packet [APB09]. Because an RTO implies that a loss occurred, TCP also sets *ssthresh* to half of the old *cwnd* value. TCP can then retransmit the lost packet, slow-start to half its speed prior to the RTO, and enter congestion avoidance [APB09].

A later improvement to TCP was the addition of a second method for detecting packet loss and a set of algorithms known as fast retransmit and fast recovery [Ste97]. This method relies on the fact that TCP's acknowledgements are cumulative so the acknowledgement number cannot advance beyond a missing packet. This results in duplicate acknowledgements when later packets with higher sequence numbers arrive following a loss. If TCP receives three duplicate acknowledgements in a row, it assumes that the indicated packet has been lost and retransmits the missing packet [APB09]. This is the fast retransmit algorithm.

Fast recovery now takes over and sets *ssthresh* to half of *cwnd*. *Cwnd* is set to half of its previous value plus the size of the three packets that are known to have left the network, as indicated by the three duplicate acknowledgements [APB09]. Each additional duplicate acknowledgement increases *cwnd* by one packet because it indicates that another packet has left the network. This continues until a cumulative acknowledgement is received acknowledging new data. At this point *cwnd* is set back to *ssthresh*, undoing the increases made for the duplicate acknowledgments. TCP then exits fast recovery [APB09].

A final optimization to TCP's congestion control utilizes TCP's Selective ACKnowledgement (SACK) option. SACK, introduced in 1996 as RFC 2018 [MMFR96], provides more information about what packets have actually been received. Basically, SACK allows a receiver to signal up to four blocks of received packets above the cumulative acknowledgement. A later RFC extended this concept to allow signaling needlessly retransmitted blocks as well [FMMP00].

This additional information can be leveraged to further improve TCP's congestion control, particularly determining how much data is actually in the network and what packet to send next. RFC 3517, standardized in 2003, provides the details [BAFW03]. A structure called a "scoreboard" stores information about all packets currently in TCP's sliding window. This scoreboard is updated as packets are sent, cumulative acknowledgements received, and SACKs processed.

The scoreboard is used to choose the optimal packet to send in fast retransmit/fast recovery. In addition, a variable called *pipe*, representing the amount of data in the network, can be computed from the scoreboard. In fast recovery, *cwnd* is no longer inflated for each duplicate acknowledgement. Instead, data is transmitted when *pipe* is less than *cwnd* [BAFW03]. Since *pipe* represents a more accurate understanding of the data actually in the network, this enables TCP to utilize its now reduced window more fully. For connections with high loss rates, this can result in significant improvements.

TCP congestion control has a long and complex history. Its basic principle is simple: a multiplicative decrease on loss and additive increase otherwise. This is combined with slow-start to rapidly increase speed on start up. However, a variety of algorithms have been added over the years to more accurately follow this principle. As a result, we have a highly-optimized mechanism that avoids congestion and competes fairly with other network flows.

2.2.2 RTP-Based Congestion Control

For those real-time applications using RTP, where TCP is undesirable because of its retransmissions and head of line blocking, a different set of congestion control algorithms have evolved. These algorithms rely on the loss rate and round trip time information supplied in RTCP feedback packets in order to detect and mitigate network congestion.

Interestingly, the *Extended Profile for RTCP-based Feedback* [OWS+06] explicitly states that RTCP feedback should not be used for congestion control purposes. This applies even to the extensions made by that document, which remove the five second minimum report period. The reason given is that RTCP operates over a much longer timescale than effective congestion control. Instead, that RFC recommends the use of TCP-Friendly Rate Control (TFRC) [FHPW08]. Nevertheless, applications continue to rely on RTCP feedback for congestion control [Lin13, BDS96, WHZ+00, FB02].

The scheme used by linphone, an open source SIP VoIP client [Lin13], appears to be fairly representative. From examination of the source code,[4] it appears that linphone uses a very simple heuristic for congestion control. Linphone monitors the loss rate and round trip time (RTT) indicated in each RTCP report. If the loss rate exceeds 10%, linphone asks the media encoder to reduce its encoding rate by a percentage equal to the current loss rate. If the RTT doubles between reports, linphone reduces its encoding rate, and hence its sending rate, by 20%.

To take advantage of any additional bandwidth that might come available, linphone enters an increasing mode after ten RTCP reports without high loss rates or sudden increases in RTT. In this mode, linphone allows its media encoder to increase its rate by 20% over the next four RTCP report intervals.

Other congestion control schemes based on RTCP include [BDS96] and [WHZ+00]. The scheme presented in [BDS96] is designed for use in environments with multiple receivers and uses two loss thresholds. The first threshold is 5% loss and indicates a network in a loaded, but acceptable, state. The second threshold is 10% loss and indicates a congested network. The algorithm presented in this paper decreases throughput when

[4] Available from https://www.linphone.org/eng/download/git.html. Our version pulled on 12-27-2012.

10% of the receivers are congested and increases throughput when 80% of the receivers are unloaded (i.e. below the first threshold).

By contrast, [WHZ$^+$00] proposes an AIMD scheme. A multiplicitve decrease (5% in their tests) in sending rate occurs when an RTCP report indicating a loss rate above 5% is received. Otherwise, an additive increase in sending rate of 0.5Kbits/sec occurs. These adjustments are bounded between minimum and maximum allowed sending rates.

All of these schemes suffer from the basic issue that RTCP feedback occurs only once every five seconds. For real-time data flows, this represents tens to hundreds of round trips; hardly an effective granularity for avoiding congestion, although it does work well enough to detect significant, *persistent* congestion.

Because of the slower response time to changing network conditions, these RTCP-based congestion control methods are likely to be unfair to TCP, which will respond to congestion within a single round trip. In fact, it is possible that a TCP connection will slow down multiple times before an RTCP regulated connection even learns about the congestion.

Further, Mathis [MSMO97] showed that that sending rate achievable using TCP's congestion control algorithm is a function of the round trip time. This is in contrast to these RTCP-based algorithms which have no round trip time dependence. As a result, the competition and fairness between TCP and these RTCP-based algorithms will depend on the connection's round trip time.

It is also important to realize that these RTCP-based algorithms are application level. That is, each application must implement its own congestion control. This is in marked contrast to TCP where congestion control is built into the protocol itself and automatically available without any extra work on the part of the application. Indeed, the desire to eliminate the duplication of effort involved in application level congestion control was a major motivator in the design of DCCP and its pluggable congestion control modules [FHK06a, FHK06b].

2.2.3 DCCP Congestion Control

DCCP offers a selection of modular congestion control algorithms to applications, enabling them to select that which is most appropriate [FHK06a]. Some applications, like VoIP, may desire a smooth sending rate instead of the fastest rate possible while other applications, like video games, may be able to tolerate sudden changes in rate and would prefer to send as fast as possible.

To accommodate such diverse applications, two congestion control algorithms, or CCIDs, are currently standardized, with two more experimental CCIDs in development. We focus on the two standardized CCIDs in the discussion below.

2.2.3.1 CCID 2: TCP-Like Congestion Control

TCP-Like Congestion Control is one of the standardized DCCP CCIDs and is numbered CCID 2 [FK06] (CCIDs 0 and 1 are reserved [FHK06a]). This CCID is designed to closely replicate the behavior of SACK TCP.

Recall from section 1.5 that DCCP offers congestion controlled transmission of a stream of datagrams between two hosts. Each datagram has a unique sequence number and some datagrams contain an acknowledgement number indicating the highest sequence number received by the sending host. Note that this acknowledgement number is not cumulative like TCP. DCCP does not retransmit lost packets or ensure in-order delivery; the acknowledgement information is utilized only for congestion control.

Because CCID 2 is designed to follow TCP's congestion control, it utilizes similar slow-start and congestion avoidance modes with, *cwnd*, *ssthresh*, and *pipe* variables. An important difference is that CCID 2 measures these variables in packets while TCP measures them in bytes. This is necessary because DCCP's sequence numbers are also in terms of packets and not bytes [FK06]. Just as in TCP, *cwnd* represents the current congestion window, or the amount of data CCID 2 attempts to keep in the network, and *ssthresh* represents the point of transition between the slow-start and congestion avoidance modes. When *cwnd* is less than *ssthresh*, CCID 2 is in slow-start mode. Otherwise, it is in congestion avoidance mode [FK06].

CCID 2 starts with *cwnd* equal to between 2 and 4 packets depending on the path MTU. This places it in slow start mode. In this mode, CCID 2 increases *cwnd* by one packet for every two newly acknowledged packets. This results in an exponential window increase similar to TCP [FK06].

When a loss is detected, either by a Time Out (TO) or the observation of three acknowledged packets above some unacknowledged packet, CCID 2 halves *cwnd*, sets *ssthresh* to this new value of *cwnd*, and enters congestion avoidance mode [FK06]. In this mode CCID 2 increases *cwnd* by one packet every *cwnd* packets (i.e. once a round trip time). This results in AIMD behavior identical to TCP [FK06].

The Time Out (TO) timer mentioned above parallels TCP's RTO timer and is used to recover from an entire window of lost packets. TO is set to twice the round trip time. Unlike TCP, there is no one second minimum [FK06]. If a TO occurs, CCID 2 assumes that all data has left the network, sets *pipe* to zero, *ssthresh* to half of the old value of *cwnd*, sets *cwnd* to one, and enters slow-start [FK06].

CCID 2 requires the use of the ack vector DCCP option on acknowledgement packets from the receiver [FK06]. This option lists the state of all packets between the highest sequence packet received and the last sender side acknowledgement and is conceptually similar to a TCP SACK option. It allows the sender to know exactly what packets were received, ECN-marked, or presumed lost [FHK06a]. This enables the computation of *pipe*, the number of packets in the network, in a manner very similar to SACK TCP.

CCID 2 also utilizes DCCP's ack ratio option to provide simple reverse path congestion control [FK06]. DCCP's ack ratio option allows the sender to control how often a receiver sends acknowledgement packets. Specifically, a sender asks that every *n*th packet be acknowledged [FHK06a] (CCID 2 starts with $n = 2$ by default). When combined with sequence numbers that increment for every packet, which enable the detection of lost acknowledgement packets, this allows DCCP to adjust the rate of acknowledgement packets to avoid congestion on the reverse path [FK06]. While this is usually not an issue, given the small size of most acknowledgements, it can become a problem, particularly on links with asymmetric bandwidth. CCID 2 includes controls to adjust the ack ratio so that the acknowledgement rate is roughly TCP friendly [FK06].

While CCID 2 is similar to SACK TCP in many respects, it does have some important differences. One of these is that CCID 2 lacks Fast Recovery, the TCP algorithm to artificially increase the window while retransmitting data following a triple duplicate acknowledgement. The reason for omitting this algorithm is obvious: DCCP doesn't retransmit any data. However, this omission has the rather interesting side effect, as pointed out in [TKI$^+$05], of causing unfairness with TCP. In particular, CCID 2 can take significantly more than its fair share of bandwidth when competing against TCP because CCID 2 can grow its window in the round trip immediately following a loss while TCP will reset its window on exiting Fast Recovery. The authors of [LL08] have proposed introducing a virtual recovery period into DCCP in order to counteract this tendency, but this has not yet been standardized.

One of the experimental CCIDs currently being developed is similar to CCID 2 but is based on the CUBIC TCP algorithm instead of SACK TCP [Ren11]. This CCID has not been standardized yet, but is being implemented in Linux. It is currently being referred to as CCID 5.

2.2.3.2 CCID 3: TCP-Friendly Rate Control

CCID 3 is an implementation of TCP-Friendly Rate Control (TFRC) [FKP06]. TFRC is designed for applications that prefer smooth changes in sending rate over maximum throughput [FHPW08]. In particular, it avoids the halving of the sending rate in response to a single loss that TCP incurs. In theory, this should be beneficial for VoIP traffic and most other media streaming applications.

To accomplish this goal, TFRC takes a distinctly different approach to congestion control. The basic idea is to measure RTT and loss rate and then utilize an equation to identify the sending rate that TCP would achieve in this environment [FHPW08]. Because TCP is responsive to congestion, a sending rate calculated in this manner is reactive to congestion. Further, TFRC should, at least in theory, share bandwidth fairly with TCP since it achieves the same sending rate.

The throughput equation used by TFRC is based on the throughput achieved by Reno TCP and is as follows [FHPW08]:

$$X_{bps} = \frac{S}{R * \sqrt{2 * b * p/3} + (t_{RTO} * (3 * \sqrt{3 * b * p/8} * p * (1 + 32 * p^2)))}$$

Where X_{bps} is the allowed sending rate in bytes per second, S is the packet size in bytes, R is the round trip time in seconds, b is the maximum number of packets acknowledged by a single TCP acknowledgement, t_{RTO} is the TCP RTO time in seconds, and p is the loss event rate [FHPW08]. Note that p is the loss event rate, not the loss rate; the two are similar but not identical. A loss event is an RTT's worth of lost or ECN-marked packets [FHPW08].

The TFRC specification recommends setting $b = 1$ and $t_{RTO} = 4 * R$ which simplifies the throughput equation to [FHPW08]:

$$X_{bps} = \frac{S}{R * \sqrt{2 * p/3} + 12 * \sqrt{3 * p/8} * p * (1 + 32 * p^2)}$$

It is recommended that CCID 3 either use the path MTU as the packet size, S, or compute an average over the last four loss intervals of packets [FKP06]. The round trip time, R, can be measured using the feedback packets from the receiver while the loss event rate, p, can be computed at the receiver, if it has some estimate of the round trip time, and returned on the feedback packets.

A TFRC receiver sends feedback packets to the sender approximately once per round trip time. In order for the receiver to have some idea of the round trip time, the DCCP header includes a CCVAL field which CCID 3 uses to store a quarter RTT counter [FKP06, FHPW08]. In this manner, the receiver can identify a round trip as the length of time between packets whose CCVAL fields differ by four.

Each CCID 3 feedback packet contains an elapsed time option, indicating the amount of time between the reception of the acknowledged packet and the sending of this feedback packet, a receive rate option, specifying the rate at which data has been received since the last feedback packet was sent, and a loss intervals option or loss event rate option [FKP06]. The loss event rate option specifies the loss event rate, p, used in the TFRC formula while the loss intervals option specifies the length of the last eight loss intervals, enabling the sender to calculate p independently [FKP06].

Using the information from the feedback packets, and the TFRC equation above, TFRC can calculate the equivalent sending rate of TCP under these conditions. TFRC further restricts the allowed sending rate to be no more than twice the maximum of the receiver's receive rate in the last two RTTs [FHPW08]. This prevents sudden bursts in throughput when an application attempts to send a large amount of data after having sent very little for several round trips. In other words, when TFRC exits an application limited period.

At this point TFRC has an allowed sending rate in bytes per second. This rate is then divided by the packet size, S, to determine the sending rate in packets per second. TFRC then uses a timer to clock out packets at this rate, provided it has packets to send [FHPW08].

TFRC employs a No Feedback timer to reduce throughput if no feedback packets have been received from the receiver in the last four round trips. If this timer expires, the current sending rate is cut in half and the No Feedback timer reset [FHPW08].

Using the TFRC throughput equation to compute an initial sending rate would be problematic because both the loss event rate and the round trip time are initially unknown. For that reason, CCID 3 initializes the sending rate to between 2 and 4 packets per second, depending on the path MTU, and doubles the allowed sending rate every round trip until the first loss [FKP06]. The result is a controlled, but rapid, increase in throughput at the beginning of the connection.

This rounds out the TFRC congestion control algorithm. The goal of this algorithm is to offer smoother changes in sending rate while being reasonably fair to TCP for those applications that would prefer to avoid TCP's sudden changes in rate, particularly its halving the sending rate in response to a single packet loss.

A slightly modified version of TFRC, optimized for the small packet sizes often seen in VoIP systems, has been proposed. It is still very much experimental but has been numbered CCID 4 [FK09].

2.3 Literature Review

This section examines the literature on MPEG-4/RTP streaming and DCCP performance.

2.3.1 Performance of MPEG-4/RTP Streaming

MPEG-4 and RTP are both well established standards with a significant body of research examining their performance. This body of work includes application case studies as well as studies on the effects of packet loss in MPEG-4 streams and examinations of the fairness of MPEG-4/RTP flows to other traffic.

Unlike other applications of MPEG-4 and video compression technologies, data loss, in the form of dropped packets, is an expected part of video streaming. As a result, much research has gone into examining the quality of streamed video and its relationship to the loss rate. [PUN12] analyzes the quality of MPEG-4/AVC and MPEG-2 video using the PEVQ method while [LAG03] examines how quality degrades as loss run length changes. The authors of [JLddM10] examined the bursty nature of video frames in wireless networks and how that relates to loss rate and video quality. They found that evenly spacing the packets making up a frame reduced loss rate and noticeably improved quality.

A significant amount of work has also gone into mitigating the effects of packet loss. Research efforts include modifying MPEG-4 to use conditional replenishment instead of motion vectors in P-frames as done in [LTG99] and using different RTP encapsulation mechanisms as suggested by [GMM04]. Some form of retransmission is a common suggestion [FB02, WSL00, RCPC99]; however, all successful tests of such systems utilize latencies far too high for real-time, interactive video. Given typical round trip times on the Internet, it seems unlikely that this would ever be practical for real-time, interactive applications.

Forward Error Correction (FEC) is another common proposal for mitigating the effects of packet loss and is being used by several commercial videoconferencing products [WSL00, AML03, Wai08, ZXH+12]. An interesting hybrid approach is to combine FEC or retransmission with MPEG-4 scalable video, where additional video streams can enhance the quality of a so-called base stream. The error correction via retransmission or FEC is only done on the base layer. This approach is utilized in both [RCPC99] and [AML03].

There has also been active research into rate control methods for video streams. The authors of [WHZ$^+$00] suggest an Additive Increase, Multiplicative Decrease scheme similar to TCP while [LM03] proposes a binomial scheme where increase is inversely proportional, and decrease directly proportional, to the current window. The authors argue that this binomial scheme shows reduced window variation relative to AIMD. Yet another proposal is [LTG99], which recommends computing a TCP-friendly sending rate based on network loss rate and round trip time. This approach is very similar to TFRC, used by DCCP CCID 3.

The rate control algorithm presented in [BDS96] determines its sending rate by attempting to keep the network loss rate within a certain range and increasing or decreasing its bitrate as needed to do so. What all of these studies have in common is the usage of RTCP reports to provide information for the congestion control algorithms. As we have already discussed in section 2.2.2, this introduces significant granularity into the measurements, making effective congestion control difficult.

Another complexity of video traffic is its general burstiness. The authors of [KT97] propose a model of video bitrate variation that develops models for the sizes of I,P, and B-frames separately and then mixes those models together according to the GOV pattern, which is assumed to be fixed. Realizing that burstiness complicates round trip and retransmission timeout calculations, the authors of [BA05] develop a retransmission timeout algorithm that outperforms TCP's RTO timer algorithm for video data. They also assume a fixed, relatively small, GOV pattern and make assumptions about packet timing from this pattern. General models of network traffic burstiness include [DR06] and [LV91]. These works are mostly concerned with the burstiness of aggregate network traffic, but offer insights into, and metrics for, burstiness in general.

Several researchers have examined the TCP-friendliness of video streams from a few different applications and found mixed results. In 2001, [HAOS01] examined a variety of common streaming media applications, including RealPlayer and Windows Media Player, and found them all to be responsive to congestion but not fair to TCP. However, by 2003, when [CCZ03] examined the TCP-friendliness of RealPlayer, its congestion control had been improved to be TCP-friendly, in that it achieved similar rates given equivalent network conditions. However, the authors still observed noticeable unfairness in UDP's favor when in competition with TCP, particularly at low throughput. This confirms our assertion that congestion control is notoriously hard to design and implement properly. That said, it is not impossible. Two separate studies have examined Skype's congestion control and found it to be reasonably fair to TCP [ZXH+12, CMP08].

A final work that holds particular interest for us is [LKP08]. This work develops a model for MPEG-4 videoconference type streams and points out that videoconference material is distinctly different from other common video sources, particularly because of its lack of scene changes. We are particularly interested in videoconference material because that is exactly the purpose for which linphone [Lin13], our test application, was designed.

2.3.2 DCCP Performance

Since its inception, a few researchers have examined DCCP with the intent of understanding its performance. Most of these studies have been content to simulate the performance of DCCP in general or have focused on using CCID 3 in challenging environments like wireless or satellite links.

One aspect of DCCP that has been well studied is the fairness of its CCIDs with TCP. The authors of [GDW06] examined DCCP CCID 3's fairness with TCP using the OPNET Modeler and found that CCID 3 is reasonably fair with TCP in low to medium loss environments. In high loss environments CCID 3 gains more bandwidth than TCP, likely because of RTOs. Note that reasonably fair competition is generally accepted in the

Internet community to mean that the throughput of the competing protocols are within a factor of two of each other [WH06, FHPW08]. CCID 2's fairness with TCP was investigated experimentally in [BBM08]. The authors found that CCID 2 is reasonably fair to TCP for round trip times between 20 and 200ms. Beyond that point there is significant unfairness. [TKI$^+$05] elaborates on this by suggesting that DCCP's lack of Fast Recovery is at least part of the reason for this unfairness.

DCCP's fairness with TCP has also been examined in [NHG10]. This paper used simulations between TCP, CCID 2, CCID 3, and UDP to examine fairness in long delay environments. The authors show that both DCCP CCIDs behave nicely with TCP in congested environments, unlike UDP. The strength of their results is significantly reduced because they kept TCP window limited during the course of their simulations, not allowing it to compete for its full share of link bandwidth.

CCID 3 has been a particularly hot topic for researchers, especially its behavior in challenging environments. The authors of [FdFPM10] examined CCID 3 at Gigabit speeds and identified issues with the scalability of single CCID 3 connections at that speed. However, CCID 3 was able to scale with respect to number of connections in this environment.

The authors of [WKD11] also investigated CCID 3, but their focus was on the relationship between round trip time, queue size, and loss rate. Using OPNET Modeler, they found an inverse relationship between queue size and loss rate and a direct relationship between queue size and round trip time. They also observe that a large portion of packet drops occur at the very end of slow start.

CCID 3 has also been investigated in wireless networks by [LLA$^+$04], [NAT06], and [dSOPdM08]. [LLA$^+$04] and [NAT06] utilize the NS-2 simulator and the DCCP module designed by Mattsson in [Mat04]. Both studies focus on ad-hoc mesh networks. The authors of [LLA$^+$04] show that CCID 3 has a difficult time detecting wireless link saturation while [NAT06] examined the competition between CCID 3 and TCP in this environment and found it to be reasonably fair.

Researchers out of the University of New South Wales have produced a large quantity of research on the performance of DCCP CCID 3 in satellite networks. In [SLB07], they found that CCID 3 performs much worse than it theoretically should and that its performance could be improved by sending feedback more than once a round trip. Then in [SBJL08] they offer a formula for computing a better feedback rate based on round trip time and receive rate. Finally, in [SBL09] they extend this analysis to CCID 4 and offer similar improvements. The authors of [SF07] also investigated the performance of CCID 4 in satellite networks and found problems with VoIP silence suppression triggering repeated slow start.

A few researchers have examined DCCP for the purposes of video streaming, like we do in this work. However, all of these authors utilized simulations instead of actual network experiments as we do. The authors of [AMM09] examine CCID 2, CCID 3, UDP, and TCP for the transport of MPEG-4 video using the NS-2 simulator. They observe that TCP obtains the best performance but at the cost of jitter and delay. CCID 2 provides a reasonable second best, beating CCID 3 because it can react faster to changing application data rates. No consideration is given to the timeliness factor of video.

[CMY09] utilizes NS-2 simulations to examine MPEG-4 video streaming in wireless networks over DCCP CCID 2 and SCTP. The authors examine network throughput, delay, and jitter for MPEG-4 video streaming over CCID 2, SCTP, and UDP in the absence of other traffic. They found that SCTP and CCID 2 both perform better than UDP. Importantly, this paper made no consideration of received video quality, focusing on network conditions instead.

The authors of [VSB06] take a larger view of video streaming and consider video quality in addition to network conditions. They focus on how DCCP CCID 3 shares bandwidth with other CCID 3 flows and the corresponding results on video quality, again using NS-2 simulations. They observe that equal bitrate does not necessarily correspond to equal quality and suggest that CCID 3 be modified to allow unused bandwidth from low complexity points to be used later for sending more complex scenes. The focus of this work is on competition between video streams so competition with other traffic types was not examined.

Our work builds on this literature by considering video quality and network conditions in real testbed and Internet networks. We compare DCCP with a modern UDP/RTP-congestion control algorithm and are forced to deal with the complexities of an actual MPEG-4 bitstream. The full details of our experiments are presented in the following chapter.

3 Experimental Setup

This chapter discusses the real-time, streaming media application used in this work and our experience in modifying it to use DCCP instead of UDP. We also discuss the Linux DCCP implementation, video quality metrics, and our various experimental setups.

3.1 Linphone

This work utilizes linphone, an open source SIP VoIP client capable of video calls, as a representative real-time, streaming media application. Linphone is sponsored by Belledonne Communications and is extremely cross platform, operating under Linux, Windows, Mac OS X, iOS, Android, Blackberry, and WebOS [Lin13].

Linphone claims to be one of the very first open source SIP VoIP clients, with development beginning in 2001 [Bel13]. It features the ability to utilize multiple SIP accounts, SIP proxy support, and call management features like hold, resume, and transfer [Lin13]. Acoustic echo cancellation is included along with the option for RTP encryption. Video calls can be made at resolutions up to 800x600 and frame rates of up to 25 frames per second. Full HD support (up to 1920x1080 at 25 frames per second) is essentially complete, but has yet to be enabled by default.[5] Finally, linphone has support for a wide variety of media encoding formats and a modular framework making it easy to add new encoders and decoders, or codecs.

Notice that linphone uses a frame rate of 25 frames per second for high quality video. This frame rate is used with the PAL analog television system used in much of Europe, Asia, and Australia [Ado13]. As linphone is sponsored by Belledonne Communications, which is based out of France [Bel13], this is not particularly surprising. In the United States, a frame rate of 29.97 (color) or 30 (black & white) frames per second is much more common because of the NTSC analog television standard used by the US, Japan,

[5] We enabled full HD resolution, 1920x1080, in the build of linphone used in our experiments.

and South America [Ado13]. We elected not to modify linphone's frame rate. This was partly because the situation with digital television is even more confusing with several of the competing formats supporting a variety of frame rates [Adv07, Adv95, Dig12]. In addition, we found it challenging enough to get linphone to encode each frame in the 40ms between frames at 25 frames per second. Increasing the frame rate did not seem to be a wise idea.

It is worth noting that linphone already has a reputation in the research community, having been used in a variety of projects examining VoIP software, NAT traversal, and VoIP codecs [JSB+09, KC11, LTHW10, WW07, WW08].

3.1.1 Architecture

Linphone is composed of several libraries that sub-divide the application into several subsystems. Liblinphone is the main library against which all the graphical front ends link. Liblinphone is written in C and links together several other libraries and protocols to provide a simple API for audio and video calls. The lower level libraries, also written in C, include oRTP, which handles RTP and RTCP, and mediastreamer2, which provides an API and framework for capturing, playing, and streaming audio or video. A set of two external libraries, libosip2 and libeXosip2, handle the SIP protocol.

The oRTP library takes media data and encapsulates it into RTP packets which are further encapsulated by UDP. It also handles all of the RTCP computations and automatically sends sender and receiver reports as needed. This library also offers NAT detection and traversal technology.

Mediastreamer2 is a comprehensive library for media streaming. It can be used to create pipelines of filters to capture, manipulate, send, receive, and display audio or video. Each input, output, and supported codec is its own filter and the library provides an API to chain these filters together and then control these chains. Mediastreamer2 also has an event queue that oRTP can use to report network conditions. This information is then used by mediastreamer2's bitrate control to adjust the codec's target bitrate.

Mediastreamer2's bitrate control algorithm operates by examining the network's loss rate and round trip time (RTT), as mentioned in section 2.2.2. If the network loss rate exceeds 10% or the RTT doubles between two RTCP reports, then the bitrate is reduced. The bitrate is slowly increased after ten RTCP reports indicating neither of these conditions. It is important to note that mediastreamer2 will only attempt to adjust the target bitrate of the video stream. Neither the frame rate nor the resolution will be adjusted as a result of network conditions.

Mediastreamer2 supports a wide range of audio formats including speex, PCMU, PCMA, GSM, G722, and L16. On the video side, H.263, theora, motion jpeg, and MPEG-4 are supported. Examination of the source code shows that the MPEG-4 encoder is using the Simple Visual Profile. This implies that no B-frames are being used. Since B-frames depend on both the preceding and following frames, they would have twice the probability of being distorted by packet losses, making their exclusion an understandable choice. Internally, encoding and decoding for H.263, motion jpeg, and MPEG-4 is done using the libav[6] library [Lib13].

3.1.2 Modifications

Standard linphone does not support DCCP, so our first task was to add DCCP support. It was also necessary to make a few other modifications in order to complete support for full 1920x1080 HD video and allow video input/output from files.

Adding DCCP support was complicated by the fact that DCCP is a connection-oriented protocol while UDP is inherently connectionless. This means that a

[6] Formerly ffmpeg.

DCCP socket has to be connected to a specific host before data can be sent while a UDP socket can send each packet to a different host.

In linphone's oRTP library, each RTP "connection" utilizes one UDP socket for RTP data and another UDP socket for RTCP data. We added DCCP support for the RTP stream while continuing to utilize UDP for the RTCP stream, because of the small amounts of RTCP data sent and the long intervals between bursts. To accomplish this task, we modified oRTP to use three sockets for RTP data in place of the one UDP socket used originally. One of these sockets is used for sending, one for receiving, and one to listen for new connections. When using DCCP, these are actually three separate sockets while with UDP they become references to the same single socket.

The first step in adding DCCP support was adding the ability to choose UDP or DCCP for data transport at RTP "connection" creation time. This was a crucial design decision in our work: we wanted to add DCCP support in such a way that one could choose between UDP and DCCP transport on the fly.

With that in place, we added a new set of functions to create the needed DCCP sockets and set CCIDs and other socket options as needed. We then added code to the RTP send function to attempt to connect the DCCP socket if it was not already connected. Similarly, we modified the RTP receive function to check if a connection is using DCCP and has a valid receive socket; if that is the case, then it tries to receive data on the socket. Otherwise, the function checks to see if any connections are pending and accepts one. In the case of socket errors, the relevant socket is simply closed. The next time a send or receive is attempted, a new connection will be started. None of this new behavior effects the old UDP processing path.

When an application sends a packet, DCCP adds this packet to the connection's send queue, from which it sends as its congestion control allows. When this send queue fills up, DCCP rejects further packets with an EAGAIN error [The09]. Provided the queue is properly sized, this rejection provides a signal to the application that it needs to slow down. If DCCP is using CCID 3, the computed allowed sending rate can also be retrieved using a socket option [Ren07]. To convey this feedback to the media encoder, we added code to check for EAGAIN errors and report those as events to mediastreamer2, linphone's media streaming library. If CCID 3 is in use, we also occasionally report the current computed sending rate to mediastreamer2. We discuss these modifications in detail in section 3.2.3 below.

In addition to adding DCCP support, we also made a few other modifications to linphone in order to carry out our experiments. This mainly consisted of adding a video file input filter to be able to play a video file into linphone and a video file output filter to be able to record the video displayed by linphone. Adding these two filters allowed us to ensure identical inputs for all of our experiments and enabled us to capture linphone's video output in a manner that allowed easy offline video quality analysis.

To successfully stream HD video, we found it necessary to multi-thread the video encoding filter. Enabling HD video simply requires setting a #define in the source code; however, we found that attempting to stream HD video after setting the #define resulted in an extremely low frame rate and recurring warning messages from mediastreamer2. Mediastreamer2 utilizes a single thread for all media pipelines and iterates over all the filters; hence, each filter only has a small fraction of the time between frames to process each frame. Adding a second thread to the video encoding filter allowed it to spend the full amount of time encoding each frame, thereby enabling a full 25 frames per second.

3.2 Application Considerations for DCCP

In the course of adding DCCP support to linphone and our initial testing, we discovered a few unexpected issues as a result of the differences between UDP and DCCP.

Those considering migration to DCCP or designing a new application using DCCP should think about these issues.

3.2.1 Socket Queues

When using DCCP, the size of the socket queue turns out to be very important. Unlike UDP, where packets are send immediately, DCCP queues application packets and sends them as its congestion control allows. When this queue fills up, DCCP rejects new packets with an EAGAIN error [The09].

Media streaming applications typically deal with chunks of data (video frames or short chunks of audio data) that are sent at regular intervals, typically 10-40ms [WW08]. For video data in particular, these chunks can be tens or hundreds of packets in size. UDP will happily take all of these packets and blast them out onto the network in a short, high-rate burst, causing all sorts of problems for other network traffic and dramatically increasing the risk for queue overflow at nearby routers. DCCP however, will only queue as many packets as allowed by its queue size and then send them as allowed by its congestion control.

If the DCCP socket queue is smaller than a chunk of data, DCCP will reject packets that it may actually be able to send before the next chunk is ready. However, if the socket queue is too large, several chunks of data can be queued before DCCP rejects data and alerts the application to slow down. This dramatically increases the apparent delay between endpoints as well as reducing the promptness of application reactions to changing network conditions.

The ideal queue would accept exactly one chunk of data and discard any remaining packets when the next chunk is ready. Unfortunately, Linux only offers FIFO and priority queues at this time [Ren07]. Given those options, it seems best to select a FIFO queue whose size equals one chunk of data.

Compressed video, however, tends to exhibit dramatic variations in frame sizes. I-frames are much larger than P-frames which are larger than B-frames; a factor of four difference between I-frames and P-frames is not uncommon. In addition, image complexity and the amount of motion between frames create large differences even within these groups. This complicates determining the appropriate DCCP queue size significantly.

The algorithm we settled on for adjusting DCCP's send queue size was to double the queue size each time we encountered a video frame that was larger than the current queue size. Every thousand frames we determined the largest frame seen in that time and reduced the queue to that size if needed. This seems to maintain a reasonable balance between ensuring that we don't reject data unnecessarily and minimizing the amount of time frames spend in this queue.

3.2.2 Feedback to Application

DCCP provides feedback to the application about its sending rate by refusing to queue packets or, for some CCIDs, a socket option that returns the current allowed sending rate. These methods of feedback can be rather awkward for media streaming applications to utilize.

Using a getsockopt() call to get the current allowed sending rate and then adjusting the media sending rate based on that information is fairly simple. However, only CCID 3 currently offers such an option. CCID 2 has no such socket option.[7] In addition, the relevant socket option actually returns a structure containing information about a variety of important CCID 3 parameters. It is highly unlikely that another implementation of DCCP would include an identical socket option.

[7] Although such an option could be implemented rather easily. The allowed sending rate is simply $\frac{window\_size * avg\_pkt\_size}{RTT}$.

Further, experience has shown that CCID 3's computed sending rate varies wildly from one moment to the next. We only attempt to adjust the bandwidth on a 10% decrease or 20% increase and even then found it necessary to further limit the number of rate changes by enforcing a four sampling period wait time between adjustments.

While CCID 3 offers a relatively convenient socket option to get the allowed sending rate, CCID 2 has no such feature. The only feedback given by CCID 2 is its refusal to queue a new packet when the send queue is full. This has the major downside that an application has no idea when it could send *faster* than it is currently sending. Applications must, therefore, periodically probe DCCP for additional bandwidth. Hence, application authors need to implement a scheme where bandwidth is slowly increased as long as CCID 2 has not recently rejected packets.

Since linphone already has bitrate management built in for use with RTCP, we reused that mechanism to probe for additional bandwidth if ten RTCP report intervals pass without CCID 2 rejecting any packets. On a packet reject, we reduce the requested bitrate by 10%. We also combine packet rejects that occur within a single frame into one reduction event and implement a minimum time between bandwidth adjustments.

Reusing linphone's existing bandwidth management with CCID 2 has a significant complication. Linphone's bandwidth management uses an exponential, percentage-based increase to probe for available bandwidth. CCID 2, by contrast, increases its bandwidth in a linear manner. This mismatch can cause linphone to quickly overshoot CCID 2's available bandwidth. A linear bandwidth probe increase would avoid this tendency to overshoot, but would take longer to find CCID 2's sending rate.

We justify our decision by pointing out that an application converting to DCCP is very likely to reuse its existing code just like we did. Further, percentage-based increases and decreases, which are inherently exponential, are rampant in real-time, streaming media applications and their RTP congestion control schemes [Lin13, BDS96, LM03].

Altering that tendency will take time and effort and requires these programmers to have at least a basic idea of CCID 2's congestion control algorithm.

Using DCCP's refusal to queue packets as application feedback also complicates queue length management as mentioned above. Make the queue too large, and the application will not get its feedback until significantly after network conditions have changed. Make the queue too small, and the application will artificially limit its bandwidth.

3.2.3 Codec Rate Adjustment

Once an application receives feedback from DCCP about its allowed sending rate, the application needs to pass this information on to the media encoder. This can have unexpected side-effects if there are limitations on when an encoder can adjust its bitrate.

When mediastreamer2 decides to adjust the bitrate of an encoded media stream, it sets the new bitrate and then re-initializes the encoder, essentially creating a new MPEG-4 stream. Examination of the libav source code[8] shows that this is a libav limitation; libav was simply not designed with the ability for the user to adjust the target bitrate on the fly.

Re-initializing the decoder results in the next frame, the first frame in the new stream, being a new I-frame. This has to be the case because there is no previous frame in this new stream to base a P-frame off of. As mentioned before, I-frames are usually significantly larger than P-frames, often by a factor of four or more. This causes an interesting phenomenon where reducing the bitrate of the decoder actually results in sending an I-frame which is much larger than a P-frame at the previous bitrate would have been. While the average bitrate will be lower (the large I-frame is balanced by a very large number of small P-frames), the short-term bitrate actually increases.

[8] Available from git://git.libav.org/libav.git. Our copy pulled April 22, 2013.

This interacts badly with DCCP's rejection-based feedback mechanism. A reject results in a bitrate reduction which results in a new I-frame. This new I-frame is significantly larger than the P-frame that just caused a DCCP reject so it also triggers a reject which results in another bitrate reduction. This cycle will converge toward the minimum bitrate, and quality, for the video stream and may never stabilize if that minimum bitrate is close to DCCP's allowed sending rate.

In order to alleviate this problem, we were forced to limit the frequency with which bitrate updates were made. In this way, the video bitrate could properly average out before another change was made. We ran a variety of experiments to examine how this update interval related to video quality and present our results in section 4.3.

3.3 The Linux Kernel DCCP Implementation

The Linux kernel contains the only maintained implementation of DCCP that we are aware of. There have been several attempts to implement DCCP under BSD or in user space [Hag03, Phe08, Zol02]. All are now, unfortunately, unmaintained and abandoned.

DCCP support was added to the Linux kernel in version 2.6.14 in 2005 [Ker07]. However, at that time, the implementation was far from complete, as only CCID 3 was supported. A small team of developers continued to improve the implementation over time. CCID 3 continued to be the primary focus, but CCID 2 was added around 2008.

We began working with DCCP on Linux in the summer of 2010 and quickly noticed severe performance issues, particularly with CCID 2. By examining DCCP traces using tcptrace [Ost03] and a conversion utility we developed,[9] we were able to identify eight separate bugs in the Linux DCCP implementation. About half of these bugs resulted from the interactions between the DCCP protocol itself and the pluggable congestion control modules. Most of the other bugs were problems with the CCID 2 implementation. We

[9] dccp2tcp. Available from https://github.com/samueljero/dccp2tcp.

submitted these bug fixes back to the Linux development team, and they were included in Linux 3.2.[10]

For this work, we started with the default Ubuntu 12.04 LTS 3.2.0-39 kernel because we wanted our results to be representative of the DCCP implementation in most modern Linux kernels, not some customized, experimental DCCP kernel that no one other than Linux DCCP developers use. However, in the process of running experiments for this work, we discovered a bug in CCID 3's loss interval handling.

This bug is that CCID 3 does not update the length of the second loss interval until the loss starting the third interval. It should be updating the length throughout the entire loss interval and recomputing the loss event rate. Instead, the loss event rate stays constant throughout the second loss interval. We have observed this bug causing connections to maintain a very low sending rate for several minutes because the computed loss event rate is around one packet in every thirty. When the second loss finally occurs, starting the third loss interval, the computed loss event rate suddenly jumps to one packet in several thousand and the allowed sending rate increases dramatically.

We submitted a patch for this bug and had it accepted into the Linux DCCP testing tree [Jer13]. We expect this patch to make it into the mainline kernel eventually but do not have a definite timeline as of this writing. We also backported this patch to the DCCP module in our 3.2.0-39 kernel[11] and utilized this 3.2.0-39+CCID3_fix kernel for our experiments.

The Linux DCCP developers do maintain a DCCP testing tree where experimental DCCP features and new patches can be tested.[12] A variety of new features are included in this tree that are not yet considered stable enough for the mainline kernel. These include the two experimental CCIDs mentioned previously (numbers 4 and 5) and Explicit

[10] Released January 4, 2012 [Ker12].
[11] See Appendix B for this patch.
[12] This tree is located at http://eden-feed.erg.abdn.ac.uk/cgi-bin/gitweb.cgi?p=dccp_exp.git.

Congestion Notification (ECN) [RFB01] support. ECN support, in particular, should further improve overall DCCP performance once it is stable enough for inclusion in the mainline kernel.

3.4 Video Quality Analysis

In our experiments we are particularly interested in examining the quality of the video transmitted over DCCP versus UDP. While loss rate and throughput data are important, what really matters for an interactive streaming application, like linphone, is media quality.

Measuring media quality turns out to be rather complicated because what we really want to measure is the media quality *as perceived by the human visual or auditory system*, both of which exhibit a variety of nonlinearities and masking effects [WBSS04, CZM+10, WM08]. One could argue that the only truly "correct" manner to evaluate media quality is through subjective human evaluation. However, as such evaluation is expensive, time-consuming, and difficult, much work has gone into developing objective models that offer some approximation of media quality.

In the video space, no metric seems to have emerged as the clear standard. Most studies use the Peak Signal to Noise Ratio (PSNR) while admitting that it has only approximate correlation to human perception [CZM+10, WM08, GDK+05, VN04]. The International Telecommunication Union (ITU) has proposed a set of metrics for video quality measurement as ITU-T J.247 [Int08]; however, these metrics have not caught on in the research community, possibly because they are encumbered by patents.

In this work, we utilize PSNR and SSIM, two common quality metrics in the research literature. We discuss these metrics in detail below.

3.4.1 Peak Signal to Noise Ratio

Peak Signal to Noise Ratio (PSNR) is one of the most popular metrics for video or image quality analysis [WM08]. This is due in a large part to its simplicity. PSNR simply measures the degree of distortion between a reference image and a test image in a pixel by pixel manner. Higher values indicate less distortion and, presumably, better quality. Note that PSNR is what is known as a *full-reference* metric; the video under consideration is compared against the original source video to determine its quality.

PSNR is simply a logarithmic representation of the mean squared error between corresponding pixels in corresponding frames. This makes it incredibly easy to compute. Mathematically [VN04]:

$$PSNR = 20 * \log_{10}\left(\frac{max\_pixel\_value}{\sqrt{MSE}}\right)$$

Where MSE is the mean squared error:

$$MSE = \frac{\sum_{i=0}^{i=n-1}\sum_{j=0}^{j=m-1}[f(i,j) - F(i,j)]^2}{m * n}$$

Where f is the n x m source image and F is the n x m test image.

PSNR has a theoretically unlimited range. However, typical PSNR values range between 20dB and 40dB. 20dB is typically considered unwatchably poor quality while the differences between two images at 40dB are typically considered imperceptible to the human visual system [Goo05, GMM04, Sea04].

Note that PSNR is a frame-by-frame comparison between the reference video and the test clip. If these videos are not synchronized to have exactly corresponding frames, then the PSNR measurements will suffer significantly; one missing frame results in all later comparisons being off by one. Since lost or undecodable frames are occasionally expected in streaming video, this has to be compensated for in order to meaningfully utilize PSNR to measure video quality.

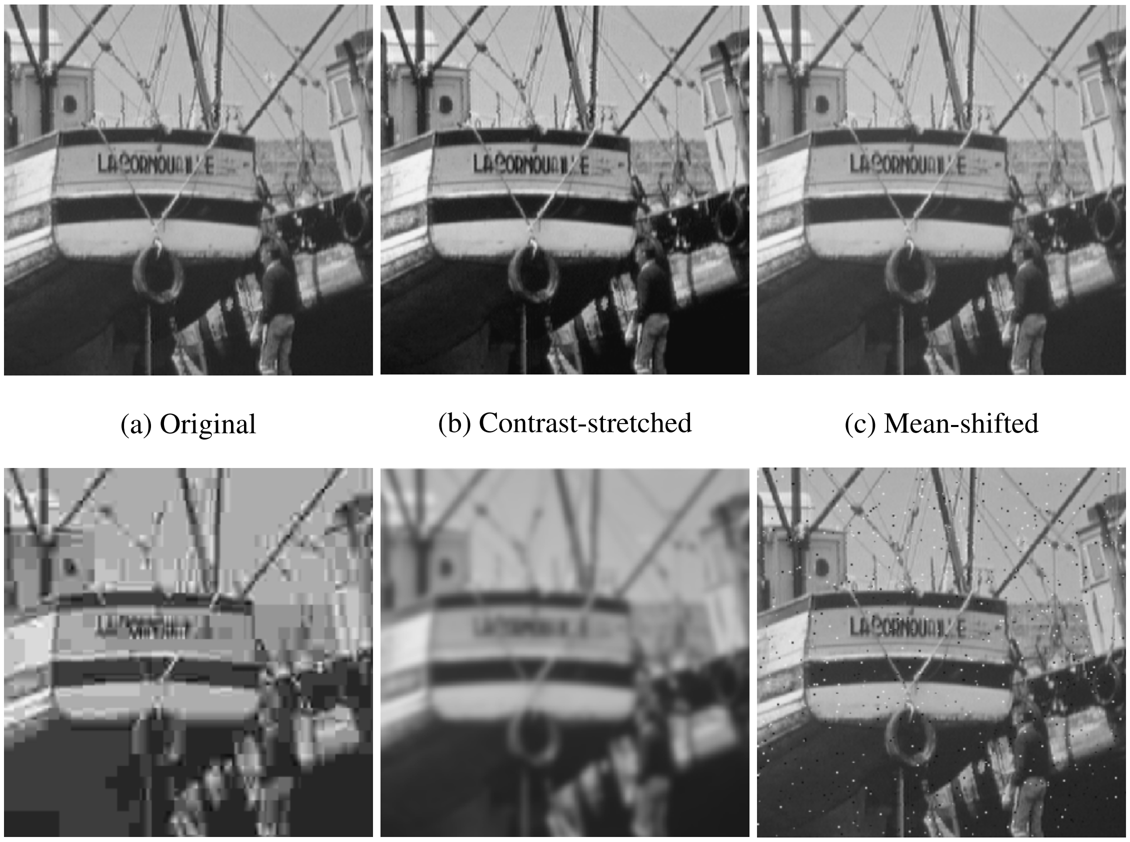
We compensated for this issue by having the video receiver compute a frame number from the RTP timestamp associated with each frame. Our video output filter then utilized this frame number to identify missing or duplicated frames and produce a frame synchronized output file. In place of a missing frame, we duplicated the previous frame. This approach emulates the common practice of continuing to display the prior frame when a loss occurs and has precedence in the literature [GMM04]. We utilized qpsnr [Ori10], an open source video quality analyzer, to do the actual frame-by-frame PSNR computation.

Unfortunately, while PSNR is a simple and easy to compute video quality metric, it is not a particularly good one. A variety of studies have shown that PSNR only loosely corresponds to quality as perceived by the human visual system [WBSS04, WM08, CZM+10]. This is because PSNR weights all distortions equally without taking into account how noticeable they are to the human visual system. To illustrate this, we have reproduced [WBSS04]'s figure 2 as our figure 3.1. This figure shows a variety of distorted images of various perceived quality levels, all with the same PSNR value.

3.4.2 Structural Similarity

Another common image or video quality metric is Structural Similarity (SSIM). This metric was proposed in [WBSS04] in part to compensate for some of the shortcomings of PSNR. SSIM works on the assumption that the human visual system is primarily interested in the structural information, like objects and their shapes, in an image. With that as a foundation, SSIM attempts to quantify distortions in this structural information [WBSS04].

Like PSNR, SSIM is a *full-reference* metric, requiring the original source video from which the test clip was produced. In addition, it performs a similar frame by frame



(a) Original (b) Contrast-stretched (c) Mean-shifted

(d) JPEG-compressed (e) Blurred (f) Salt-pepper contaminated

Figure 3.1: Comparison of images with different types of distortion and identical PSNR values. Figure from [WBSS04].

comparison between the reference and test videos, requiring frame synchronized video. SSIM values range between 0 and 1 with higher values indicating less distortion of structural information.

SSIM is computed mathematically as follows [WBSS04]:

$$SSIM = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

Where μ_x and μ_y are the average luminance values in the reference and test images respectively, and σ_x and σ_y are the corresponding standard deviations. σ_{xy} is the covariance between the reference and test images. C_1 and C_2 are constants to stabilize the division. They should be proportional to the square of the maximum pixel value [WBSS04].

The covariance can be computed as [WBSS04]:

$$\sigma_{xy} = \frac{1}{N-1}\sum_{i=1}^{i=N}(x_i - \mu_x)(y_i - \mu_y)$$

where x_i and y_i are the values of the *i*th pixel in the reference and test images respectively, and N is the number of pixels in each image.

While SSIM could be computed over the whole image, it is more useful to compute it over small blocks of the image and then average these values to produce a single SSIM value for each frame [WBSS04]. The utility we used to compute the SSIM of our test clips, qpsnr, computes the SSIM over each frame using 8x8 blocks in a tiled manner [Ori10].

Because SSIM is a newer quality metric and there are multiple ways to compute it over a given image, there is less agreement on what typical or reasonable SSIM values are. However, our experience is that values below 0.8 indicate very poor quality.

SSIM has been shown to be superior to PSNR in gauging the effect of at least some types of distortions [WBSS04] and has seen some acceptance in the literature [RPB+08, WM08]. However, PSNR is still the default video quality metric at this time.

3.5 Experiment Configuration

This section examines the details of our experiments and discusses our reasons for selecting these configurations.

3.5.1 Test Video Clips

We utilized two different video clips in our experiments representing two distinct types of material. The first was a 12 minute movie clip taken from the movie "Tears of

Steel" [Hub12], a science fiction/action short film.[13] Like most films of this type, this clip featured a variety of sudden scene changes and action sequences.[14] We were able to procure this film in the completely uncompressed yuv format with a resolution of 1920 x 1080 and a frame rate of 25 frames per second.

The second clip consisted of five minutes of videoconference material that we recorded ourselves. As is typical of most videoconference material, it contained no significant changes in scene and consisted primarily of talking-head-type video.[15] This clip was created with a resolution of 1920 x 1080 at 25 frames per second. We found it necessary, however, to utilize motion jpeg compression[16] because of resource limitations at capture time.

These two clips represent distinctly different types of video with distinct characteristics. The sudden scene changes and complex action sequences contained in our movie clip imply a very complex and extremely bursty video stream. This is because the sudden scene changes will generate additional I-frames, and the variations in complexity will affect the relative sizes of all frames. On the other hand, the lack of scene changes and relatively small quantity of motion in our videoconference clip implies a relatively simpler video stream that should be much smoother.

We consider both of these clips because both represent types of video that one may desire to transmit in real-time. Linphone is designed for videoconferences and that is probably the most obvious use for real-time video. However, applications may also desire to transfer complex, high-motion video in real-time. Remote rendered video games are just one example.

[13] We downloaded this movie from https://media.xiph.org/tearsofsteel/ as a series of png images, one per frame. We combined these png's into a yuv video file using the ffmpeg utility from libav [Lib13].

[14] This clip can be viewed at http://youtu.be/Yv71A73nx5E.

[15] This clip can be viewed at http://youtu.be/I4bqcPzea00.

[16] Motion jpeg compression simply applies jpeg compression to each frame independently [Wal91].

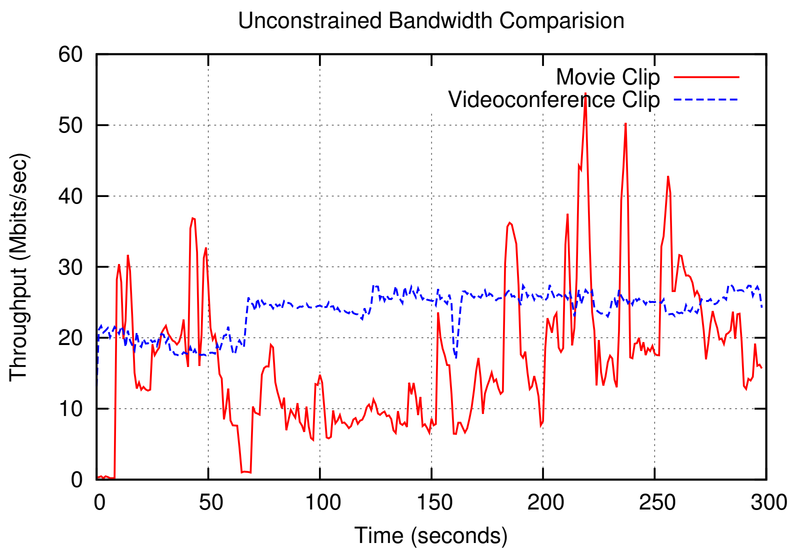


Figure 3.2: Bandwidth utilization for the first five minutes of our test clips in an unconstrained environment with no competition.

To examine the difference in video complexity between our clips, we utilize two different methods. The first is simply an analysis of the bandwidth used by our video clips in an unconstrained environment with no other network traffic. In such a situation, linphone's bitrate control will not adjust the encoder's target bitrate because there is no loss. Hence, the only factor influencing the bandwidth of the video will be the video encoder. We assume that higher bandwidth roughly translates to greater complexity. Figure 3.2 examines the first five minutes of our two video clips using this method.

Our movie clip exhibits much greater variation in its bandwidth and, hence, approximate complexity than our videoconference clip does. While our videoconference clip stays within a 10Mbit window over pretty much its whole length, our movie clip ranges from a few Kbits to 55Mbits, usually staying in the 8-30Mbit range. This implies not only a higher peak complexity but also much larger variations in scene complexity.

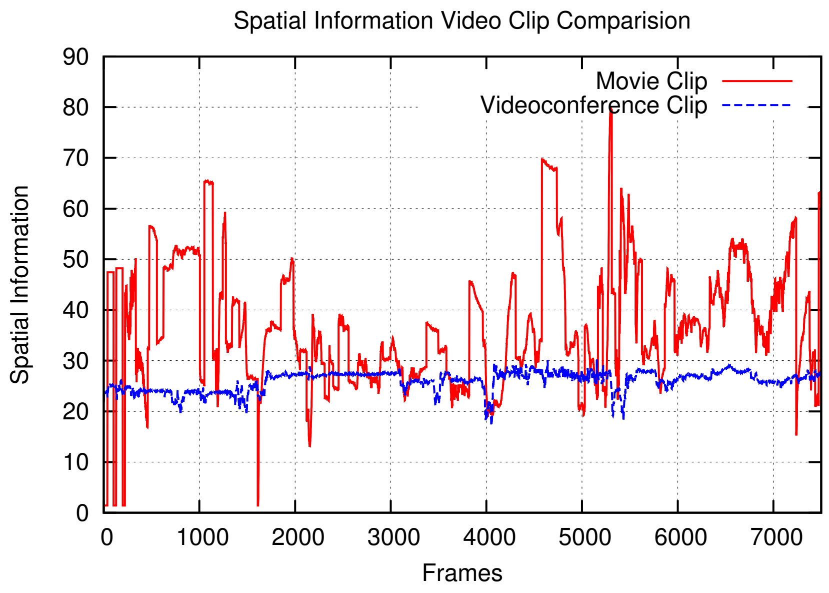
In addition to this rather approximate examination, we consider the video complexity of our clips using a set of metrics designed by the International Telecommunication Union (ITU) as ITU-T Recommendation P.910 for use in gauging video complexity [Int99]. Two separate metrics are specified, one considering the spatial complexity of each frame and one examining the temporal complexity between frames.

The Spatial perceptual Information measurement (SI) filters each frame through a Sobel filter[17] and then computes the standard deviation across the filtered image. This standard deviation is the SI measurement for the frame [Int99].

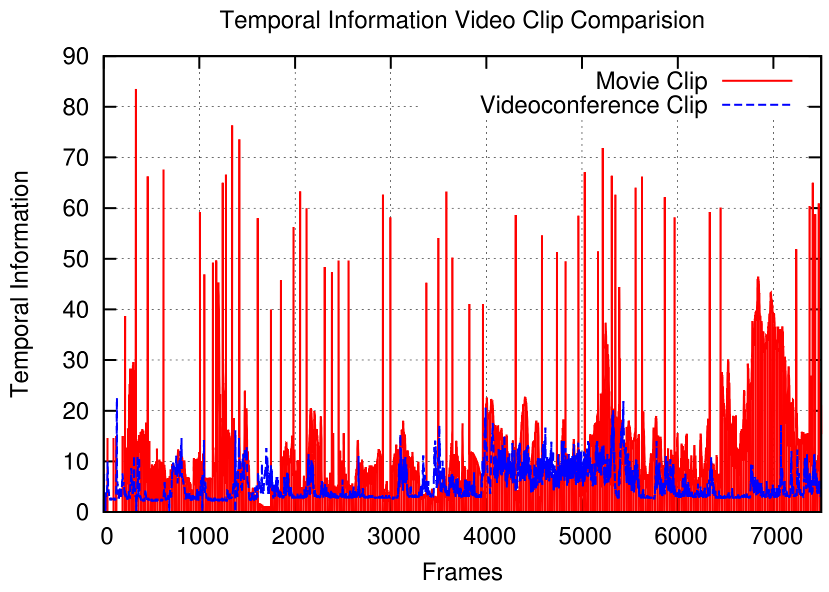
The Temporal perceptual Information measurement (TI) computes the difference between the luminance values of corresponding pixels in consecutive frames. The standard deviation of this difference across the entire image is the TI measurement for that frame [Int99].

Figures 3.3a and 3.3b graph SI and TI, respectively, over the first five minutes of our test clips. Notice that our movie clip not only exhibits much greater variation in its complexity than our videoconference clip but also has higher average complexity. The few points where our movie clip dips below our videoconference clip in spatial complexity correspond to title sequences. Also of note is the number of scene changes in our movie clip, as indicated by the number of sharp peaks in the temporal information graph.

[17] A convolution of two 3x3 kernels over the luminance component of the image. See [CPM07] for details.



(a) Spatial Perceptual Information



(b) Temporal Perceptual Information

Figure 3.3: Video complexity of the first five minutes of our test clips using the ITU-T P.910 video complexity metrics.

3.5.2 Testbed Configuration

Our first experimental setup was a simple “dumbbell” test network as shown in figure 3.4. Two machines were connected to a switch which was connected to a middle machine doing packet capturing and bridging and then to another switch. This switch was connected to two more machines. We ran a linphone videoconference between one pair of these machines while the other pair supplied background TCP traffic using Iperf [Nat10]. Iperf sends a single, continuous stream of TCP traffic from a sender to a receiver; to properly compete with our bidirectional videoconferencing traffic, we ran two Iperf tests in opposite directions.

The pair of machines running the linphone videoconference were quad-core AMD A8’s at 3.0GHz with 16GB of RAM running Ubuntu 12.04.2 LTS 64bit with kernel 3.2.0-39 and our patched DCCP module, as per section 3.3. Because we were playing video into linphone from files and recording the received video, as well as doing the normal video encoding and decoding for streaming, these machines needed significant

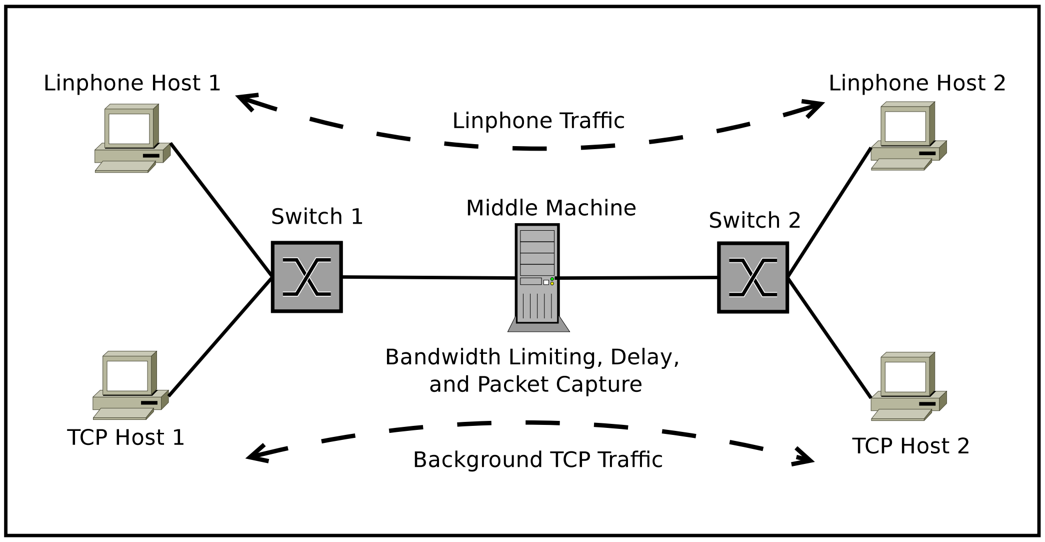


Figure 3.4: Our testbed environment

processing power. The pair of machines providing background traffic were hyperthreaded Intel Pentium 4's at 2.8GHz with 1GB of RAM each. These machines were running Ubuntu 12.04.2 LTS 32bit with kernel 3.2.0-38.

The middle box was a hyperthreaded Intel Pentium 4 at 3.0GHz with 1GB of RAM running FreeBSD 9.1 32bit and setup to bridge traffic between the two switches. We utilized ipfw [The13], the FreeBSD firewall utility, to artificially limit bandwidth and introduce delay on this link. Packet capturing was done just prior to this bandwidth limit using tcpdump [Tcp13].

In our tests, we limited the link bandwidth to 10Mbits/sec and introduced an artificial 10ms delay at the middle link, bringing the round trip time between our test machines to about 22ms. Since a linphone video conference typically uses between 8-10Mbits/sec, the bandwidth constraint of 10Mbits/sec forces the video stream to compete heavily with TCP, which enables us to meaningfully evaluate the fairness between linphone's video stream and TCP. The 10ms delay brings the round trip more in line with the traversal of a large local area network. We then ran a linphone videoconference between one pair of machines while also running Iperf in both directions between the other pair of machines. This roughly simulates a severely congested local area network.

Our tests were operated from a shell script on one of the machines that launched ssh sessions to initiate the iperf connections and the linphone video conference. This ensured close synchronization between linphone and iperf.

We performed tests with both our 12 minute movie clip and our 5 minute videoconference clip over UDP, DCCP CCID 2, and DCCP CCID 3. Each of these configurations was repeated ten times. The repetitions occurred in two batches, first a batch of three and then a batch of seven, separated by several weeks and represent samples over at least a five hour period. In total, 2 hours of movie clip video and 50 minutes of videoconference material were transferred per protocol.

3.5.3 Short Distance Internet Configuration

Our second experimental setup consisted of the same pair of quad-core AMD A8's, located in our lab, obtaining IPv6 connectivity via 6in4 tunnels from Hurricane Electric [Hur13]. The remote endpoint for both tunnels was at the same site in Virginia. Using this setup, we could route IPv6 traffic from one machine in our lab to Virginia using one tunnel and then immediately back to the other machine in our lab using the other tunnel.

Since the traffic in these tunnels traveled the standard IPv4 Internet between our lab in Ohio and Virginia, this setup allowed us to test the performance of DCCP and UDP in a fairly short distance Internet environment. The round trip time in this configuration was roughly 56ms, a fairly typical value for short distance Internet connections. There are no bandwidth limits placed on Hurricane Electric tunnels; the only bandwidth limits in this test come from any traffic shaping that may be occurring on the path and the congestion control algorithm under test. In practice, we observed bandwidth in excess of 30Mbits/sec between our test systems.

Our tests in this environment consisted of running a linphone videoconference between these machines while simultaneously running Iperf [Nat10] tests in both directions and measuring ping times. The Iperf tests ran on the same machines as linphone in order to have access to the IPv6 tunnels and gauge what a reasonable share of the bandwidth was in this environment.

Much like our testbed environment, we ran tests with both our 12 minute movie clip and our 5 minute videoconference clip. For each of these videos, we ran tests using DCCP CCID 2, DCCP CCID 3, and UDP. Each of these configurations was repeated ten times. The repetitions occurred in two batches, first a batch of three and then a batch of seven, separated by several weeks and represent samples over at least a five hour period.

3.5.4 Long Distance Internet Configuration

Our last experimental setup consisted of these same two machines, again using IPv6 tunnels. However, this time one endpoint was in Virginia and the other was in California. Using this setup, we could send IPv6 traffic from one machine in our lab to Virginia using one tunnel, then across the IPv6 Internet to California, and then back to the other machine in our lab via the other tunnel.

The round trip time in this configuration was roughly 178ms, which is quite good for traveling most of the way across the country and back. The purpose of this experimental configuration was to examine the performance of DCCP and UDP for long distance Internet connections and to examine how performance changes relative to our previous, shorter round trip time, tests. Once again, there are no explicit bandwidth limits in this configuration.

Our tests again consist of running a linphone videoconference while also running Iperf [Nat10] in both directions and recording the ping times between machines. Because of the longer round trip time in this configuration, we found it necessary to introduce a constraint on how often bitrate updates could be performed. To determine this parameter, we performed a set of tests, using our movie clip, with bitrate update intervals between 40ms (framerate) and 6000ms. We ultimately settled on 3000ms for CCID 2 and 1000ms for CCID 3.

With the bitrate update interval determined, we ran tests using both our 12 minute movie clip and our 5 minute videoconference clip over DCCP CCID 2, DCCP CCID 3, and UDP. Each of these configurations was repeated ten times using a batched pattern similar to our previous configurations. Our results are presented in the next chapter.

4 Results and Discussion

This chapter presents our experimental results and offers analysis and discussion. We have grouped our results based on the experimental setup and video clip used.

4.1 Testbed Experiments

We start by examining our testbed experiments. This simple environment should provide us with a good understanding of DCCP's basic behavior. In addition, this environment is, by design, our most congested configuration, allowing us to observe how DCCP and UDP/RTP react to extreme congestion.

4.1.1 Movie Clip

Recall that our testbed environment was a standard "dumbbell" network with two machines on either end connected to a switch and a middle machine doing packet capturing and bandwidth limiting. Our tests consisted of a linphone video conference between two machines while the other two machines supplied background TCP traffic. The middle machine capped the bandwidth at 10Mbits/sec and introduced enough delay to make the round trip time about 22 milliseconds. Each test ran for 12 minutes, the length of our movie clip.

Figures 4.1 and 4.2 show Cumulative Distribution Functions (CDFs) of the received video quality in our tests using PSNR and SSIM, respectively. These graphs plot video frame quality versus the probability of receiving video frames with up to that quality. Each of these cumulative distribution functions is an average of the CDFs for 10 separate test runs, or 2 hours of video. The error bars shown at 0.1 probability intervals indicate the 95% confidence interval.

As can be seen, both DCCP CCIDs achieve better video quality than UDP by a consistent 3-5dB PSNR. CCID 3 typically performs better than CCID 2. Taking the

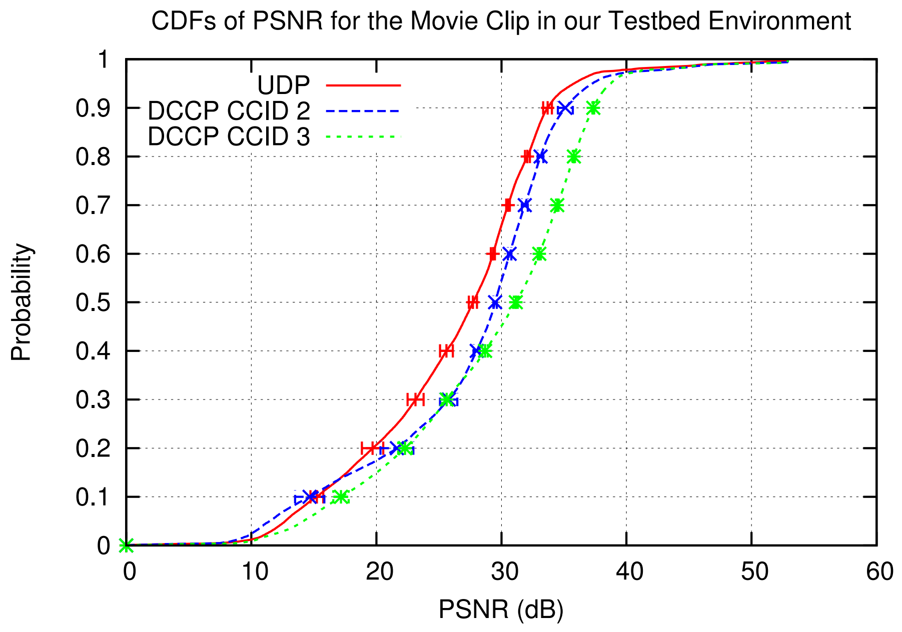


Figure 4.1: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by PSNR, for our movie clip in our testbed environment. Error bars indicate the 95% confidence interval.

20-40dB PSNR range as the range of reasonable video quality,[18] UDP spends about 21% of its time sending unwatchable video while CCID 2 sends unwatchable video about 18% of the time and CCID 3 only 16% of the time. This is a fairly modest, but not insignificant, improvement.

Note also that CCID 2 varies between being similar to UDP at higher probabilities, then achieving identical performance to CCID 3 around a probability of 0.3 and then swinging back to be worse than UDP below 0.1 probability. The first swing occurs because, as we will discuss in detail later on, linphone is much more conservative in increasing its sending rate when using CCID 2. This results in CCID 3 outperforming CCID 2 throughout the upper portion of these graphs. Towards a probability of 0.3, the

[18] See section 3.4.1 and [Goo05, Sea04].

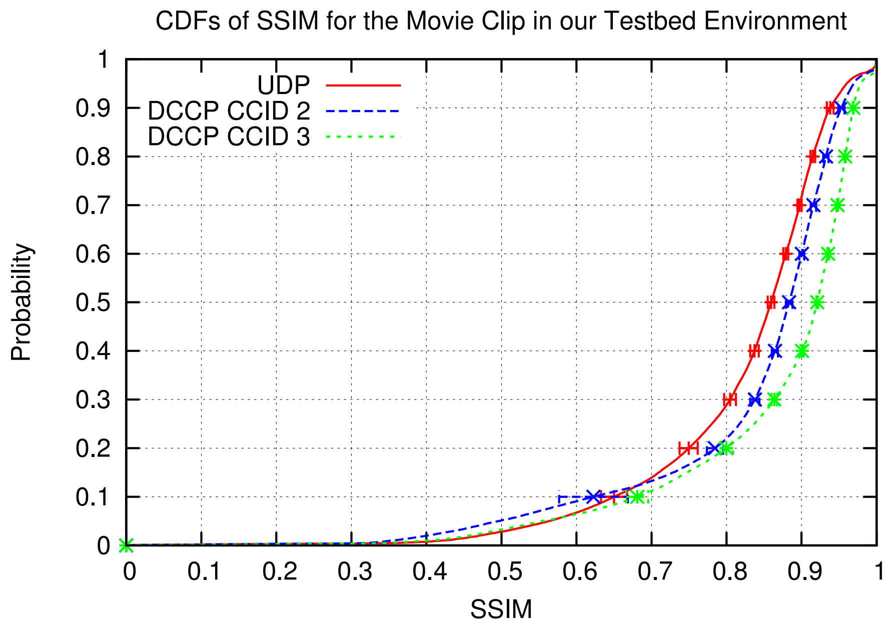


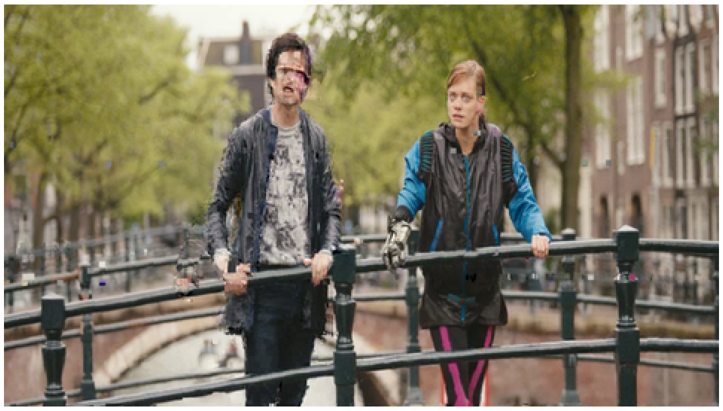
Figure 4.2: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by SSIM, for our movie clip in our testbed environment. Error bars indicate the 95% confidence interval.

better congestion control of both CCIDs shows its advantage and even CCID 2 does much better than UDP. CCID 2's final swing back below UDP occurs because CCID 2 is much harsher in its reaction to congestion than CCID 3 and this can cause additional video artifacts as linphone tries to catch up.

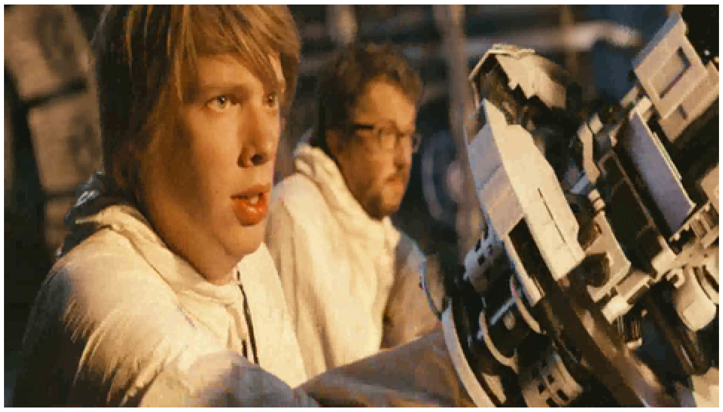
The SSIM CDFs, figure 4.2, tell a similar story. DCCP achieves better quality than UDP over nearly the whole graph, with CCID 3 performing better than CCID 2. Similarly to the PSNR CDFs, we see CCID 2 swing from similar to UDP to nearly matching CCID 2 and then back.

To help clarify the relationship between PSNR, SSIM, and perceived video quality, figure 4.3 shows several video frames with their PSNR and SSIM values.[19] You will notice

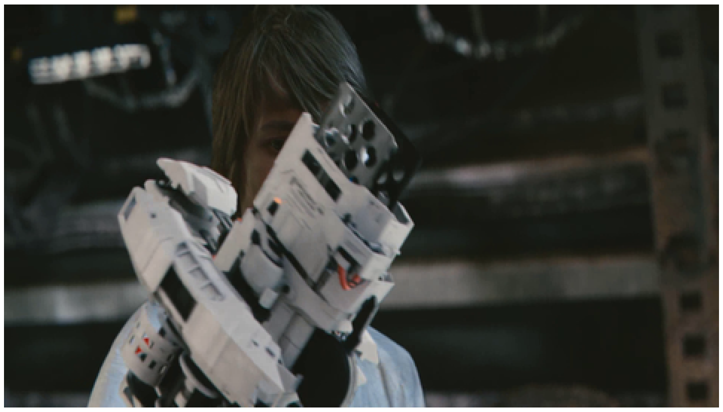
[19] An example of the video received over UDP can also be viewed at http://youtu.be/KXkpTm6PGaQ.



(a) 20dB PSNR, 0.73 SSIM



(b) 30dB PSNR, 0.87 SSIM



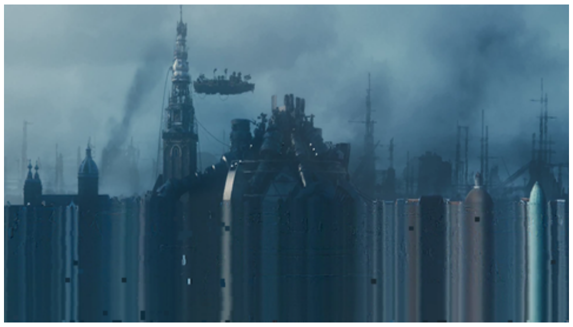
(c) 40dB PSNR, 0.98 SSIM

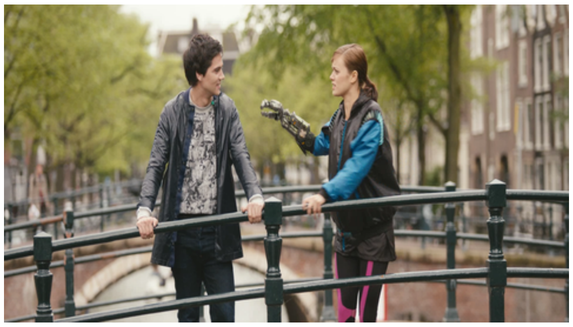
Figure 4.3: Examples of the quality implied by different PSNR and SSIM values. These frames are taken from our UDP movie clip tests in our testbed environment.

that the 20 dB PSNR frame exhibits severe distortion while the 30 dB frame exhibits only a few slight compression artifacts, and the 40 dB frame contains no noticeable issues.

In addition to examining quality using metrics like PSNR and SSIM, it is useful to examine the types of visual artifacts that commonly occur in received video and how they differ between DCCP and UDP. These common artifacts include partial I-frames, lost motion vector data, and undecodable frames. Partial I-frames occur when a large contiguous chunk of an I-frame is lost, often because DCCP overflowed its send queue or because a router queue overflowed and dropped a contiguous set of packets. Figure 4.4b shows an example of a partial I-frame. Because P-frames are derived from the information

(a) Original


(b) Corrupted (Partial I-frame)


(c) Original

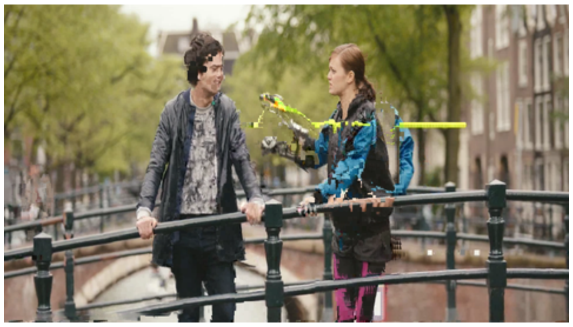

(d) Corrupted (Lost Motion Vector Data)

Figure 4.4: Examples of visual artifacts resulting from packet loss.

in the preceding I-frame, this visual artifact propagates through the succeeding P-frames up until the next I-frame.

Lost motion vector data occurs when the motion vector data in a P-frame is lost either because of a network drop or DCCP send queue overflow. This usually results in the doubling or distortion of the moving objects in a frame as seen in figure 4.4d. These artifacts persist until overwritten by later motion vector data or the next I-frame. Fortunately, areas of motion tend to be in motion over several frames so later motion vectors often correct these artifacts fairly quickly.

The final visual artifact that occurs in received video is undecodable frames. These are frames that the MPEG-4 decoder, for whatever reason, is unable to decode so they are simply skipped. A single undecodable frame is usually invisible to the human visual system [CZM+10]; however, it represents a severe error in the bitstream, typically the loss of a critical header. MPEG-4 utilizes resynchronization markers and reversible variable length codes to recover from most errors. Further, texture or motion vector data can simply be skipped over if it cannot be decoded. MPEG-4 headers, like the VOP header or the GOV header, however, cannot simply be skipped if they cannot be decoded; their loss renders the frame undecodable. Such headers usually occur at the beginning of the frame, and their loss tends to indicate severe congestion since the last frame still has not drained from the network.

DCCP, particularly CCID 2, results in noticeably more partial I-frames than UDP and distinctly fewer motion vector data losses. This is because I-frames are much larger than P-frames and so are significantly more likely to overflow DCCP's send queue and be rejected. Table 4.1 shows that both DCCP CCIDs achieve significantly reduced network losses compared to UDP but reject a significant number of packets because of send queue overflow. The reduced network losses will reduce the number of short, random losses while the send queue overflows will tend to occur in the larger I-frames. This will cause a

Table 4.1: Packet Statistics for our Movie Clip in our Testbed Environment

	Sent Packets	Received	% Received	Lost	% Lost	Rejected	% Rejected
UDP	159713	143374	89.77%	16339	10.23%	0	0.00%
CCID 2	245283	164856	67.21%	952	0.39%	79473	32.40%
CCID 3	247909	235735	95.09%	7111	2.87%	5063	2.04%

predominance of partial I-frames, particularly for CCID 2, where rejected packets outnumber network losses by about two orders of magnitude.

The more frequent occurrence of partial I-frames produces rather hard to characterize effects on the received video. The loss of motion vector data usually results in some sort of doubling of the moving component, as in figure 4.4d, making it very difficult to follow the motion and understand what is going on in the scene. By contrast, the occasional partial I-frame can be preferable because the scene is either consistent or not present. This is especially true when using DCCP, where a rejected packet will quickly result in a rate reduction and a new I-frame, which will refresh the scene.

In addition to more partial I-frames, DCCP also results in more undecodable frames than UDP, as table 4.2 shows. Streaming video over CCID 3 results in a minor increase in

Table 4.2: Frame Statistics for our Movie Clip in our Testbed Environment.
This table shows the total number of I-frames sent and the total number of undecodable frames received as well as percentages for each.

	Total Frames	I-frames	% I-frames	Undecodable	% Undecodable
UDP	17957	375	2.09%	629	3.50%
CCID 2	17953	1398	7.79%	1685	9.39%
CCID 3	17961	1566	8.72%	869	4.84%

undecodable frames over UDP while using CCID 2 results in nearly three times as many. DCCP's send queue is a simple FIFO queue[20] so attempting to send a frame on a nearly full queue, which could happen because the previous frame was a large I-frame or because DCCP just slowed down, will result in DCCP rejecting almost all packets in the frame, including those containing the critical frame headers necessary for decoding. Fortunately, the human visual system is not particularly sensitive to frame rate so end users will not usually notice the occasional missing frame [CZM+10].

It is interesting to note that CCID 2 achieves better video quality despite its miserable 32% packet rejection rate and nearly 10% undecodable frame rate. This makes sense since a loss will result in CCID 2 suddenly halving its sending rate; the first indication the application will have of this lower sending rate is the send queue overflowing. Even presuming the application reacts to this overflow instantly, the rate change will not take effect until the *next* frame. Worse, this next frame will be a large I-frame. This will result in the loss of a very large chunk of a frame; whatever makes it through is unlikely to be decodable.

CCID 3 avoids a large rejection rate by its TFRC design and sending rate socket option. Part of the design goal of TFRC was to avoid the sudden rate changes involved in AIMD congestion control, like CCID 2. Since CCID 3 gets feedback once each round trip time, which in this environment is very close to the time between frames, a packet loss will result in a small reduction in sending rate during the next frame instead of a drastic halving. In addition, CCID 3 exposes its current sending rate via a socket option allowing the application to detect the slow down and reduce its sending rate before the send queue overflows, avoiding a massive burst of loss. This socket option also allows the application to speed up as soon as CCID 3 increases its sending rate.

[20] See section 3.2.1 for a detailed discussion of DCCP's send queue.

CCID 3, however, suffers an almost 3% network loss rate, as shown in table 4.1, compared to CCID 2's 0.39% network loss rate. While it seems reasonable that CCID 3 would have a higher loss rate than CCID 2 because it is designed to be somewhat less harsh in reducing its throughput, an order of magnitude difference in loss rate seems higher than expected.

We utilize Mathis's model of TCP throughput from [MSMO97] as a quick sanity check on CCID 3's behavior. With a network loss rate of 3% and an round trip time of 20ms, this model indicates that TCP would achieve roughly 3.3Mbits/sec in this situation. That lines up nicely with CCID 3's average throughput of 3.52Mbits/sec. Since CCID 3 is designed to compete reasonably fairly with TCP, it would appear that CCID 3 is behaving reasonably. Given that CCID 2 only achieves 2.37Mbits/sec with a 0.39% loss rate, this analysis indicates that CCID 2 is not limited by network loss rate. We will examine this issue in great detail later on.

We now turn to consider another important piece of DCCP's performance; its throughput relative to UDP. Figure 4.5 shows representative examples of the throughput achieved by UDP, CCID 2, and CCID 3 while streaming our movie clip. The most obvious feature of this graph is the large spike in UDP's throughput at the beginning of the clip. This occurs because linphone initializes the video encoder to the maximum allowed bitrate. This maximum bitrate is user configurable, but is hidden deep within the application settings. We used a value of 100Mbits/sec in all of our experiments so that it would not interfere with the congestion control algorithms. While for this configuration that is much too high, we argue that a maximum bitrate much greater than the network could sustain is a common case; users are not likely to adjust this setting based on the network environment they are in and application authors will not artificially limit how well their applications can perform.

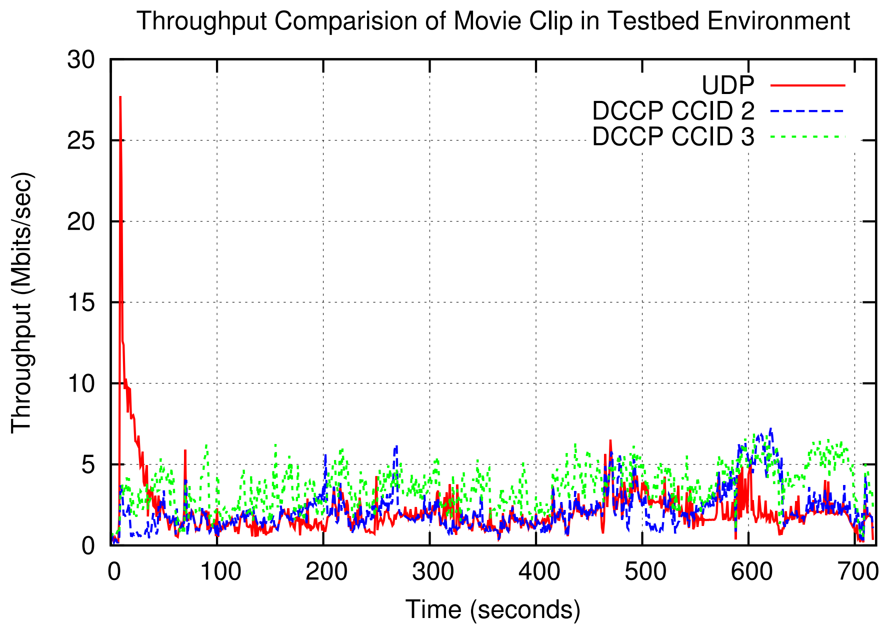


Figure 4.5: Throughput achieved by UDP, CCID 2, and CCID 3 over our 12 minute movie clip in the testbed environment.

This large spike, then, represents the time it takes for linphone's UDP/RTP congestion control to reduce the codec's bitrate to an acceptable level. This reduction takes roughly one minute; far too long for a single application to be congesting the network. Any other network traffic unfortunate enough to be already running when such a video stream starts will experience terrible throughput for well over a minute. If this other traffic were interactive, another video stream for instance, the user impact would be substantial. DCCP, by contrast, exhibits no such spike because both CCID 2 and CCID 3 utilize a form of slow start, a very low initial sending rate with an exponential increase as the connection continues. This algorithm enables DCCP to quickly finding the optimal sending rate without drastically exceeding it.

In order to examine throughput behavior once the connection enters a steady state, figure 4.6 shows a close up of the same throughput graph, focusing on the 0-9Mbit/sec

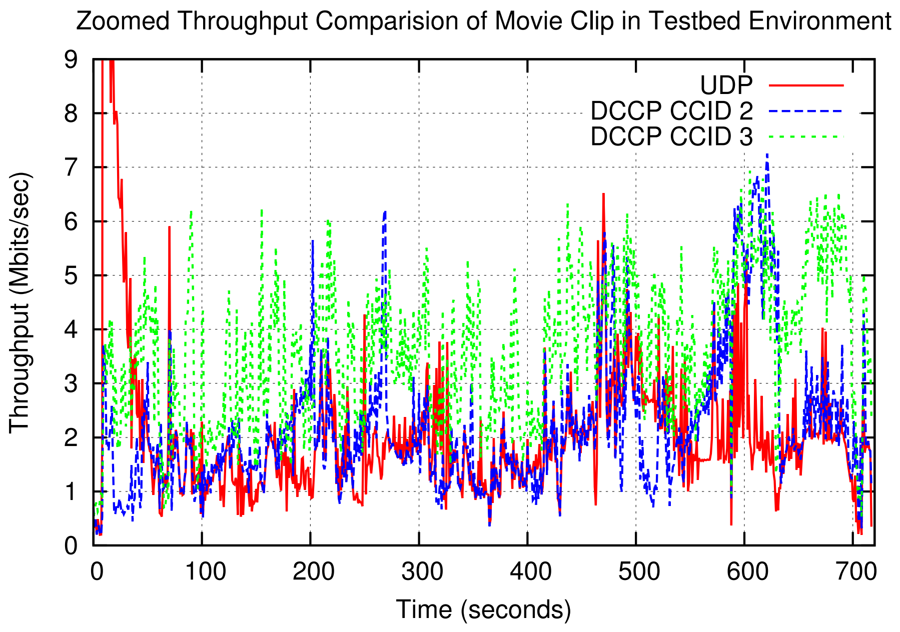


Figure 4.6: More detailed look at the variation in throughput of UDP, CCID 2, and CCID 3 using our movie clip in our testbed environment.

range. Interestingly, this figure shows that CCID 2 and UDP actually exhibit very similar throughput patterns while CCID 3 has a much more volatile rate with a higher average. This more volatile sending rate occurs because an application using CCID 3 can determine its exact allowed sending rate, allowing the application to follow this rate quickly as it varies. Both CCID 2 and UDP basically guess their allowed sending rate by slowly increasing their bitrate when they have not observed lost or rejected packets for quite a while. This results in a much smoother increase but a reduced ability to take full advantage of available bandwidth.

Distinctly related to the relative throughput of UDP and DCCP is the idea of fairness to other network traffic. Fairness is a crucial requirement for network traffic, particularly high bandwidth traffic like these video conferences, and is one of the main goals of congestion control.

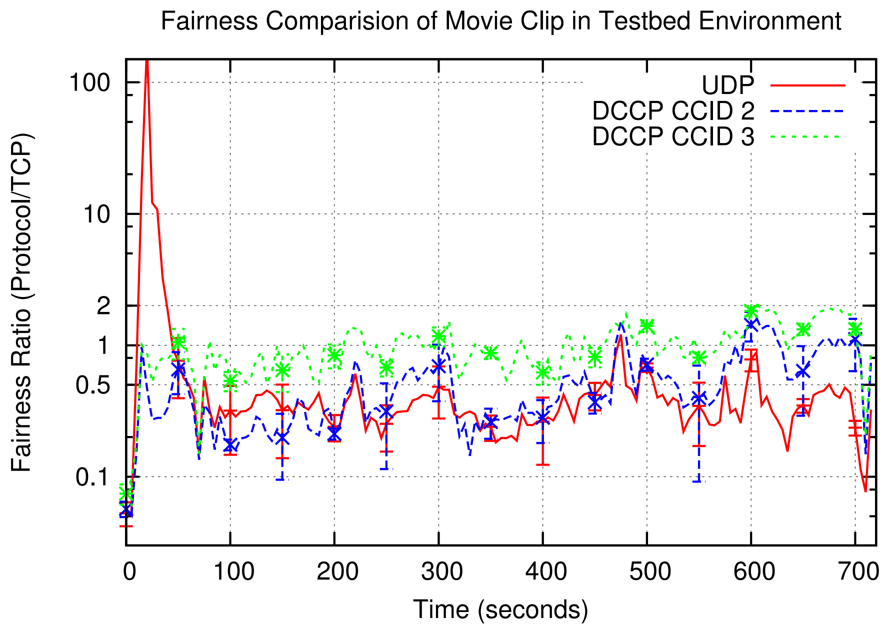


Figure 4.7: Average fairness of UDP, CCID 2, and CCID 3 to TCP over our 12 minute movie clip in the testbed environment. Error bars indicate the 95% confidence interval.

Figure 4.7 examines the average fairness of UDP, CCID 2, and CCID 3 to TCP over all our experiments with the movie clip in this environment. The error bars at 50 second intervals indicate the 95% confidence interval. The fairness metric used here is a simple ratio of *protocol_throughput*/*tcp_throughput*, so 1 is perfect fairness, 2 indicates that *protocol* is using twice as much bandwidth as TCP, and 0.5 indicates the opposite. Because of the range of fairness values over these connections, we plot the fairness ratio on a log scale.

The first thing to notice from this graph is the massive unfairness that linphone's UDP/RTP congestion control exhibits at the beginning of the connection. This corresponds with the our observations from the throughput graphs. Apparently, linphone's UDP/RTP congestion control allows the videoconference to overwhelm any competing traffic at the beginning of the connection. Worse, UDP/RTP takes nearly 50 seconds to get this behavior under control.

The Internet community has traditionally defined acceptable fairness between two network flows to be achieving throughput within a factor of two of each other [WH06, FHPW08]. From figure 4.7 it is clear that UDP, after the initial 60 seconds, is less aggressive than needed to achieve reasonable fairness with TCP. CCID 2 is in a similar boat, achieving reasonable fairness only occasionally. CCID 3, meanwhile, maintains much more even competition with other network flows. In fact, over all of our tests, CCID 3 maintained an average fairness ratio of 0.91, well within acceptable fairness.

For our movie clip, both DCCP CCIDs achieve better video quality than UDP. When visual artifacts do occur, DCCP tends to cause partial I-frames while UDP usually experiences lost motion vector data. CCID 2 manages to achieve better quality than UDP despite rejecting nearly a third of all packets that linphone tries to send. We have also observed that linphone's UDP/RTP congestion control causes a very large throughput spike on connection startup.

4.1.2 Videoconference Clip

We now turn to consider our videoconference clip in our testbed environment. These experiments follow exactly the same procedure as our movie clip experiments: "dumbbell" testbed network with a 10Mbit/sec bandwidth constraint at the middle node. One pair of machines runs a linphone videoconference with our video clip while the other pair supplies background TCP traffic using iperf. The only difference is that this video clip is only five minutes long.

Figures 4.8 and 4.9 show cumulative distribution functions of the received video quality over UDP, DCCP CCID 2, and DCCP CCID 3 using the PSNR and SSIM metrics. These graphs are average CDFs for ten experiments and contain error bars indicating the

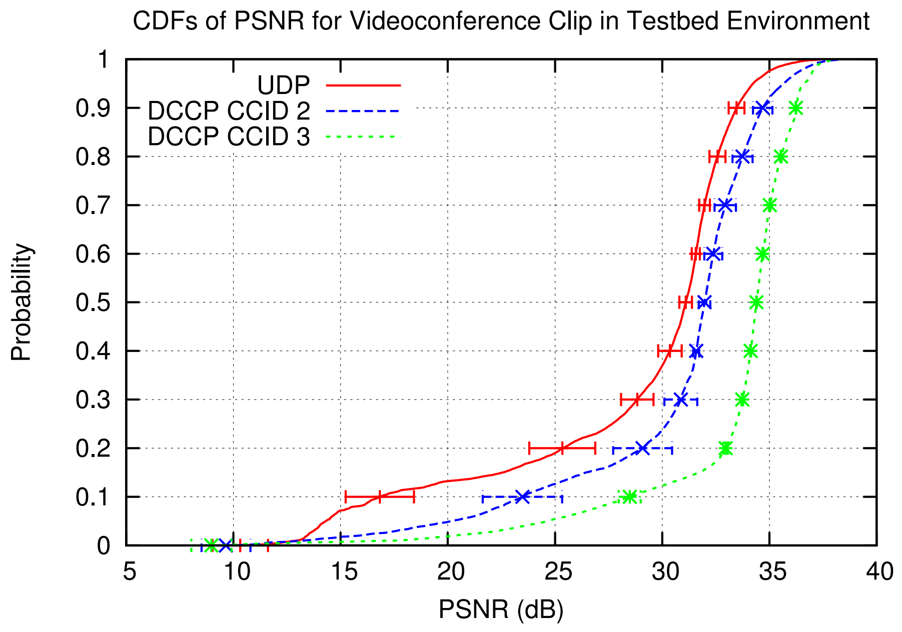


Figure 4.8: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by PSNR, for our videoconference clip in our testbed environment. Error bars indicate the 95% confidence interval.

standard deviation at 0.1 probability intervals. Notice that both DCCP CCID's achieve better quality than UDP, with CCID 3 performing the best. This indicates that DCCP's more responsive congestion control provides noticeable improvements in this environment. DCCP's maximum improvement over UDP occurs in the low to middling quality range, where DCCP exhibits a much lower and sharper "knee" that UDP does. DCCP is at least three times less likely to deliver unwatchable video (<20 PSNR) and increases the percentage of video above 30 PSNR by either 10 or 22 percentage points depending on CCID.

Interestingly, the SSIM CDFs show a much sharper divergence between UDP and DCCP in the bottom 10%. This implies that UDP suffers from a loss of primarily structural information at these poor qualities while DCCP does not. This is consistent with

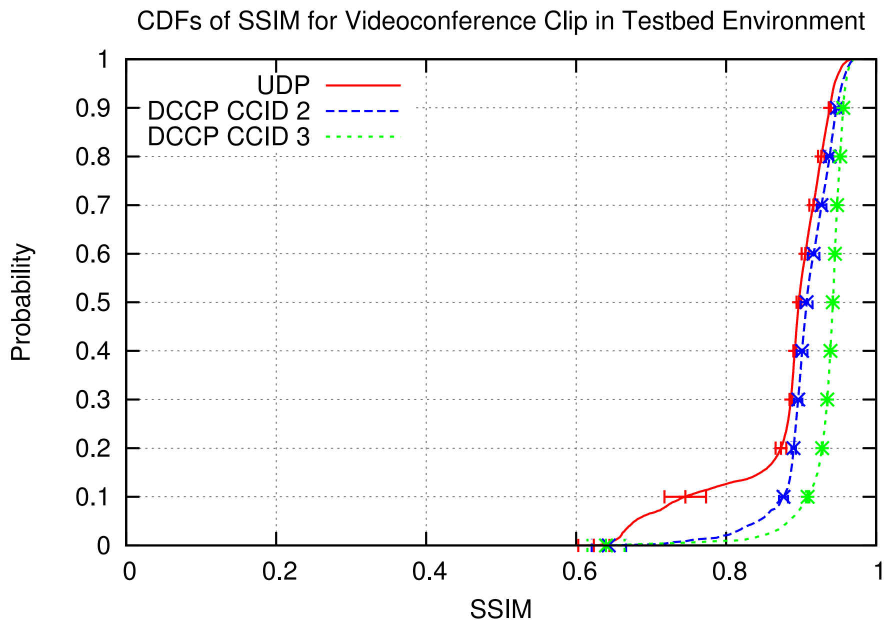
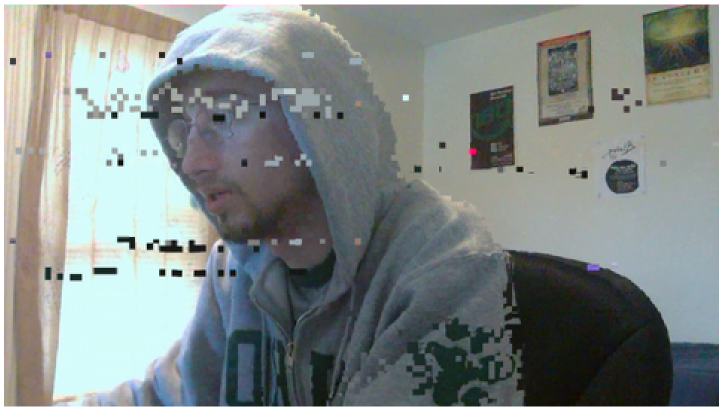


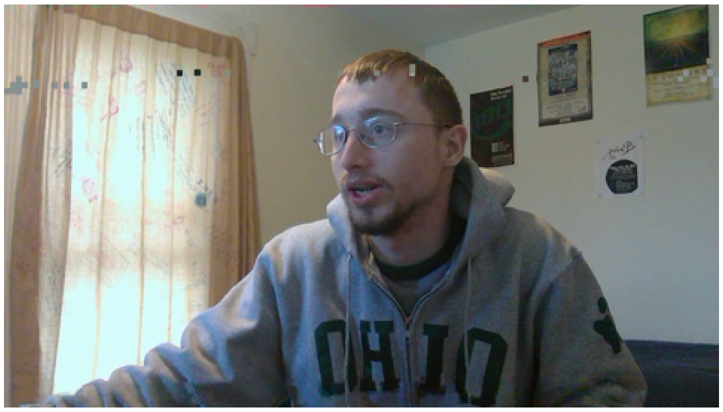
Figure 4.9: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by SSIM, for our videoconference clip in our testbed environment. Error bars indicate the 95% confidence interval.

a loss of motion vector information from random, network losses, resulting in distortion and duplication of the moving objects.

Figure 4.10 shows several video frames with their PSNR and SSIM values. You will notice that the 20 dB PSNR frame exhibits significant corruption while the 30 dB frame exhibits only a few slight minor artifacts, and the 37 dB frame contains no noticeable issues. Notice also that 0.8 SSIM is very poor quality while an SSIM value of 0.97 indicates no noticeable artifacts.

Comparing figure 4.8 with figure 4.1, the CDFs of PSNR for our movie clip, we observe that overall video quality for all protocols is higher for our videoconferencing clip than our movie clip. Given that videoconference material is distinctly less complex than


(a) 20dB PSNR, 0.80 SSIM


(b) 30dB PSNR, 0.93 SSIM

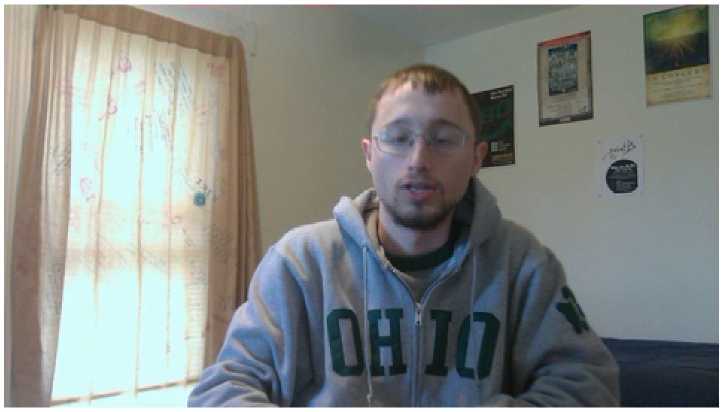

(c) 37dB PSNR, 0.97 SSIM

Figure 4.10: Examples of the quality implied by different PSNR and SSIM values. These frames are taken from our UDP videoconference clip tests in our testbed environment.

multi-scene, action movie material this makes sense. For a given allowed sending bitrate the lower complexity video can be encoded at higher quality.

It is interesting to note that both CCIDs achieve the same peak video quality, but diverge rapidly. This at least partly due to the fact that CCID 2 has no way to alert the application to slow down before the sending queue overflows and visual artifacts are induced. In fact, we observe that CCID 2 rejects 9.89% of all packets that the application tries to send. CCID 3, as mentioned previously, makes its computed sending rate available to applications via a socket option allowing them to adjust their rate without the send queue overflowing.

In addition, this socket option means that linphone knows exactly when it is allowed to speed up when using CCID 3 while it has to heuristically guess when using CCID 2. This results in linphone sending roughly 1.5Mbit/sec faster on average when using CCID 3 than when using CCID 2. Hence, CCID 2 achieves lower quality because it is sending at a lower bitrate. This further contributes to the divergence between CCID 2 and CCID 3.

Having examined the differences in video quality between UDP, CCID 2, and CCID 3, we now turn to consider the network throughput achieved by these protocols and how it varies with time. Figure 4.11 plots throughput over time for a set of example video streams over UDP, CCID 2 and CCID 3 with our videoconference clip in this environment. Just as in our movie clip experiments, UDP exhibits a large spike in sending rate for the first minute of the connection. This occurs as linphone's UDP/RTP congestion control converges to an acceptable sending rate for the network conditions. As before, this large burst of throughput posses a significant problem for fairness with competing traffic.

Figure 4.12 shows a close up of the 0-7Mbit/sec portion of this same example throughput graph to better illustrate the throughput of these protocols after UDP stabilizes. From this graph we observe that CCID 3 not only sends faster in general but also changes its sending rate much more rapidly than CCID 2. Since CCID 3's TFRC congestion

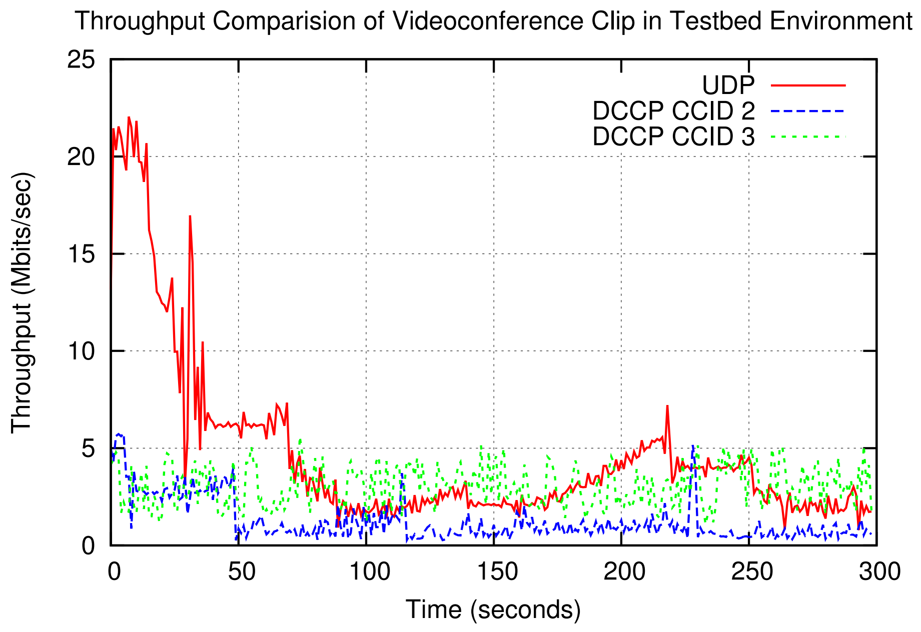


Figure 4.11: Throughput achieved by UDP, CCID 2, and CCID 3 over our 5 minute videoconference clip in the testbed environment.

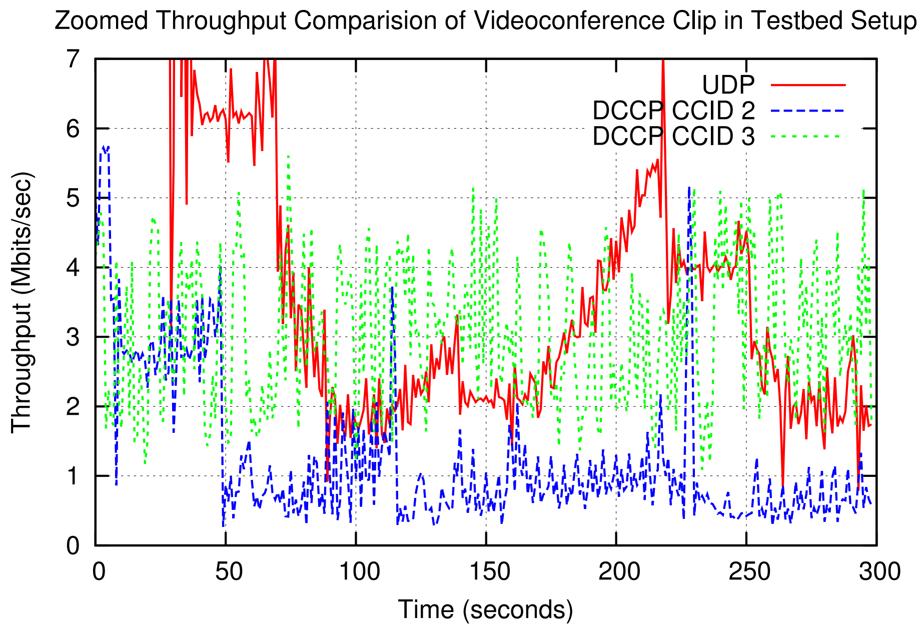


Figure 4.12: More detailed look at the variation in throughput of UDP, CCID 2, and CCID 3 using our videoconference clip in our testbed environment.

control is designed to be less aggressive in adjusting its throughput than either CCID 2 or TCP, this must be the result of linphone adjusting its sending rate to follow the fluctuations in CCID 3's allowed sending rate while CCID 2 follows only the large scale trends in allowed sending rate. We can confirm this by noting that table 4.3 shows CCID 3 sending almost three times as many I-frames as CCID 2. This implies significantly more rate changes since linphone's video codec issues a new I-frame every time the target sending rate is changed.

Figure 4.12 also provides a nice example of linphone's UDP/RTP congestion control. The sending rate starts high and is brought under control after roughly a minute. Then 50 seconds (10 five second RTCP reports) of stable operation pass. At that point, the congestion control begins to increase the sending rate linearly until the loss rate exceeds 10%. A decrease phase then begins reducing the loss rate to under 10%. The two plateaus in throughput at 40-60 seconds and 225-250 seconds are unexpected, however. These likely correspond to points where the video maintains a loss rate just under 10% until a large motion event occurs, at which point the loss rate jumps above 10% and linphone slows down further.

We now turn to examine the fairness of our video streams to other network traffic. Figure 4.13 shows the average fairness of video streaming over UDP, CCID 2, and CCID 3 to background TCP traffic for all our video conference testbed experiments. The

Table 4.3: Frame Statistics for our Videoconference Clip in our Testbed Environment

	Total Frames	I-frames	% I-frames	Undecodable	% Undecodable
UDP	7476	110	1.47%	869	11.62%
CCID 2	7479	252	3.37%	244	3.26%
CCID 3	7476	650	8.69%	155	2.07%

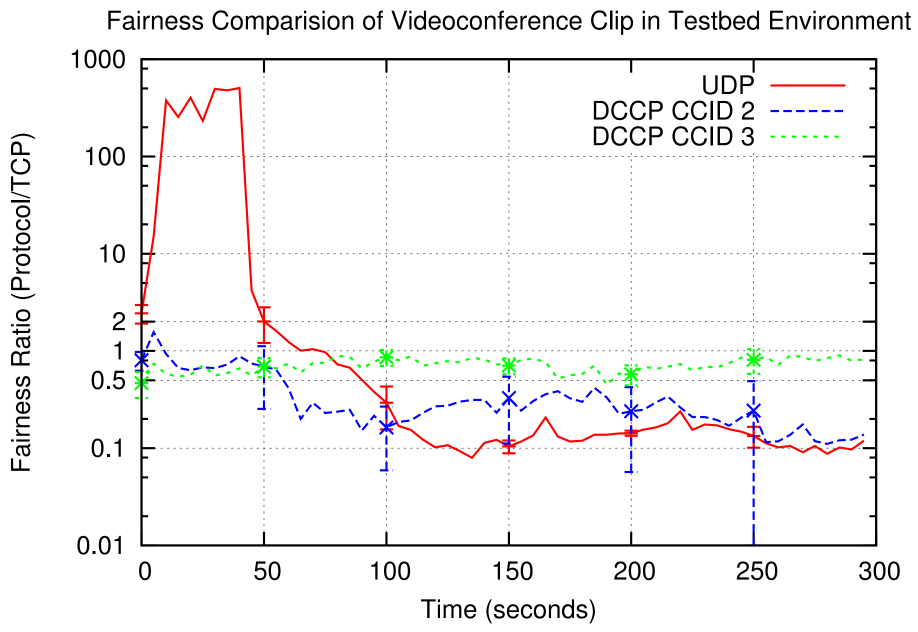


Figure 4.13: Average fairness of UDP, CCID 2, and CCID 3 to TCP over our 5 minute videoconference clip in the testbed environment. Error bars indicate the 95% confidence interval.

error bars at 50 second intervals indicate the 95% confidence interval at those locations. We again use the fairness ratio, $protocol_throughput/tcp_throughput$, as our fairness metric. Because of the range of fairness values over these connections, we plot the fairness ratio on a log scale.

The most immediately noticeable feature of this figure is the massive unfairness that linphone's UDP/RTP congestion control exhibits at the beginning of the connection. This corresponds to the large spike in throughput we observed in our throughput graphs. Clearly, UDP/RTP congestion control allows the application to completely overwhelm any competing traffic. Worse, UDP/RTP takes nearly 50 seconds to get this behavior under control.

Another interesting feature of figure 4.13 is the similarity in fairness between UDP and CCID 2 after the initial 80 seconds. Both protocols achieve throughput that is a factor of eight or so less than the competing TCP flows. This likely stems from the mechanism these protocols use to increase their bitrate at the application level. Both slowly increase their video bitrate at the same rate after 50 seconds of good network conditions. These video flows are competing against TCP flows with essentially unlimited data to send so TCP will utilize any free bandwidth within a few dozen round trip times, much faster than linphone will increase the video bitrate. Hence the unfairness that we observe.

Recall that acceptable fairness between two network flows is generally understood to be achieving throughput within a factor of two of each other [WH06, FHPW08]. From figure 4.13 it is abundantly clear that CCID 2 and UDP, after the initial 80 seconds, are significantly less aggressive than needed to achieve reasonable fairness with TCP. Although, in CCID 2's case, being more aggressive would probably not help quality since CCID 2 already rejects 9.89% of the packets linphone attempts to send.

CCID 3, meanwhile, maintains much more even competition with other network flows. In fact, over all of our tests, CCID 3 maintained an average fairness ratio of 0.59, just within our definition of acceptable fairness. The smoother responses of CCID 3 and its sending rate socket option once again show benefit.

Stepping back to consider all of our testbed experiments, DCCP CCID 3 seems to be the clear winner. It not only achieves better video quality and higher throughput, but also results in better fairness to competing traffic. CCID 2 achieves better video quality than UDP, but suffers from an API limitation in the Linux implementation that forces the application to guess when it can speed up and from a sudden, harsh response to packet loss that occurs more quickly than the application can adjust.

4.2 Short Distance Internet Experiments

Our short distance Internet experiments occurred in a realistic Internet environment with a round trip time of about 56ms, fairly typical for a connection traversing a few hundred miles. In this environment, our video streams competed with a variety of types of background traffic with varying round trip times and constantly changing bandwidth demands.

4.2.1 Movie Clip

Recall that our short distance Internet environment consisted of two hosts in our lab with IPv6 tunnels to endpoints located in Virginia. Traffic traveled from our lab in Ohio to Virginia over one tunnel, and then into the other tunnel and back to our lab. Each test consisted of a linphone videoconference between these two machines along with Iperf TCP connections in both directions to measure fairness. The tests ran for the full 12 minutes of our movie clip.

Figures 4.14 and 4.15 compare the received video quality, measured using PSNR and SSIM respectively, for tests using UDP, CCID 2, and CCID 3. These figures show average cumulative distribution functions of the video quality from ten test runs for each configuration. The error bars shown at 0.1 probability intervals indicate the 95% confidence interval. Observe that UDP achieves moderately better quality on average but has a larger low quality tail than either DCCP CCID. This is consistent between PSNR and SSIM, although UDP's higher quality persists for a few more percentage points when measuring with SSIM instead of PSNR.

The much smaller tail of the DCCP cumulative distribution functions, and tighter confidence intervals, mean that DCCP achieves much more even video quality and fewer visual artifacts than UDP. This is actually a significant benefit since evenness of received

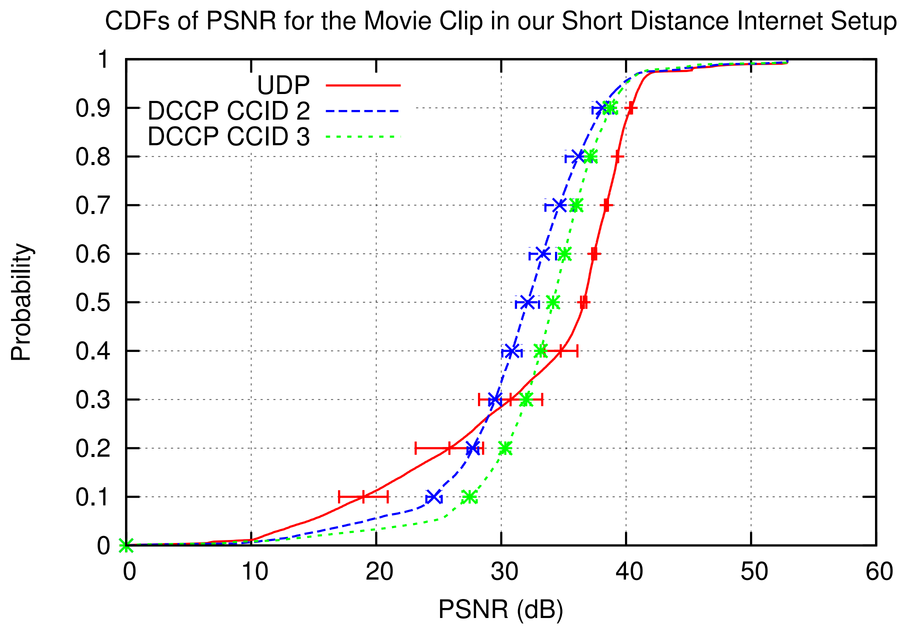


Figure 4.14: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by PSNR, for our movie clip in the short distance Internet environment. Error bars indicate the 95% confidence interval.

video is an important quality. At least one author has found that the evenness of video quality is more important than absolute quality [PG06].

This smaller tail is due to the much more responsive congestion control of DCCP reducing the network loss rate. As table 4.4 shows, UDP has a much higher loss rate than either DCCP CCID, which results in a much greater number of visual artifacts and a large, low quality tail. As we shall see shortly, while CCID 2 has a large rejection percentage, much of that loss occurs in very large bursts that render several frames undecodable instead of introducing visual artifacts.

The lower network loss rate of DCCP is beneficial not only to the video stream itself but also to other network traffic on the link. This is not only because less data is dropped in total but also because most network flows utilize some form of congestion control that

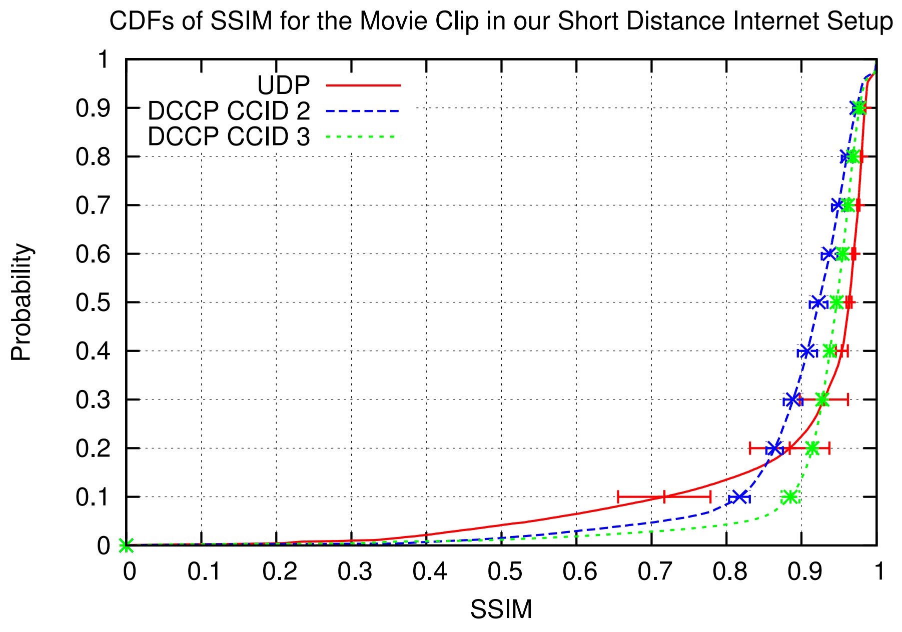


Figure 4.15: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by SSIM, for our movie clip in the short distance Internet environment. Error bars indicate the 95% confidence interval.

reduces their throughput in the presence of loss. As a result, reducing the loss rate results in increased throughput for most protocols, especially TCP.

That said, UDP still achieves higher quality than either DCCP CCID the majority of the time. This turns out to be because it achieves significantly larger network throughput,

Table 4.4: Packet Statistics for our Movie Clip in the Short Distance Internet Environment

	Sent	Received	% Received	Lost	% Lost	Rejected	% Rejected
UDP	1383187	1357927	98.17%	18724	1.35%	0	0.00%
CCID 2	333748	307487	92.13%	89	0.03%	26171	7.84%
CCID 3	442706	439909	99.37%	800	0.18%	1995	0.45%

16.52Mbits/sec on average, instead of CCID 2's 3.55Mbits/sec or CCID 3's 5.13Mbits/sec. The information on network conditions in table 4.4 sheds some light on this situation.

From table 4.4 we see that CCID 2 rejects, because of send queue overflow, almost 8% of the packets that the application attempts to send. Hence, it is not surprising that CCID 2 does not achieve better throughput. That rate of send queue overflow would seem to imply issues between CCID 2 and the application's bitrate control. In fact, from Mathis's TCP throughput model in [MSMO97] we can determine that a TCP connection in this situation with a 0.03% loss rate, like CCID 2, should be able to achieve throughput of about 10Mbits/sec. As CCID 2's congestion control algorithm is nearly identical to TCP's, CCID 2 should be able to attain nearly identical throughput if it were being driven perfectly.

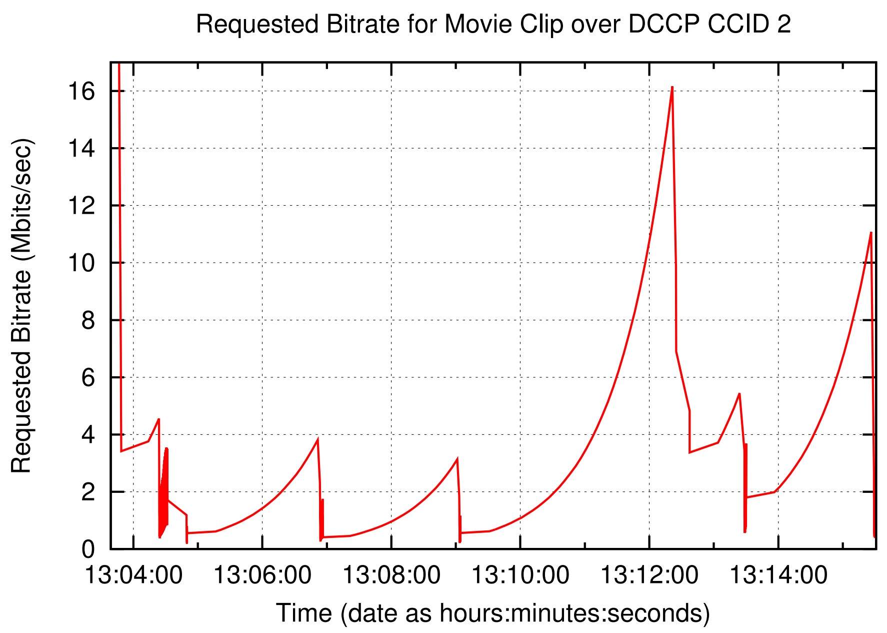


Figure 4.16: Bitrate requested from the video encoder by linphone as a result of feedback from CCID 2 in a representative experiment. This may or may not correspond to the bitrate actually achieved by the video encoder or actual network throughput.

This issue here is that CCID 2 is very harsh in its bitrate decreases and linphone is very gradual in its increases. This is clearly illustrated in figure 4.16. This figure shows the bitrate that linphone requests from the video codec based on feedback from CCID 2 for a representative experiment. Note that the video codec may not follow this bitrate perfectly because of limitations in the compressibility of the video. In this figure there are a number of sharp drops in requested bitrate, corresponding to network losses. Notice that these drops in bitrate are usually larger than the factor of two decrease that CCID 2 performs on a loss. Because the send queue has to overflow before linphone learns that it needs to slow down, it must slow down more than theoretically necessary in order to drain the queue to a usable level again. As the sending rate increases, this additional reduction also increases, as illustrated by the large drop around 13:12:00. This is, of course, compounded by the fact that a rate change requires sending a new I-frame, which is much larger than the average frame.

Notice also that the increase in requested bitrate when using CCID 2 has a significant idle period before beginning and occurs slowly relative to the round trip time of the connection. This contributes to the low throughput achieved. Ideally, linphone should request exactly the bitrate that CCID 2 can send; however, since linphone has no way to discover this rate, it employs a slow increase mechanism as seen in this figure.

Interestingly, while the increase occurs slowly over time, the rate of increase is exponential, which is almost certainly too fast; a linear increase would probably be more appropriate. Recall that we reused linphone's existing bandwidth increase algorithm with CCID 2 since CCID 2 offers no indication of when the application could send faster. This algorithm uses an exponential increase of 20% every 10 seconds[21]. By contrast, CCID 2 increases its sending rate in a linear manner at 1 packet every round trip. This mismatch

[21] Strictly speaking, 20% every two RTCP report intervals. See section 2.2.2 for the full algorithm.

causes linphone to easily overshoot CCID 2's available bandwidth, triggering a packet rejection.

Thus this algorithm is both too conservative in the timing of the increases and too aggressive in the rate of increase once it gets going. This results in the poor bandwidth utilization that we have observed.

Returning to the difference in throughput between UDP and DCCP and table 4.4, we can determine that CCID 3 achieves lower throughput than UDP for a different reason that CCID 2, since it experiences sub-0.5% loss and rejection rates. For CCID 3, the limiting factor turns out to be its congestion control algorithm, TFRC. Recall that TFRC uses an equation to determine its allowed sending rate in such a way that it is fair to TCP.

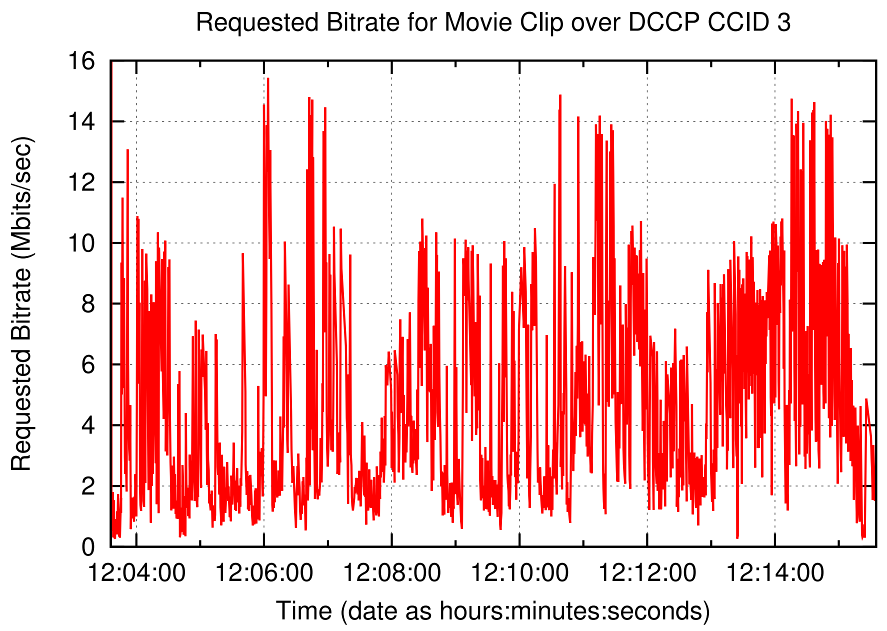


Figure 4.17: Bitrate requested from the video encoder by linphone as a result of feedback from CCID 3 in a representative experiment. This may or may not correspond to the bitrate actually achieved by the video encoder or actual network throughput.

Since CCID 3 gives applications direct access to its computed sending rate via a socket option, the bitrate that linphone requests is directly based off of the bandwidth that CCID 3 would allow. Figure 4.17 shows this requested bitrate for CCID 3 in a representative experiment. The most noticeable feature of this graph is the dramatic variation in requested bitrate. CCID 3 seems to be all over the map here. A large part of the issue is the bursty nature of the data transfer. The allowed sending rate is limited to at most twice the rate that CCID 3 received data in the last round trip, which is, in turn, based off the size of the previous few frames. Since frame sizes vary frame to frame, so does the received rate for the previous round trip and, hence, the current allowed sending rate.

Another important factor in CCID 3's allowed sending rate is the round trip time of the connection. Figure 4.18 shows the experienced round trip for the same sample connection as figure 4.17. Notice the significant amount of variation, in part because of the bursty nature of the video stream resulting from size variations between P-frames and

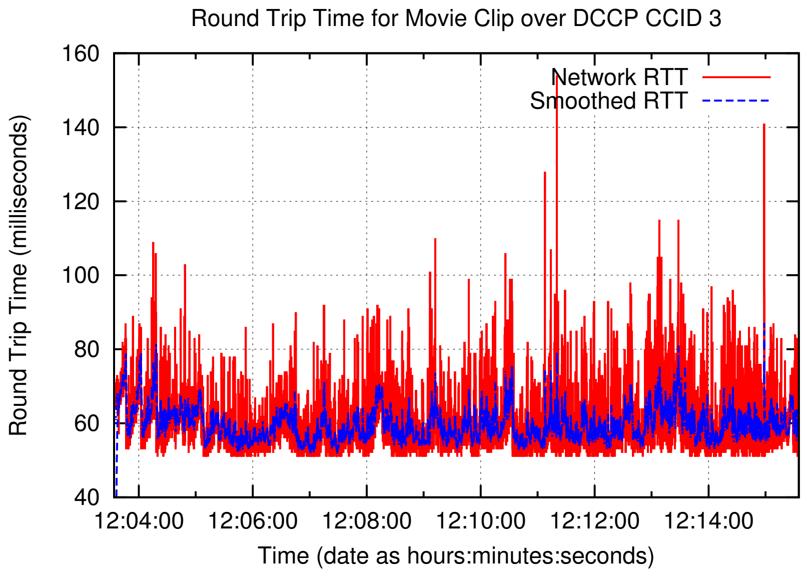


Figure 4.18: Round trip times experienced by a sample CCID 3 connection while streaming our movie clip through the short Internet environment.

the relative size difference between I and P-frames. In order to combat this, CCID 3 uses a low pass filter to smooth the round trip time. This filter is as follows [FHPW08]:

$$RTT = q * RTT + (1 - q) * RTT\_Sample$$

where $q = 0.9$ is recommended. This is the same round trip time smoothing algorithm used in TCP with great success. We have also plotted this smoothed round trip time in 4.18 and still observe noticeable variation. Note that CCID 3 only sends feedback from receiver to sender once per round trip time and these return packets are required for an round trip time sample. This means that CCID 3 obtains only a small fraction of the round trip time samples that TCP could obtain. Fewer samples imply a less accurate round trip time value, particularly when there is high variation.

Because of these issues, neither DCCP CCID achieves the throughput that UDP is able to reach. As a result, UDP is able to achieve higher video quality, although, as noted earlier, it has a much larger tail of poor quality video.

4.2.2 Videoconference Clip

The experiments with our videoconference clip follow exactly the same procedure. Our five minute video conference clip is streamed between two machines in our lab with IPv6 tunnels such that traffic traveled from our lab to Virginia and back to our lab.[22] We ran Iperf flows in both directions along with our video streams in order to measure fairness.

Figures 4.19 and 4.20 show cumulative distribution functions of video quality, as measured by PSNR and SSIM respectively, for video streams over UDP, CCID 2, and CCID 3 in this environment. Each cumulative distribution function is averaged over ten experimental runs. The error bars shown at 0.1 probability intervals indicate the 95% confidence interval. These figures show that UDP achieves better video quality than either

[22] An example of the video received over CCID 3 can be viewed at http://youtu.be/2YhofRUzLDQ.

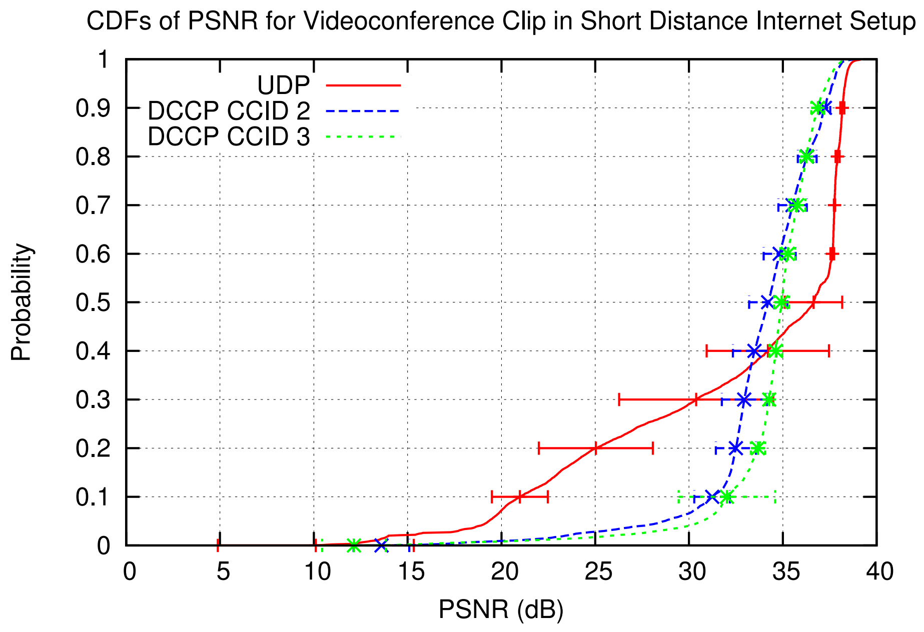


Figure 4.19: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by PSNR, for our videoconference clip in the short distance Internet environment. Error bars indicate the 95% confidence interval.

DCCP CCID about 68% of the time. Below a probability of 0.4 an interesting low quality bulge occurs in UDP's PSNR CDF and, to a much lesser extent, its SSIM CDF.

Nearly identical low quality bulges in UDP's CDFs also occur in our long distance Internet environment while streaming our videoconference clip. For that reason, we defer detailed discussion of the causes of these bulges until section 4.4.2. Suffice it to say that UDP packet losses and receive queue overflows cause the loss of texture data in I-frames which results in visual artifacts that persist for quite some time.

As indicated by this low quality bulge, both DCCP CCIDs achieve much more even video quality than UDP does. Notice also that both CCIDs have smaller standard deviations that UDP does. Although neither CCID is able to achieve the same peak video

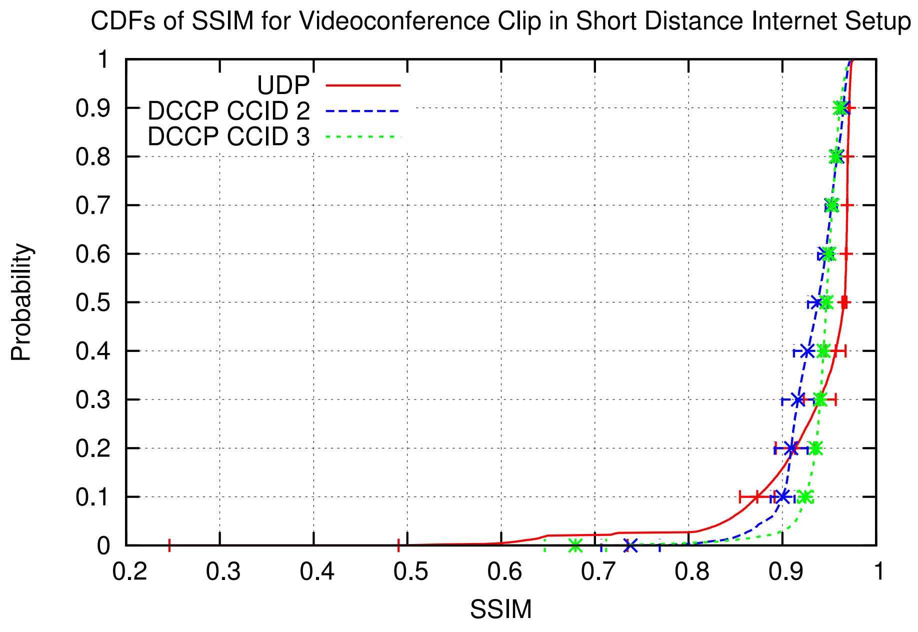


Figure 4.20: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by SSIM, for our videoconference clip in our short Internet environment. Error bars indicate the 95% confidence interval.

quality that UDP does, both CCIDs achieve very even video quality over 95% of the video stream.

UDP achieves better peak video quality because it achieves a much higher sending rate, just like we observed with our movie clip. While CCID 3 achieves 4.56Mbits/sec on average and CCID 2 achieves 4.03Mbits/sec, UDP achieves 22.39Mbits/sec on average. Interestingly, in about half of our tests UDP never attempted to adjust the video encoder's bitrate, leaving the requested bitrate at 100Mbits/sec and essentially letting the video encoder achieve the maximum bitrate that the video complexity warranted. Even in those tests where UDP did adjust the requested bitrate, it rarely made more than ten adjustments during the whole test.

The lack of adjustments in linphone's requested bitrate when using UDP/RTP congestion control simply indicates that the loss rate never exceeded 10% and the the round trip time never doubled between two RTCP reports. This is roughly equivalent to saying that network conditions never became terrible. It does not indicate good fairness between the video stream TCP as figure 4.21 shows. Recall that a fairness ratio of 1 is perfect fairness and that a factor of two difference in throughput is considered to be acceptable fairness. UDP clearly competes unfairly with TCP the vast majority of the time, despite linphone's UDP/RTP congestion control.

A major part of this unfairness is that TCP is distinctly more sensitive to loss than linphone's UDP/RTP congestion control. TCP will halve its sending rate in response to a single loss while UDP/RTP will not react until the loss rate exceeds 10% over a five second period. This will tend to result in TCP eliminating the congestion before UDP

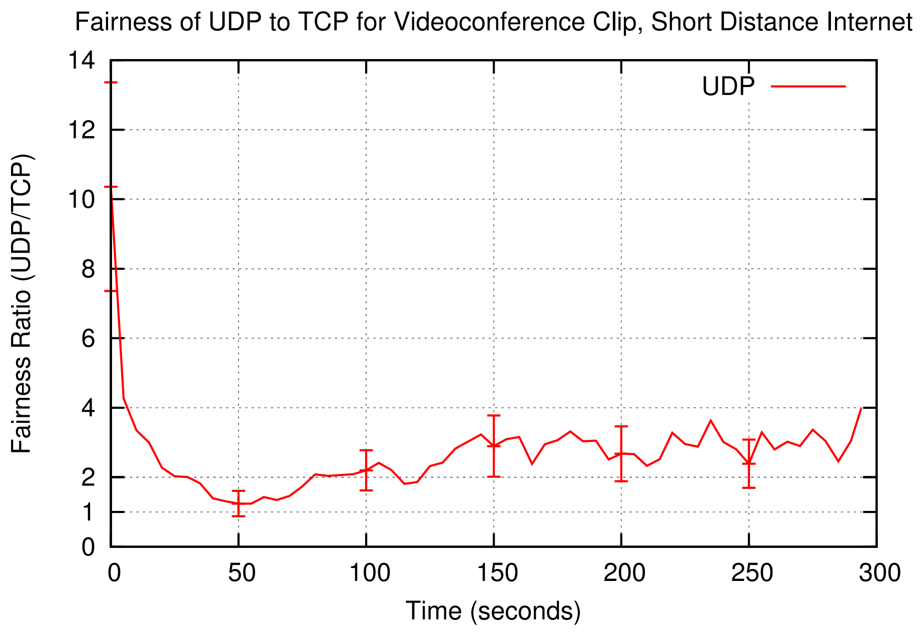


Figure 4.21: Average fairness of UDP to TCP while streaming our videoconference clip in the short distance Internet environment. Error bars indicate the 95% confidence interval.

slow downs, effectively ceding bandwidth to UDP. In addition, linphone's UDP/RTP congestion control has no dependence on the round trip time, unlike TCP and most other congestion controlled protocols. The bursty data stream that linphone sends tends to cause fairly large changes in round trip time, which will effect TCP but not phase linphone's UDP/RTP congestion control in the slightest.

In this environment, there is also a difference in received video quality between CCID 2 and CCID 3 as seen in figure 4.19. As we have noticed before, CCID 3 performs better than CCID 2 at the low probability end of the CDF, despite virtually identical peak performance. As discussed previously, this is caused by CCID 2's harsh response to loss and slow application level increase afterward. CCID 3 meanwhile, experiences rapid sending rate oscillations, but achieves a higher average throughput in this environment,

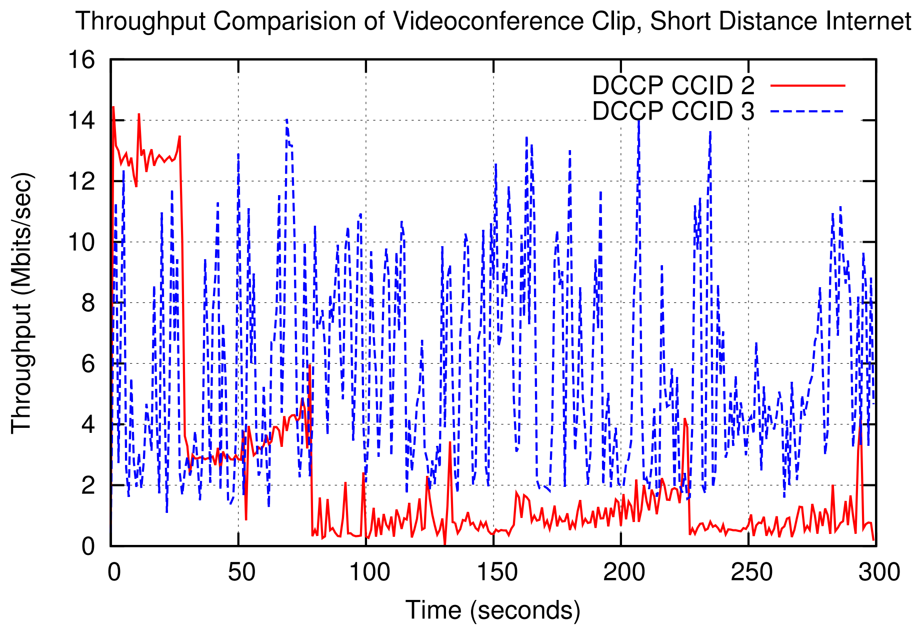


Figure 4.22: Throughout of representative CCID 2 and CCID 3 flows while streaming our videoconference clip in the short distance Internet environment.

where the round trip time is just a few frames. Figure 4.22 shows the network throughput achieved by representative CCID 2 and CCID 3 flows and illustrates these issues nicely.

Both DCCP CCIDs achieve much more even video quality than UDP in our short distance Internet environment although they are unable to match UDP's overall video quality because they cannot match its throughput. CCID 3 performs better than CCID 2 by giving the application a better understanding of its allowed sending rate. UDP's high throughput comes from being distinctly unfair to competing TCP traffic.

4.3 Bitrate Adjustment Interval Analysis

When we began to run our long distance Internet tests, we observed major issues with DCCP. In particular, both CCIDs would reject over 70% of the packets linphone attempted to send while apparently encouraging linphone to send nearly twice as many packets as when using UDP. A further oddity was that a very large quantity of the frames sent were I-frames, nearly 50% in some cases.

Figure 4.23 shows linphone's requested bitrate over a 30 second segment of one of these DCCP CCID 2 connections. Notice the rapid oscillation between high and low bitrate. As this is CCID 2, each of the bitrate decreases represents a send queue overflow and associated packet loss, which explains the 70%+ loss rate. Explaining the bitrate increases requires understanding in detail how linphone adjusts its requested bitrate.

Linphone keeps a variable containing the bitrate that it requests from the video encoder and on loss decreases this value by the maximum of 10% or the loss rate (if using UDP). However, as mentioned before, the encoder views this requested bitrate as a recommendation and may not follow it if the video stream's complexity is unsuitable to compression at the requested bitrate.

When modifying linphone to support DCCP, we found it necessary to include a sanity check on this requested bitrate. If this requested bitrate is not within a factor of 10

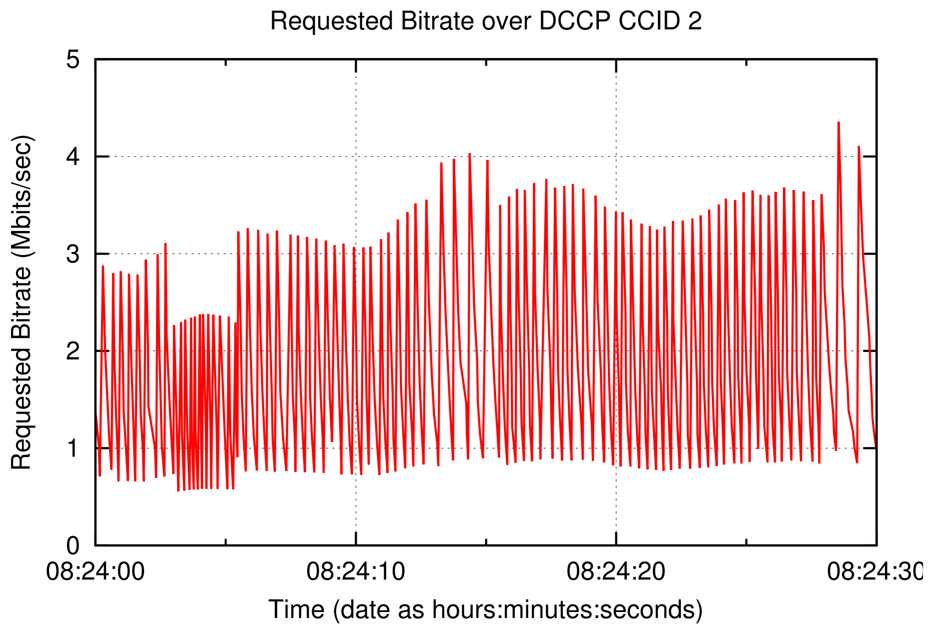


Figure 4.23: Requested bitrate oscillations in a DCCP CCID 2 connection with no bitrate update interval and high round trip time variation in the long distance Internet environment.

of the video encoder's recent actual bitrate, reset the requested bitrate to be within a factor of two of this actual bitrate (either half or twice, as appropriate). Without this sanity check, we found that linphone using CCID 2 had a tendency to decrease its requested bitrate to the minimum allowed bitrate, 128kbits/sec, irrespective of what bitrate the video encoder was actually putting out.

This sanity check causes the behavior observed in figure 4.23. Linphone starts at some bitrate and suffers a send queue overflow, so the requested bitrate is decreased. This triggers an I-frame, which is larger than the average frame. Since the send queue just overflowed from a normal sized frame, this I-frame is almost certain to cause another queue overflow, which triggers a further bitrate reduction. Eventually, the requested bitrate is a factor of 10 below the video encoder's actual achieved bitrate and our algorithm resets it to half the achieved bitrate. This, of course, results in another I-frame, send queue overflow, and bitrate reduction. Unintended consequences like this are part of the reason that designing and implementing congestion control is so hard.

The trigger for this behavior is large variation in round trip time. Figure 4.24 shows the round trip time for this same CCID 2 connection along with the smoothed round trip time used by both CCIDs. Observe the high variation, even in the smoothed round trip time. As the round trip time varies, the effective sending rate of CCID 2, one window of data per round trip time, also varies. This can cause queue overflow without an actual packet loss and appears to be the initial trigger for the nasty oscillations just discussed.

The reason this behavior did not show up in our other tests is that the round trip time in previous environments was much shorter. In fact, it was less than or roughly equal to the time between frames in both of the other environments. As a result, CCID 2 had more flexibility since the whole window contained only a single frame instead of being shared

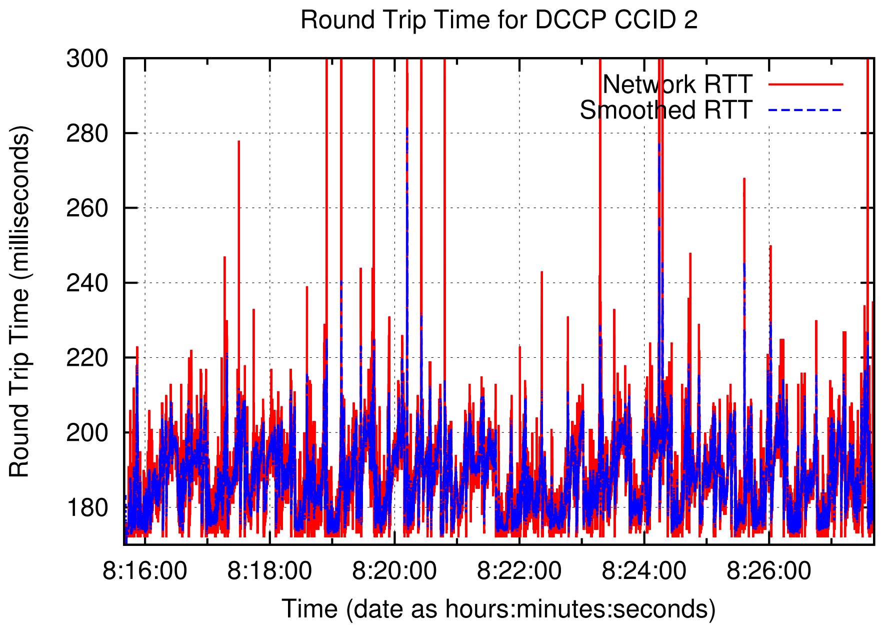


Figure 4.24: Round trip time for a DCCP CCID 2 connection in our long distance Internet environment that exhibits very poor streaming performance.

between multiple frames. Further, the allowed sending rate would increase more rapidly and CCID 2 would be able to adjust to variations in application sending rate much faster. In our long distance Internet environment, the round trip time is about 8-10 frames in length so CCID 2 is constrained to react much more slowly. This slower reaction increases the likelihood that packets will be rejected.

CCID 3 also experiences similar behavior with only minor differences in cause. The round trip time variations cause many requested bitrate updates. Because of the longer round trip, CCID 3 is similarly less flexible in how rapidly its sending rate can vary. In particularly, CCID 3 is constrained to sending at most twice the receiving rate over the last round trip time. As a result, the emission of I-frames will tend to trigger send queue overflow and a similar cycle emerges.

Our solution to this problem was to limit the frequency with which linphone would attempt to update its bitrate based on feedback from DCCP. Since the much larger size of I-frames, and the send queue overflows that result, are a large part of the problem, simply ignoring these overflows will break this cycle.

An alternative solution that would be to simply not update the requested bitrate if doing so would take it outside of a factor of 10 of the recent actual bitrate. This, however, has the problem that the requested bitrate could slip outside this factor of 10 not by explicit update but by a change in video complexity. Even ignoring this issue, the requested bitrate would not be as accurate as in our current scheme, which would reduce the effectiveness of increases and decreases to this requested bitrate. For these reasons, we prefer to limit the frequency with which linphone will update this requested bitrate.

The ideal update frequency was not immediately obvious to us so we performed an examination of various update intervals. The sections below discuss our findings.

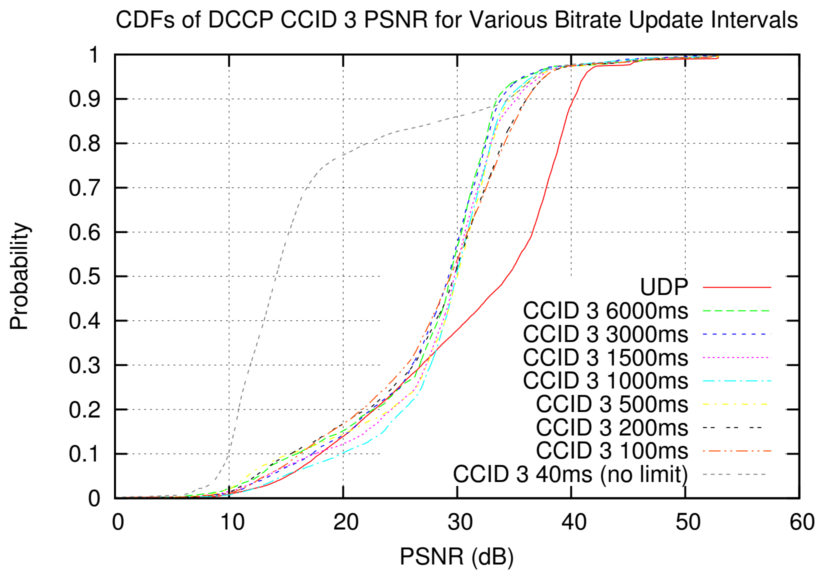
4.3.1 Long Distance Internet Experiments

Figures 4.25 and 4.26 show cumulative distribution functions of the video quality achieved by CCID 3 and CCID 2, respectively, with various bitrate update intervals. All of these CDFs are averaged over three test runs with our 12 minute movie clip. The video quality metric being used is PSNR. Both of these figures also include UDP's video quality CDF for comparison. Note that 40ms is the amount of time between video frames, so the no limitation on the bitrate update interval is equivalent to 40ms interval. These unlimited tests are the ones that achieved a greater than 70% packet reject rate and severe requested bitrate oscillations as mentioned earlier.

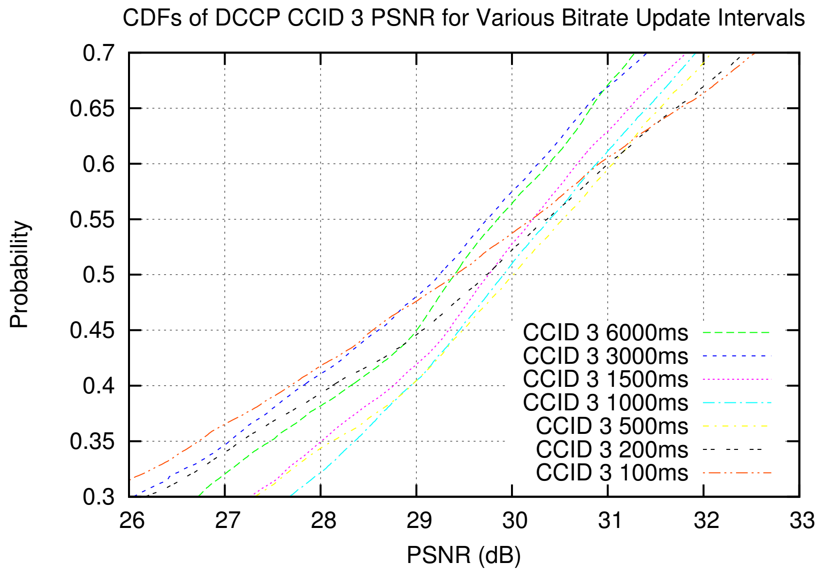
Considering CCID 3 and figure 4.25 first, observe that our unlimited 40ms curve achieves extremely poor quality, with 80% of frames being unwatchably poor. This should come as no surprise given that linphone suffers greater than 70% packet rejection in this case. Further, notice that UDP achieves higher video quality than any CCID 3 test about 70% of the time. This occurs because UDP achieves much greater throughput than either DCCP CCID. We will discuss why this occurs in more detail in section 4.4.1.

Among the CCID 3 curves, there is relatively little variation in quality once an initial bitrate update interval of roughly the same length as the round trip time has been imposed. The difference between any two bitrate update intervals between 100ms and 6 seconds is 1dB PSNR at 50% and no more than 5dB PSNR anywhere. The shorter update intervals achieve slightly better maximum quality while an interval of around one second exhibits the tightest lower knee, representing a minimum of extremely poor quality frames.

Figure 4.25b shows a close up of the CCID 3 CDFs around the 0.5 probability mark in order to illustrate the differences there more clearly. At this point, the short update intervals of 100ms and 200ms transition from being the best option in the upper half to the worst choice in the lower half. It is interesting to note that the round trip time in this environment is right about 180ms. The 500ms and 1000ms update intervals move from



(a) Full CDFs



(b) Closeup of CDFs around 0.5 probability

Figure 4.25: Average cumulative distribution functions of video quality, as measured by PSNR, for CCID 3 using various bitrate update intervals and our movie clip in the long distance Internet environment.

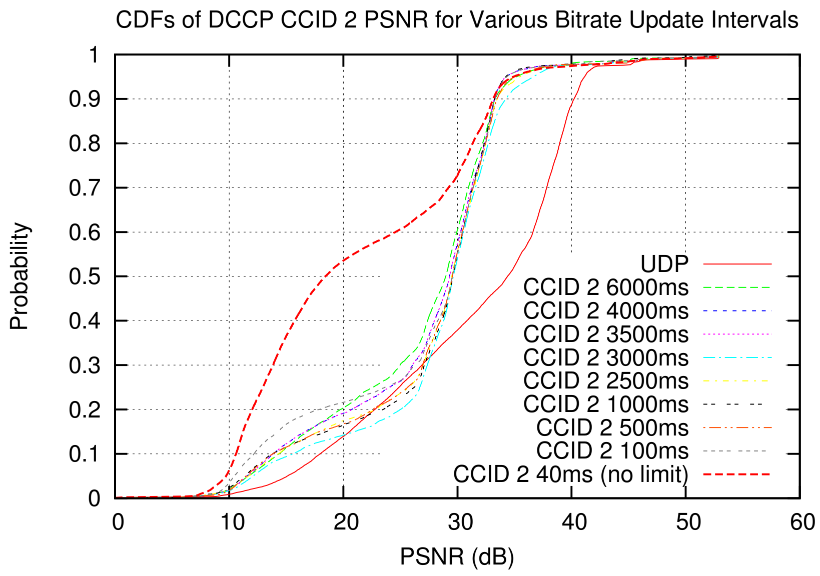
middle of the road at the high end to best option at the low end. The long 4000ms and 6000ms bitrate update intervals do uniformly poorly.

We consider the 1000ms update interval to be optimal in this situation because it minimizes the number of extremely poor quality frames while also achieving mid-pack performance at the high end. A lower update interval would achieve better high end performance, but UDP already does much better in that space than any DCCP configuration so we felt it more important to focus on the lower end where CCID 3 can compete with UDP.

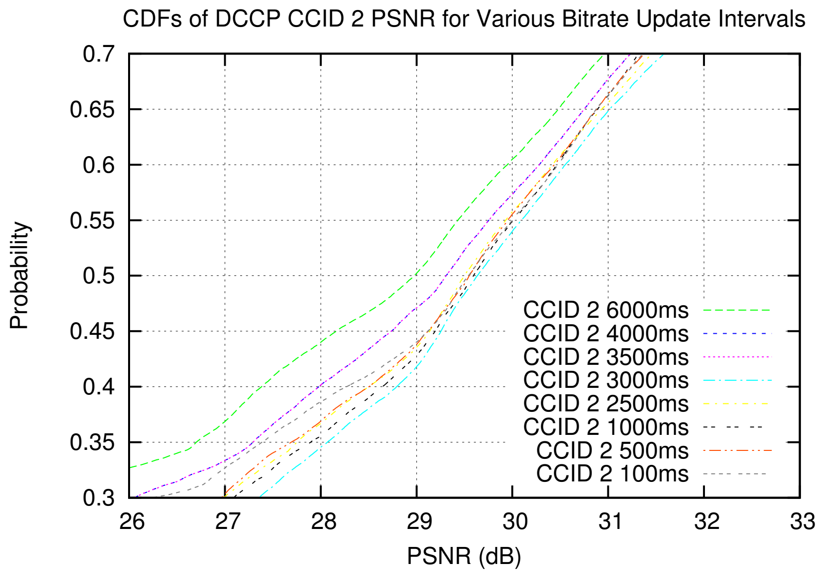
Turning now to CCID 2, consider figure 4.26. Notice that, compared to CCID 3, there are larger differences in video quality as the bitrate update interval changes. UDP still achieves noticeably higher quality, because it allows much greater throughput, and no limitation on the bitrate update interval still produces incredibly poor quality. Between these two extremes lie our CCID 2 CDFs, with various bitrate update intervals between 100ms and 6 seconds. The best bitrate update interval improves video quality by up to 10dB PSNR.

Not only does the bitrate update interval make more difference with CCID 2 than CCID 3, it also makes a more consistent difference. The 3000ms update interval leads the pack with the others spreading out behind it based on their distance from this optimal value. This ordering is fairly consistent over the whole probability range; the CDFs do not cross each other. This makes it easy to pick 3000ms as our optimal bitrate update interval for CCID 2 in our long distance Internet environment.

While we have been able to identify optimal bitrate update intervals in this particular environment, our experiments have not revealed a specific model or heuristic for choosing a bitrate update interval in an arbitrary environment. It seems likely that the optimal bitrate is at least related to the round trip time and the round trip time variance, but we



(a) Full CDFs



(b) Closeup of CDFs around 0.5 probability

Figure 4.26: Average cumulative distribution functions of video quality, as measured by PSNR, for CCID 2 using various bitrate update intervals and our movie clip in the long distance Internet environment.

have not found a specific model for how they relate. It is also interesting to note how different the optimal values are between DCCP CCIDs.

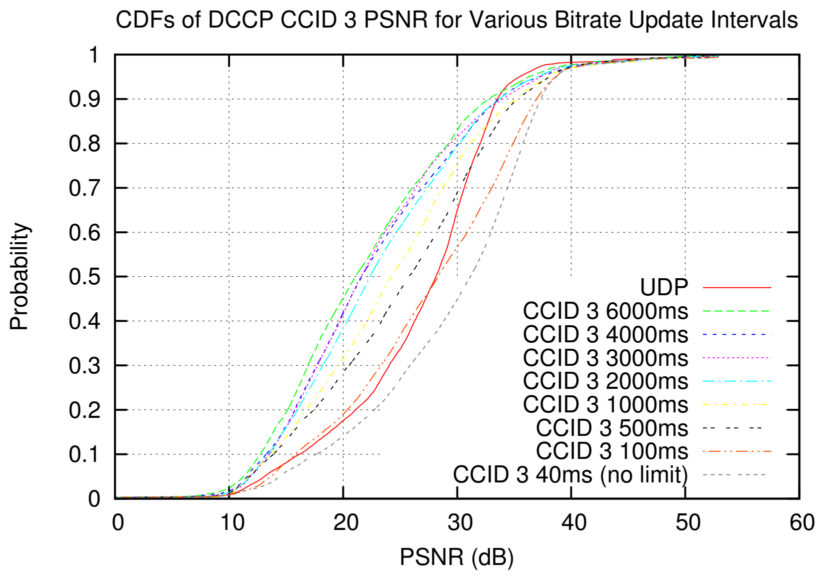
4.3.2 Testbed Experiments

To gain a deeper understanding of the bitrate update interval and examine how much difference this parameter makes between environments, we also examined the bitrate update interval in our testbed environment.

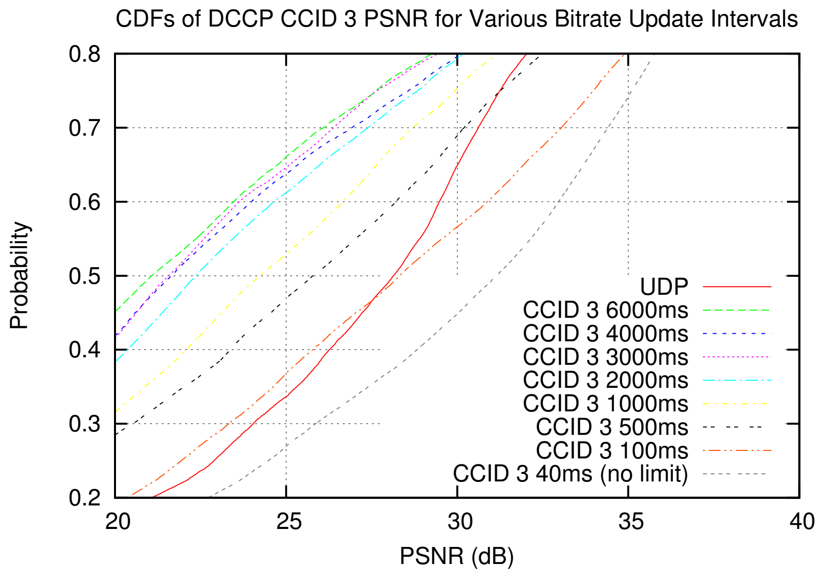
Figures 4.27 and 4.28 show cumulative distribution functions of received video quality over DCCP CCID 3 and CCID 2, respectively, for different bitrate update intervals. The video quality metric utilized is PSNR, and the UDP video quality CDF is also included for comparison. Each of these CDFs represents a average over three tests with our 12 minute movie clip. Note again that 40ms is the time between frames so no limitation on the bitrate update interval produces a 40ms update interval.

In this environment, DCCP is quite competitive with UDP. Considering CCID 3 and figure 4.27 first, notice that bitrate update intervals of 40ms and 100ms achieve higher or virtually identical video quality to UDP. Recall that the round trip time in this environment with a loaded network is around 50ms. This suggests an optimal update interval of a small multiple of the round trip time, which is inconsistent with our experience in the long distance Internet environment. This difference is likely due to increased flexibility in sending rate at lower round trip times. Recall that CCID 3's sending rate is limited to twice the receiving rate in the last round trip time.

Above a bitrate update interval of 100ms, CCID 3's performance falls behind UDP. CCID 3 suffers from very poor video quality with the 2000, 3000, 4000, and 6000 ms update intervals while the 500ms and 1000ms intervals achieve only mediocre quality compared to UDP. This is because CCID 3's allowed sending rate can change greatly in



(a) Full CDFs



(b) Closeup of CDFs around 0.5 probability

Figure 4.27: Average cumulative distribution functions of video quality, as measured by PSNR, for CCID 3 using various bitrate update intervals and our movie clip in the testbed environment.

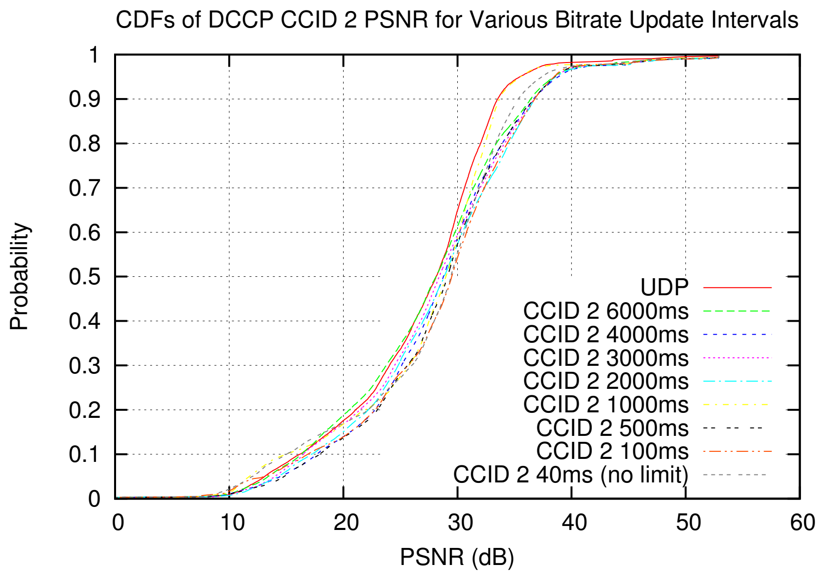
the 10-600 round trips that these intervals represent, but the video encoder will not be updated to match until the end of the interval.

It seems pretty clear that a 40ms update interval is optimal for CCID 3 in our testbed environment. Using 100ms is reasonable, although a few dB PSNR worse, but above that the video quality falls off badly. At 500ms, the average video quality is 11dB PSNR less than at 40ms, and by 6000ms it is 20dB less.

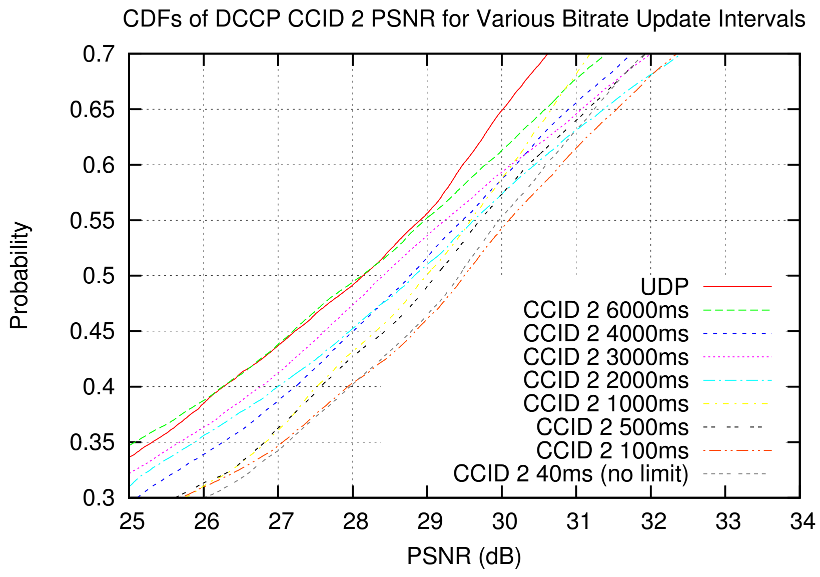
Turning to CCID 2, figure 4.28 shows a very tight grouping of all cumulative distribution functions; the total spread is only about 1.5dB PSNR. From the close up, we can observe that CCID 2 performs better than UDP at all update intervals except 6000ms. Once again, the 40ms and 100ms update intervals achieve the best video quality. Below a probability of 0.5, the 500ms and 1000ms intervals are next in line, but above this point, they trade places with the 2000ms and 3000ms intervals.

With the short round trip time in our testbed environment, CCID 2 increases its window much faster than in our long distance Internet environment, leading to quicker recovery after a loss. Similarly, the window can be increased more quickly to accommodate a sudden increase in frame size. This allows CCID 2 to handle more variation in bitrate without needing the settle down period that the bitrate update interval provides. Further, all of the tested intervals are shorter than the 10 second interval (after a 50 second idle period) between application bitrate increases so they have no impact on application level bitrate increases. This explains why the bitrate update interval makes virtually no difference for CCID 2 in this environment.

Overall, while the bitrate update interval is crucial for long round trip time environments, it appears to be less necessary in environments with shorter round trips. Utilizing a long bitrate update interval anyway can result in distinctly poorer CCID 3 video quality, but has little impact on CCID 2.



(a) Full CDFs



(b) Closeup of CDFs around 0.5 probability

Figure 4.28: Average cumulative distribution functions of video quality, as measured by PSNR, for CCID 2 using various bitrate update intervals and our movie clip in the testbed environment.

4.4 Long Distance Internet Experiments

We now turn to examine the performance of video streams over DCCP and UDP in our long distance Internet environment. This environment has a round trip time of about 178ms, which is typical for transcontinental connections. In addition, video streams in this environment face competition from a variety of other traffic flows of different types and characteristics.

4.4.1 Movie Clip

Recall that our long distance Internet environment consisted of two hosts in our lab with IPv6 tunnels. One tunnel's endpoint was in Virginia and the other's was in California. Traffic traveled from our lab in Ohio to Virginia over one tunnel, across the IPv6 Internet to California, and then into the other tunnel and back to our lab. Each test consisted of a linphone videoconference between these two machines along with Iperf TCP connections in both directions to measure fairness. Each test ran for 12 minutes, which is the length of our movie clip. All tests used a bitrate adjustment interval of 1 second for CCID 3 and 3 seconds for CCID 2. We identified these as the optimal bitrate adjustment intervals in the previous section.

Figures 4.29 and 4.30 show cumulative distribution functions of the received video quality, measured using PSNR and SSIM respectively, for video streams over UDP, CCID 2, and CCID 3. Each CDF is the average of ten separate test CDFs, representing two hours of video. These CDFs plot the probability of a connection achieving at most the specified video quality. The error bars shown at 0.1 probability intervals indicate the 95% confidence interval. The most obvious feature of these graphs is that UDP outperforms both DCCP CCIDs nearly all the time, often by 5-8dB PSNR. DCCP only does better using CCID 3 and only for the poorest 15-20% of each connection.

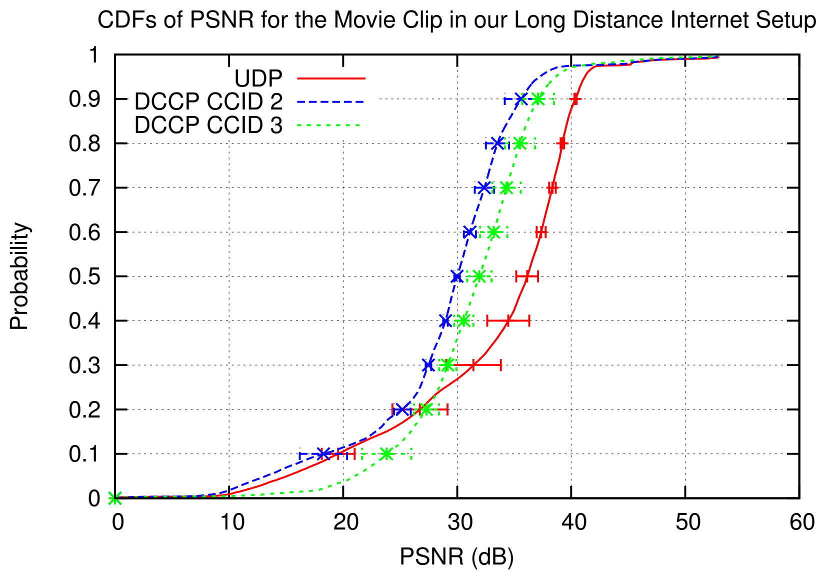


Figure 4.29: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by PSNR, for our movie clip in the long distance Internet environment. Error bars indicate the 95% confidence interval.

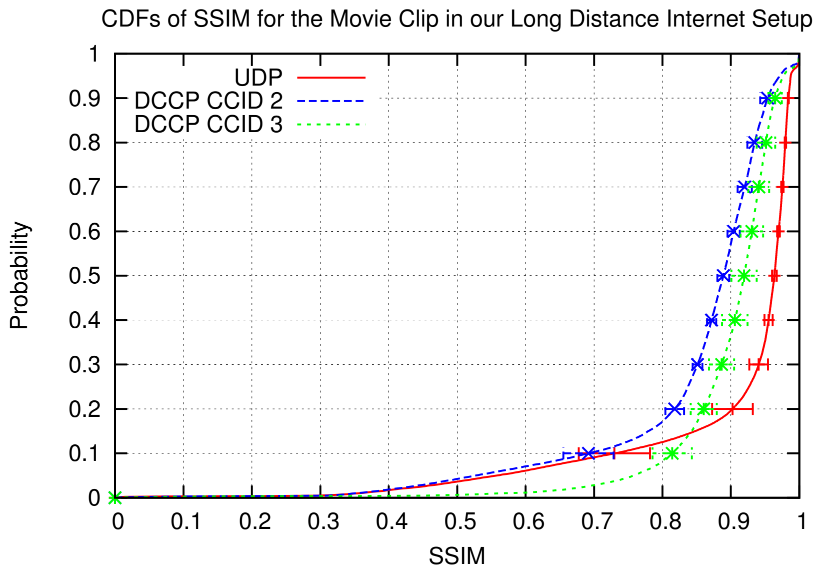
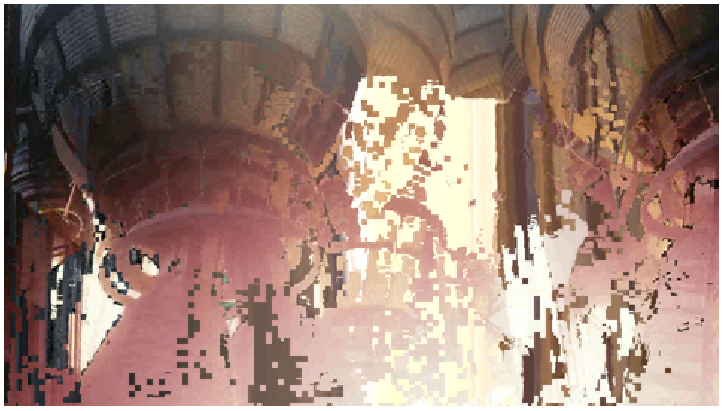
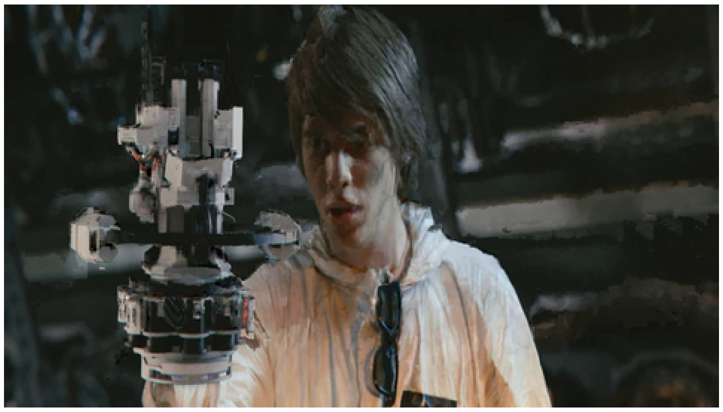


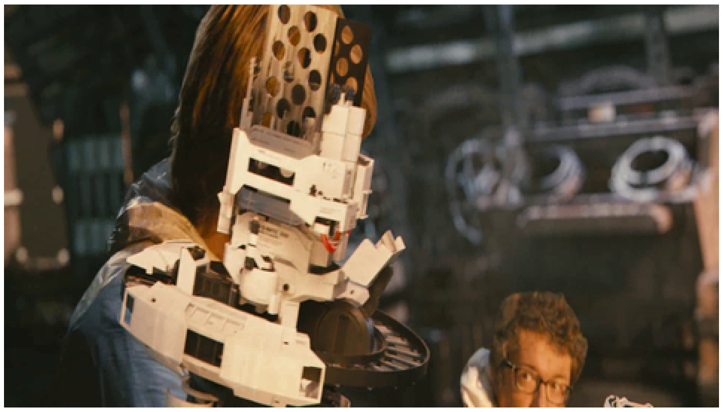
Figure 4.30: Average cumulative distribution functions of UDP, CCID 2, and CCID 3 video quality, as measured by SSIM, for our movie clip in the long distance Internet environment. Error bars indicate the 95% confidence interval.



(a) 15dB PSNR, 0.64 SSIM



(b) 25dB PSNR, 0.88 SSIM



(c) 35dB PSNR, 0.96 SSIM

Figure 4.31: Example video frames with various PSNR and SSIM values. These frames are taken from our CCID 3 movie clip tests in our long distance Internet environment.

To demonstrate how PSNR and SSIM values align with perceived quality, figure 4.31 shows several video frames from these tests with their PSNR and SSIM values.[23] You will notice that the 15 and 25 dB PSNR frames exhibit significant corruption while the 35 dB frame exhibits no noticeable issues.

To understand why DCCP achieves lower video quality than UDP, it is helpful to examine linphone's requested bitrate. This requested bitrate is the bitrate that linphone requests from the video encoder based on the feedback it receives from DCCP or UDP/RTP. It is important to note that the video codec may deviate from this bitrate and may even ignore it completely if it is unachievable given the complexity of the current video material. In other words, this bitrate indicates the throughput that the congestion control algorithm *would* allow the video encoder to achieve.

Figure 4.32 shows average cumulative distribution functions for the requested bitrate when using UDP, CCID 2, or CCID 3 in our experiments. The error bars at 0.1 probability intervals indicate the 95% confidence interval. It is immediately obvious that UDP allows a vastly higher sending bitrate than either DCCP CCID. Both DCCP CCIDs constrain their sending rate to between 1 and 2Mbits/sec while UDP is actually constrained by its maximum allowed sending rate, 100Mbits/sec, 10% of the time. Note that linphone never actually sends that fast, the video complexity never merits that much bandwidth; however, the UDP/RTP congestion control claims that linphone could send that fast.

The reason for the dramatic difference in requested bitrate between DCCP and UDP is that the UDP/RTP congestion control algorithm is based purely on loss rate and extreme changes in round trip time. As long as the round trip time does not double and the loss rate is under 10%, linphone's UDP/RTP congestion control algorithm will continue to increase its sending rate. What we observe in this situation is that sending our video stream at

[23] An example of the video received over CCID 2 can also be viewed at http://youtu.be/aQaiK71_2Ns.

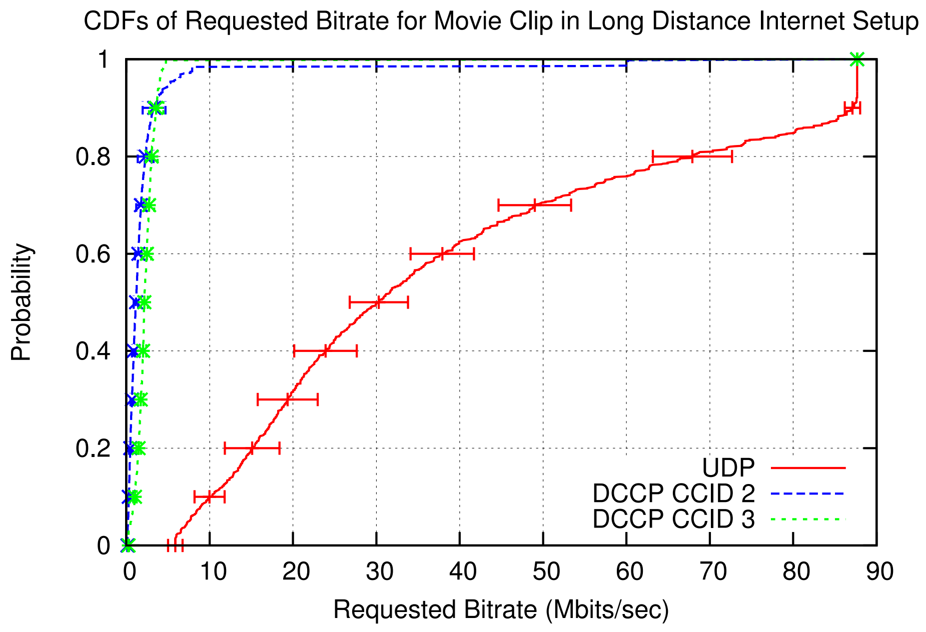


Figure 4.32: Average cumulative distribution functions of the video bitrate requested by linphone when using UDP, CCID 2, or CCID 3 while running tests with our movie clip in the long distance Internet environment. Error bars indicate the 95% confidence interval.

maximum bitrate only rarely generates enough traffic to cause a 10% loss rate; when it does, UDP/RTP slows down. Otherwise, the requested bitrate continues to increase.

Both DCCP CCIDs, on the other hand, are designed to adjust their throughput to be roughly TCP friendly [FK06, FKP06] and TCP's throughput depends on both the loss rate and the round trip time. Mathis proposed what has become a standard model for TCP's throughput in [MSMO97]. This model predicts TCP's throughput from network parameters like loss rate and round trip time, given that TCP maintains steady state operation in congestion avoidance. The model is as follows [MSMO97]:

$$BW = \frac{MSS}{RTT}\frac{C}{\sqrt{p}}$$

where BW is bandwidth, MSS is the maximum segment size, typically the MTU of the link, RTT is round trip time, C is a constant equal to $\sqrt{3/2}$, and p is the loss rate. From this model, it is readily apparent that bandwidth varies inversely with round trip time.

Since CCID 2 uses a congestion control algorithm that is nearly identical to TCP's, the Mathis model should still also be applicable. In fact, Mathis's model actually ignores completely the major point in which the algorithm's differ: fast recovery [MSMO97]. At the very least, CCID 2 should achieve no more than a factor of two more than the model would predict; otherwise, it would no longer be reasonably fair to TCP. Using the information from table 4.5 for CCID 2, we determine the predicted throughput from Mathis's model as follows:

$$BW = \frac{1480}{0.19485}\frac{\sqrt{3/2}}{\sqrt{0.0004}}$$

$$BW = 465132.8 bytes/sec$$

$$BW = 3721062.2 bits/sec$$

$$BW = 3.5 Mbits/sec$$

This is fairly close to the 2.04Mbits/sec that CCID 2 actually achieves. The difference likely occurs because linphone does not have an unlimited, instantly available supply of data for CCID 2. It supplies data in chunks at regular intervals. Nevertheless, this analysis

Table 4.5: Network Parameters for Long Distance Internet Environment

	MTU	RTT	Network Loss Rate	Average Throughput
UDP	1480 bytes	207.35ms	1.51%	16.65Mbits/sec
CCID 2	1480 bytes	194.85ms	0.04%	2.04Mbits/sec
CCID 3	1480 bytes	194.57ms	0.08%	2.57Mbits/sec

does explain why CCID 2 requests such a low bitrate relative to UDP/RTP; it's congestion control algorithm is incapable of higher throughput given the network conditions.

We can perform a similar analysis of the throughput predicted by Mathis's model under the network conditions that CCID 3 experiences. Using the information from table 4.5 for CCID 3, the predicted throughput from Mathis's model is determined as follows:

$$BW = \frac{1480}{0.19457}\frac{\sqrt{3/2}}{\sqrt{0.0008}}$$

$$BW = 329371.8 bytes/sec$$

$$BW = 2634974.8 bits/sec$$

$$BW = 2.51 Mbits/sec$$

This is nearly identical to the 2.57Mbits/sec that CCID 3 actually achieves. Note that CCID 3 uses the TFRC algorithm, which is very different from TCP's congestion control. However, TFRC is designed to achieve similar throughput to TCP given the same network conditions. Hence, this analysis actually speaks well to TFRC's friendliness to TCP. It also reveals that CCID 3 is prevented from achieving higher bandwidth by its congestion control algorithm because higher bandwidth would be unfriendly to competing TCP connections.

Given this analysis, it would be more accurate to say that UDP requests an extremely high bitrate than to say that DCCP requests a low bitrate. One of the crucial components of any congestion control scheme is fairness with other traffic, and linphone's UDP/RTP congestion control appears to be distinctly unfair to TCP flows, which make up the majority of traffic on the Internet. Linphone achieves an average of 16.65Mbits/sec with its UDP/RTP congestion control whereas Mathis's model predicts that a TCP connection under similar network conditions would achieve:

$$BW = \frac{1480}{0.20735}\frac{\sqrt{3/2}}{\sqrt{0.0151}}$$

$$BW = 71140.2 bytes/sec$$

$$BW = 569121.3 bits/sec$$

$$BW = 0.54 Mbits/sec$$

This is barely 1/36th of the throughput linphone achieves.

Figure 4.33 shows the average fairness of UDP, CCID 2, and CCID 3 to TCP in our tests. The error bars at 50 second intervals indicate the 95% confidence interval. Our fairness metric is the ratio of $protocol\_throughput/tcp\_throughput$ and is graphed on a log scale. Observe that UDP achieves throughput more than five times that of TCP for

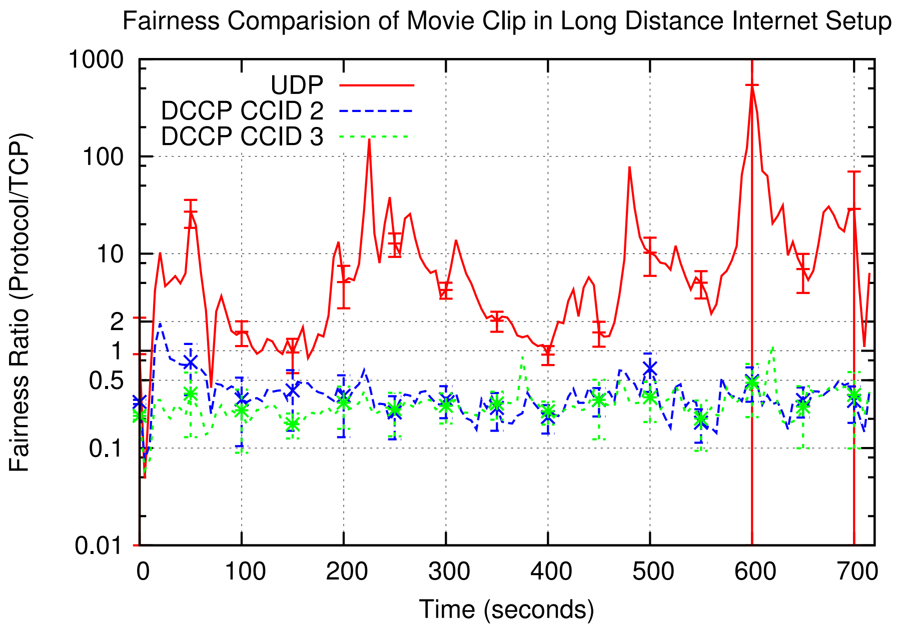


Figure 4.33: Average fairness of UDP, CCID 2, and CCID 3 to TCP over our 12 minute movie clip in the long distance Internet environment. Error bars indicate the 95% confidence interval.

significant periods and bursts to well over 100 times TCP's throughput. By comparison, both DCCP CCIDs maintain much more even fairness, hovering at around a ratio of 0.3.

The burstiness of video data has a lot to do with why both DCCP CCIDs achieve overly conservative fairness ratios. Recall that I-frames are much larger than P-frames, often by a factor of four or so, and that even within these classes there is significant size variation based on image complexity. This effectively results in sudden, and possibly major, bitrate changes every 40ms. Neither DCCP CCID is ever able to more than double its sending rate in a round trip time, in this environment about five frames, and increase much more slowly, roughly one packet per round trip, if fully utilizing their allowed bandwidth. This mismatch forces linphone to send so that the peak bitrate of the bursts match DCCP's allowed sending rate. This, of course, results in underutilization of DCCP's bandwidth at non-peaks and a corresponding conservative fairness ratio.

It is interesting to recall that our examination of fairness in the testbed environment found that UDP/RTP was distinctly less aggressive than it could have been, except at the very beginning of the connection. In our long distance Internet environment, the result is almost exactly the opposite; UDP/RTP is always too aggressive. This difference really has more to do with TCP than with UDP/RTP's congestion control algorithm. TCP's throughput is dependent on both round trip time and loss rate; as the round trip time increases, TCP's throughput decreases, given the same loss rate. The UDP/RTP algorithm, by contrast, is independent of round trip time and only dependent on loss rate. As a result, fairness varies between UDP/RTP and TCP as the round trip time changes. This is rather unfortunate as fairness with TCP is critically important in the Internet.

Both DCCP CCIDs were designed to compete fairly with TCP. This they do reasonably well. However, they achieve fair competition by making their achieved bandwidth dependent on the network round trip time in addition to the loss rate, in much the same manner as TCP. This severely limits the bandwidth they can achieve in environments with long round trip times. As a result, they compete poorly in terms of video quality with UDP in such environments.

4.4.2 Videoconference Clip

We now consider our videoconference clip in the long distance Internet environment. The same two machine setup with IPv6 tunnels to Virginia and California was utilized. Each test consisted of a linphone videoconference and Iperf connections between these two boxes such that traffic passed from our lab in Ohio, to Virginia, to California, and then back to our lab. Each test was five minutes long, which is the length of our videoconference clip. All tests used a bitrate adjustment interval of 1 second for CCID 3 and 3 seconds for CCID 2. These are the optimal bitrate adjustment intervals identified in our bitrate adjustment interval analysis.

Figures 4.34 and 4.35 show cumulative distribution functions of the received video quality, measured using PSNR and SSIM respectively, for UDP, CCID 2, and CCID 3. Each cumulative distribution function is the average of ten separate test CDFs. The error bars shown at 0.1 probability intervals indicate the 95% confidence interval.

The sharp discontinuity in the UDP PSNR cumulative distribution function along with the large increase in confidence interval size just below a probability of 0.55 is interesting. Even more interesting is how much less pronounced this discontinuity is in the UDP SSIM CDF. Recall that SSIM is designed to measure the structural content of the image instead of just total pixel differences. Hence, the likely explanation for these disparate measurements is that video streaming over UDP in this environment results in a large number of distortions that have little effect on the structural information of the scene but generate significant pixel differences.

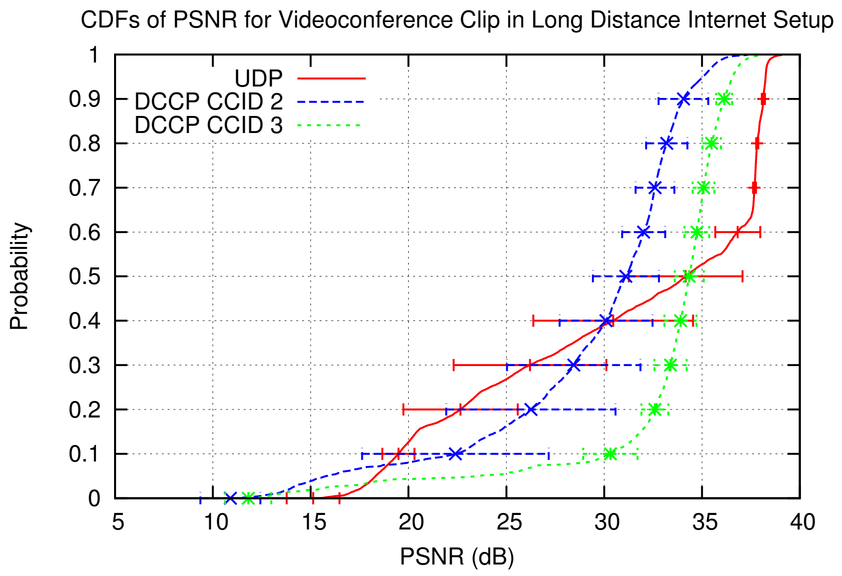


Figure 4.34: Average cumulative distribution functions for UDP, CCID 2, and CCID 3 received video quality, as measured by PSNR, for our videoconference clip in the long distance Internet environment. Error bars indicate the 95% confidence interval.

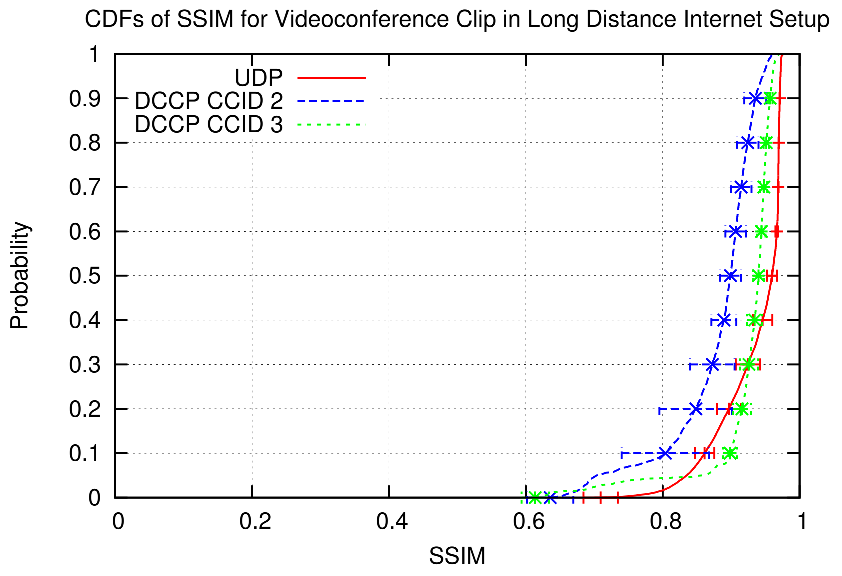
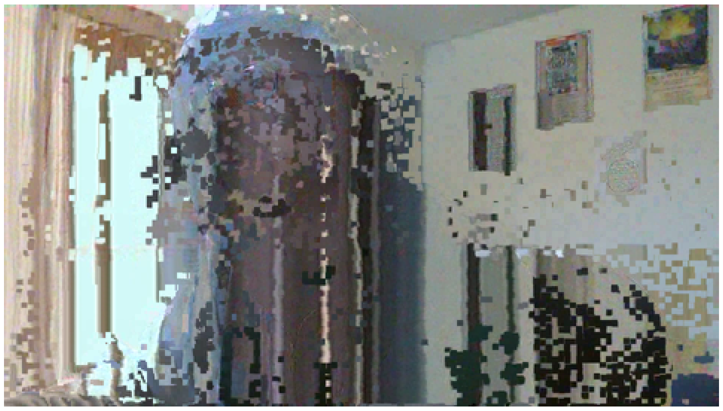
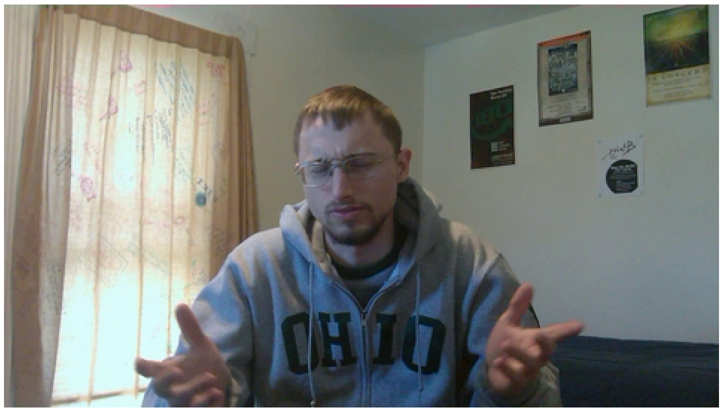


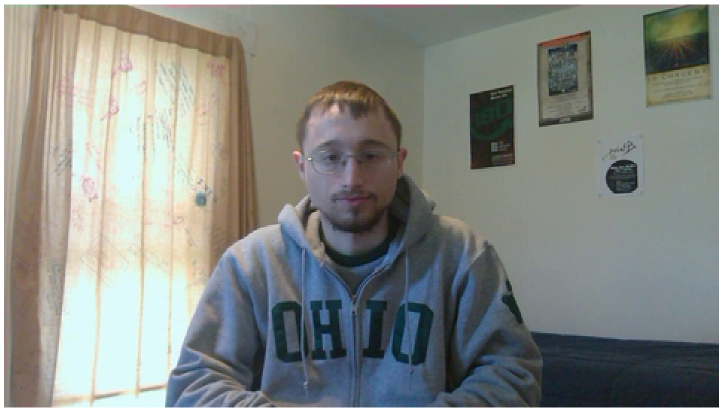
Figure 4.35: Average cumulative distribution functions for UDP, CCID 2, and CCID 3 received video quality, as measured by SSIM, for our videoconference clip in the long distance Internet environment. Error bars indicate the 95% confidence interval.



(a) 15dB PSNR, 0.64 SSIM



(b) 25dB PSNR, 0.96 SSIM



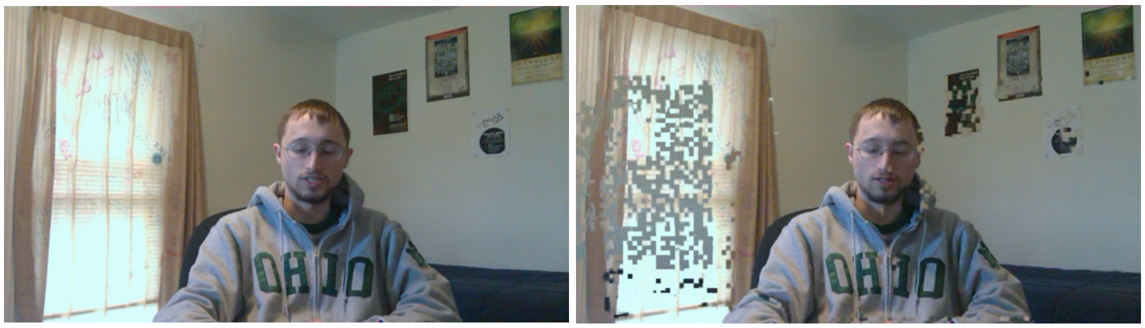
(c) 35dB PSNR, 0.95 SSIM

Figure 4.36: Example video frames with various PSNR and SSIM values. These frames are taken from our CCID 3 videoconference clip tests in our long distance Internet environment.

Figure 4.36 shows several video frames from these tests with their PSNR and SSIM values. Notice that the 25 and 35 dB PSNR frames have no noticeable artifacts and actually have nearly identical SSIM values.

Visual examination of the video received over UDP reveals numerous instances of visual artifacts like those in figure 4.37. These artifacts appear suddenly and usually last several seconds. They affect both stationary and moving objects and always consist of gray colored blocks. This implies that they result from the loss of texture data in I-frames. That would also explain the long persistence of these artifacts since they would only disappear at the next I-frame. UDP packet loss in this environment is low, only 0.13%, but only a few losses are needed to cause these artifacts since the lost data will continue to be visible over numerous P-frames. This loss rate corresponds to about two packets every second, which is more than sufficient to cause frequent artifacts of this type given that the default I-frame refresh rate is once every 10 seconds.

The SSIM cumulative distribution function, figure 4.35, indicates that this type of distortion has fairly minimal impact on image perception. We find this to the case



(a) Original (b) Corrupted

Figure 4.37: Example of the visual artifacts resulting from UDP packet loss in our videoconference clip in the long distance Internet environment.

subjectively as well. It is easy to understand what is happening in the received video despite such distortions.

Returning to the video quality CDFs in figures 4.34 and 4.35, it is apparent that CCID 3 results in much better video quality than CCID 2. As previously discussed, this occurs because CCID 3 responds less aggressively to lost packets than CCID 2 and because CCID 3 makes its sending rate available to the application via a socket option. This allows the application to adjust its sending rate without having to wait for a send queue overflow, like with CCID 2. As a result, CCID 3 not only sends more packets in total than CCID 2 but also has fewer packets rejected, as table 4.6 shows. The CCID 3 quality CDFs exhibit a much smaller confidence interval than the CCID 2 CDFs, indicating more even video quality, for the same reason.

In our long distance Internet environment, CCID 2 actually achieves lower video quality than in previous environments. The longer round trip time causes CCID 2 to increase its allowed sending rate more slowly than in our other environments. Further, a single congestion window now has to hold several frames, instead of only one, which decreases CCID 2's flexibility. CCID 3 also becomes less responsive in the face of increasing round trip time, but its less aggressive rate-based congestion control and socket option application feedback help to mitigate the effects.

Table 4.6: Packet Statistics for our Videoconference Clip in the Long Distance Internet Environment

	Sent Packets	Received	% Received	Lost	% Lost	Rejected	% Rejected
UDP	752362	751417	99.87%	945	0.13%	0	0%
CCID 2	7768420	38676	56.53%	93	0.14%	29650	43.34%
CCID 3	93973	81324	86.54%	71	0.08%	12577	13.38%

Linphone achieves significantly higher throughput when streaming our videoconference clip over UDP instead of either DCCP CCID. Recall that we observed this same behavior with our movie clip in the previous section. Table 4.6 shows that UDP sends nearly eight times as many packets as CCID 3 and eleven times as many as CCID 2. This allows UDP to obtain significantly higher video quality. However, this higher throughput is significantly unfair to TCP and other background traffic, as discussed previously.

Interestingly, both DCCP CCIDs result in more even video quality compared to UDP when streaming our video conference clip. The PSNR versus time graph for a representative test in figure 4.38 clearly illustrates this. DCCP may not achieve the absolute video quality that UDP does, but it certainly achieves a more even video quality, especially when using CCID 3. The dramatic variation in UDP's video quality occurs

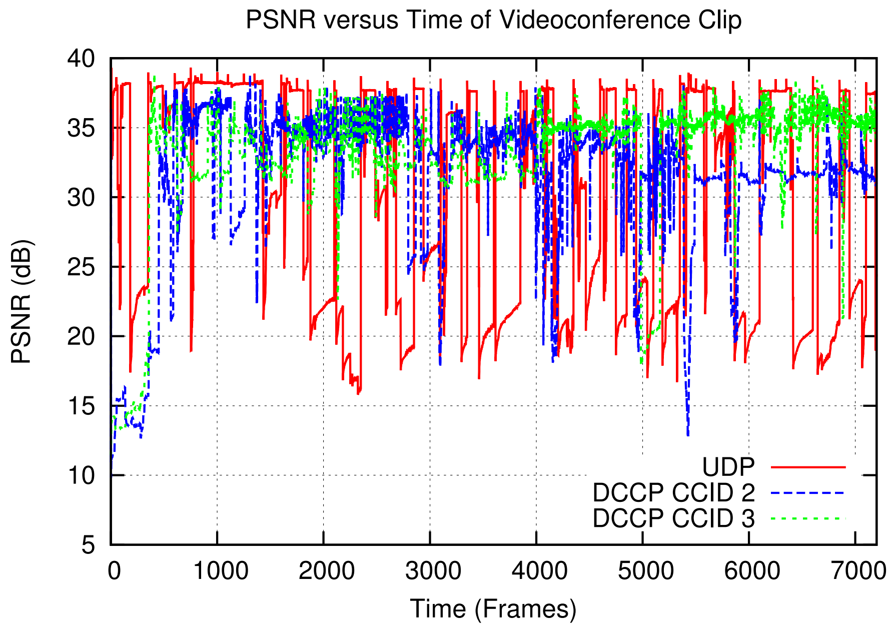


Figure 4.38: PSNR versus time for representative experiments with our videoconference clip over UDP, CCID 2, and CCID 3 in the long distance Internet environment.

because of the previously discussed visual artifacts resulting from random packet losses in I-frame texture data. Both DCCP CCIDs react much more harshly to network packet loss than UDP does. As a result, the frequency of these artifacts is much lower when using DCCP.

DCCP does suffer from significant packet rejection because of send queue overflow. However, this tends to occur in large bursts, reducing the number of frames affected. Many of these events result in undecodable frames which have a smaller impact on video quality than long duration visual artifacts. Further, when packets are lost or rejected in DCCP, it causes a bitrate reduction which forces a new I-frame that will refresh the video and reduce the persistence of these artifacts.

This examination of the performance of videoconference type streaming over UDP and DCCP in a long distance Internet environment uncovered an unusual difference between the cumulative distribution functions for PSNR and SSIM in our UDP streaming tests. This difference was found to be the be result of a visual artifact caused by the loss of texture data in I-frames as a result of random network packet loss. These visual artifacts are not apparent using either DCCP CCID because the DCCP congestion control algorithms are more sensitive to packet losses and thus slow down to a rate where these artifacts are rare. Just as with our movie clip, UDP achieves higher throughput than either DCCP CCID, allowing better video quality, but at the expense of fairness to competing traffic.

5 Conclusions and Future Work

5.1 Conclusions

This thesis examines the performance of real-time video streaming over DCCP as compared to streaming over UDP with RTP congestion control. We examine the performance in a testbed environment as well as Internet environments with long and short round trip times in order to consider data from a diverse set of environments. To account for different types and complexities of video, two different video clips were used. One being representative of typical videoconferencing content while the other was a movie clip. These experiments provide a solid basis for making conclusions about DCCP's effectiveness for video streaming.

This work improves upon the literature by comparing the performance of DCCP to an application level congestion control scheme used in a real streaming media application. Further, this comparison is done using real applications in a variety of real network environments, instead of using simulations that may not accurately reflect all the complexities of real applications in real environments. In addition, we consider both media quality and network traffic characteristics in our analysis and conclusions. Surprisingly, these two disciplines seem to rarely communicate, even about joint problems like congestion control for streaming media.

This work has also resulted in a much improved implementation of DCCP in the Linux kernel, through the contribution of nine patches fixing a variety of bugs and performance issues. Further, we have contributed our implementation of DCCP support for linphone back to the open source community, and it has been accepted for merging into the mainline. It will probably be available in linphone 3.7. This makes linphone one of the few applications that support DCCP and one of even fewer that support it without a 3rd-party, semi-maintained patch.

Our results show that both DCCP CCIDs provide much more even video quality, as measured by both PSNR and SSIM, than UDP/RTP does. This is a significant benefit for DCCP, as quality evenness may be more important than total quality, particularly once some minimum quality is reached.

Further, in those cases where UDP and DCCP are achieving roughly their fair share of network bandwidth, DCCP achieves similar or better video quality, as measured by PSNR and SSIM, compared to UDP. This showcases DCCP's value for real-time, streaming video; DCCP responds to network conditions faster than UDP/RTP would, allowing it to achieve better video quality.

However, UDP/RTP congestion control, as implemented in Linphone and probably other applications, often utilizes much more than its fair share of network bandwidth. This can be by as much as a factor of 5 or 10. This allows UDP/RTP to achieve better video quality, but at the cost of friendliness to other applications on the Internet.

A significant part of this problem is that this form of UDP/RTP congestion control is independent of round trip time, while most other congestion control mechanisms utilize the round trip time as a crucial part of their algorithm. As a result, while this UDP/RTP algorithm may be fair to TCP traffic in one environment, it will almost certainly be unfair in other environments.

This is not the only unfairness issue that UDP/RTP suffers. This congestion control algorithm also experiences a significant throughput spike at the beginning of the connection while the correct sending rate is being sought. This burst may last up to a minute and result in unfairness on the order of 100:1. We have even observed zero throughput for multiple seconds on competing TCP connections during this period.

By contrast, both DCCP CCIDs are designed to be fair to other traffic on the network and to require no intervention from the application programmer to achieve this. They accomplish that goal reasonably well, although both tend to be overly conservative when competing with TCP traffic. This conservative behavior results from the burstiness of video traffic; given an even application sending rate, the literature shows that both CCIDs compete quite well with TCP.

Another important conclusion from this study is that using send queue backpressure as feedback to the application is problematic. It results in a significant number of rejected packets since that is the only way to signal the application to slow down. In addition, this approach delays application reaction until it is too late to avoid rejecting more packets from the frame currently being encoded and requires slowing down more than strictly necessary in order to drain the send queue.

Further, while backpressure allows the application to determine when it needs to slow down, this method of feedback provides no signal to the application when it could speed up. As a result, it becomes necessary to implement some sort of gradual increase mechanism. This is exactly the kind of application infrastructure that DCCP was designed to eliminate and encourages the application to be overly conservative because of the harsh penalty for being aggressive.

A much better alternative is making the currently allowed sending rate, or some smoothed version of it, available to the application via a socket option, as done by CCID 3. While this approach is not perfect, being particularly vulnerable to rapid rate oscillations, it is much simpler, more accurate, and generally achieves better performance than backpressure feedback.

In addition, really taking advantage of DCCP and its more frequent and more granular bitrate updates requires a video encoder that allows the requested bitrate to be varied on the fly, without re-initializing the encoder and forcing a new, very large, I-frame. If updating the bitrate requires sending a new I-frame, then there is a trade off between setting a more accurate bitrate and dealing with the burst of data that the resulting I-frame will cause. This trade off becomes particularly important in long round trip time environments where DCCP adjusts its sending rate relatively slowly.

Finally, DCCP provides significant benefits to application developers by freeing them from having to design, implement, test, and maintain a complex congestion control algorithm. DCCP provides effective, well tested congestion control essentially for free; all application developers have to do is utilize the feedback it provides to adjust the sending rate of their application. We have shown that this is fairly easy to do for video streaming and results in reasonable quality and improved fairness to other applications. These benefits may also make DCCP attractive to other applications with real-time constraints and a need for congestion control. Augmented reality systems and online gaming are two prime examples.

DCCP was designed to offer congestion control to real-time applications. It provides several benefits for real-time, video streaming applications, including more even video quality. Although existing UDP/RTP congestion control mechanisms can achieve better quality, they do so by being significantly unfair to other network traffic. In addition, such mechanisms require significant programmer time to design and implement. With a few improvements and a streaming-optimized video encoder, DCCP would be extremely effective for real-time, streaming video applications.

5.2 Future Work

There remains plenty of future research that could be done with DCCP in streaming media applications. Probably one of the most promising would be the design of a new CCID more optimized to the needs of video streaming. A rate-based design similar to CCID 3 would probably be the appropriate place to start because of the milder response to congestion and inherent rate-based nature of video traffic. One of the most important things for such a video-optimized CCID to have is an understanding of what we refer to as chunked transfer, the idea that an application will try to send a group of packets now and then not send anything for the next x milliseconds. A CCID that understands chunked transfer could maintain a more even sending rate and do intelligent queuing to avoid rejecting packets that could actually be sent.

One of the major problems with using CCID 3 at the moment is that it cannot increase its sending rate fast enough to handle MPEG-4's burstiness. The TFRC specification actually constrains CCID 3 to twice its sending rate in the last round trip time. This requirement cannot be completely removed because sudden bursts, even for short periods, can cause router queue overflows and network loss, which will be interpreted by other congestion controlled protocols as a sign that they need to slow down. However, it may be possible to allow a CCID to linearly increase its sending rate over the course of either a round trip or an application data chunk, as long as the CCID's rate is below the TCP-friendly computed rate. This would allow greater flexibility to handle bursty data streams while smoothing those bursts out on the network. It would also reduce the round trip time dependence of CCID 3, which would improve performance in long round trip environments.

Another important feature of such a video-optimized CCID would be some form of time and priority based send queue that would throw away older data on queue overflow. The ability to give preferential treatment to I-frames would also be beneficial.

Another area for future work would be the design and implementation of a socket option to allow an application to query DCCP CCID 2's allowed sending rate. This sending rate could be trivially computed as $\frac{window\_size * avg\_pkt\_size}{RTT}$; however, some form of smoothing would almost certainly be needed and there may be a smarter computation. CCID 3 would also likely benefit from similar smoothing for the sending rate it makes available via socket option.

In addition, the determination of a model or heuristic for setting the bitrate adjustment interval would also be a beneficial area for future work. Our initial exploration of this space in section 4.3 suggests a dependence on round trip time and shows that the two DCCP CCIDs require different values. This sufficed for determining optimal values for use in our tests; however, we were not able to generalize this to a global model.

A final area for future work would be a detailed analysis of video traffic burstiness. Although burstiness is a well known phenomenon in network traffic, good metrics for burstiness remain illusive. Further, examination of video burstiness is nearly nonexistent in the literature.

References

[Ado13] Adobe Systems Incorporated. Video Learning Guide for Flash: NTSC and PAL video standards — Adobe Developer Connection, 2013. URL: https://www.adobe.com/devnet/flash/learning_guide/video/part06.html

[Adv95] Advanced Television Systems Committee Inc. Standard for Coding 25/50 Hz Video. Technical report, Advanced Television Systems Committee Inc, 1995.

[Adv07] Advanced Television Systems Committee Inc. A / 53: ATSC Digital Television Standard , Parts 1 - 6. Technical report, Advanced Television Systems Committee Inc, 2007.

[AML03] Toufik Ahmed, Ahmed Mehaoua, and Vincent Lecuire. Streaming MPEG-4 Audio Visual Objects Using TCP-Friendly Rate Control and Unequal Error Protection. In *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference on*, pages 1–5. IEEE, 2003.

[AMM09] Muhmmad Azad, Rashid Mahmood, and Tahir Mehmood. A Comparative Analysis of DCCP Variants (CCID2, CCID3), TCP and UDP for MPEG4 Video Applications. In *Information and Communication Technologies, 2009. ICICT '09. International Conference on*, pages 40–45. IEEE, 2009. doi:10.1109/ICICT.2009.5267215

[APB09] Mark Allman, Vern Paxson, and Ethan Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[BA05] Ali C Begen and Yucel Altunbasak. Estimating Packet Arrival Times in Bursty Video Applications. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 767 – 770. IEEE, 2005. doi:10.1109/ICME.2005.1521536

[BAFW03] Ethan Blanton, Mark Allman, Kevin Fall, and Lili Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), April 2003.

[BBM08] Saleem Bhatti, Martin Bateman, and Dimitris Miras. A Comparative Performance Evaluation of DCCP. In *Performance Evaluation of Computer and Telecommunication Systems, 2008. SPECTS 2008. International Symposium on*, pages 433–440, 2008.

[BDS96] Ingo Busse, Bernd Deffner, and Henning Schulzrinne. Dynamic QoS Control of Multimedia Applications based on RTP. *Computer Communications*, 19(1):49–58, 1996. doi:10.1016/0140-3664(95)01038-6

[Bel13] Belledonne Communications. Belledonne communications: Linphone, SIP video phone client - Voice and Video over IP experts, 2013. URL: http://www.belledonne-communications.com/linphone.html

[Bra89] Robert Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989.

[CCZ03] Jae Chung, Mark Claypool, and Yali Zhu. Measurement of the Congestion Responsiveness of RealPlayer Streaming Video Over UDP. In *Proceedings of the Packet Video Workshop*. Citeseer, 2003.

[CGL$^+$01] Gregory Conklin, Gary Greenbaum, Karl Lillevold, Alan Lippman, and Yuriy Reznik. Video Coding for Streaming Media Delivery on the Internet. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11(3):269–281, 2001.

[Cis13] Cisco Systems. Quality of Service Design for TelePresence, 2013. URL: http://www.cisco.com/en/US/docs/solutions/Enterprise/Video/tpqos.html

[CMP08] Luca De Cicco, Saverio Mascolo, and Vittorio Palmisano. Skype Video Responsiveness to Bandwidth Variations. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 81–86. ACM, 2008. doi:10.1145/1496046.1496065

[CMY09] Hafiz Muhammad Omer Chughtai, Shahzad A Malik, and Muhammad Yousaf. Performance Evaluation of Transport Layer Protocols for Video Traffic over WiMax. In *Multitopic Conference, 2009. INMIC 2009. IEEE 13th International*, pages 1–6. IEEE, 2009. doi:10.1109/INMIC.2009.5383117

[Con12] Conservatorio di musica G. Tartini. LOLA: Low Latency Audio Visual Streaming System Installation & User's Manual. Technical report, Conservatorio di musica G. Tartini, Trieste, Italy, 2012.

[CPM07] Nicola Cranley, Philip Perry, and Liam Murphy. Dynamic Content-Based Adaptation of Streamed Multimedia. *Journal of Network and Computer Applications*, 30(3):983–1006, August 2007. doi:10.1016/j.jnca.2005.12.006

[CZM$^+$10] An Chan, Kai Zeng, Prasant Mohapatra, Sung-ju Lee, and Sujata Banerjee. Metrics for Evaluating Video Streaming Quality in Lossy IEEE 802 . 11 Wireless Networks. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010. doi:10.1109/INFCOM.2010.5461979

[Dig12] Digital Video Broadcasting. Digital Video Broadcasting(DVB): Specification for the use of Video and Audio Coding in Broadcasting Applications based on the MPEG-2 Transport Stream. Technical report, Digital Video Broadcasting, 2012.

[DR06] B D'Auria and SI Resnick. Data Network Models of Burstiness. *Advances in Applied Probability*, 38(2):373–404, 2006.

[dSOPdM08] Leandro Melo de Sales, Hyggo Oliveira, Angelo Perkusich, and Arnaldo Carvalho de Melo. Measuring DCCP for Linux against TCP and UDP With Wireless Mobile Devices. In *Ottawa Linux Symposium*, pages 163–177, Ottawa, Ontario, 2008.

[FB02] Nick Feamster and Hari Balakrishnan. Packet Loss Recovery for Streaming Video. In *12th International Packet Video Workshop*, pages 9–16, 2002.

[FdFPM10] Carlos A. Froldi, Nelson L. S. da Fonseca, Carlos Papotti, and Daniel A. G. Manzato. Performance Evaluation of the DCCP Protocol in High-Speed Networks. In *Computer Aided Modeling, Analysis and Design of Communication Links and Networks (CAMAD), 2010 15th IEEE International Workshop on*, pages 41–46. IEEE, December 2010. doi:10.1109/CAMAD.2010.5686967

[FF99] Sally Floyd and Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999. doi:10.1109/90.793002

[FHK06a] Sally Floyd, Mark Handley, and Eddie Kohler. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), 2006.

[FHK06b] Sally Floyd, Mark Handley, and Eddie Kohler. Problem Statement for the Datagram Congestion Control Protocol (DCCP). RFC 4336 (Informational), 2006.

[FHPW08] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348 (Proposed Standard), September 2008.

[FK06] Sally Floyd and Eddie Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341 (Proposed Standard), 2006.

[FK09] Sally Floyd and Eddie Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion ID 4: TCP-Friendly Rate Control for Small Packets (TFRC-SP). RFC 5622 (Experimental), 2009.

[FKP06] Sally Floyd, Eddie Kohler, and Jitendra Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), 2006.

[Flo00] Sally Floyd. Congestion Control Principles. RFC 2914 (Best Current Practice), September 2000.

[FMMP00] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Matthew Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[GDK$^+$05] Xiaoyuan Gu, Matthias Dick, Zefir Kurtisi, Ulf Noyer, and Lars Wolf. Network-centric Music Performance: Practice and Experiments. *Communications Magazine, IEEE*, 43(6):86–93, 2005. doi:10.1109/MCOM.2005.1452835

[GDW06] Xiaoyuan Gu, Pengfei Di, and Lars Wolf. Performance Evaluation of DCCP: A Focus on Smoothness and TCP-friendliness. *Annals of Telecommunications*, 61(1):46–71, February 2006. doi:10.1007/BF03219968

[GMM04] Stefan A Goor, Seán Murphy, and Liam Murphy. Experimental Performance Analysis of RTP-Based Transmission Techniques for MPEG-4. In *14th International Packet Video Workshop*, 2004.

[Goo05] Stefan A Goor. *Experimental Performance Analysis of RTP-Based Approaches for Low-bitrate Transmission of MPEG-4 Video Content*. Masters thesis, University College Dublin, 2005.

[Hag03] Joacim Haggmark. [dccp] FreeBSD implementation. Email to IETF DCCP mailing list, October 2003. URL: https://www.ietf.org/mail-archive/web/dccp/current/msg00508.html

[HAOS01] Duke Hong, Celio Albuquerque, Carlos Oliveira, and Tatsuya Suda. Evaluating the Impact of Emerging Streaming Media Applications on TCP/IP Performance. *Communications Magazine, IEEE*, 39(4):76–82, 2001. doi:10.1109/35.917507

[Hub12] Ian Hubert. Tears of Steel, 2012.

[Hur13] Hurricane Electric Internet Services. Hurricane Electric Free IPv6 Tunnel Broker, 2013. URL: http://tunnelbroker.net/

[Int99] International Telecommunication Union. ITU-T P.910. Technical report, International Telecommunication Union, 1999.

[Int01] International Committee for Information Technology Standards. ISO / IEC 14496-1. Technical report, International Standarization Organization, 2001.

[Int02a] International Committee for Information Technology Standards. ISO / IEC 14496-2. Technical report, International Standarization Organization, 2002.

[Int02b] International Committee for Information Technology Standards. ISO/IEC 14496-8. Technical report, International Standarization Organization, 2002.

[Int02c] International Organization for Standardization. ISO/IEC N4668. Technical report, International Standarization Organization, 2002.

[Int08] International Telecommunication Union. ITU-T J.247. Technical report, International Telecommunication Union, 2008.

[Jac88] Van Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988. doi:10.1145/52325.52356

[Jer13] Samuel Jero. [RFC][PATCH] tfrc: Correct 2nd Loss Interval Handling. Email to the Linux DCCP mailing list, April 2013. URL: http://www.spinics.net/lists/dccp/msg04527.html

[JLddM10] G. D G Jaime, R.M.M. Leao, E. de Souza e Silva, and J.B.B. de Marca. Effect of Varying the Intra-Frame Packet Burstiness on the Performance of Wireless Video Streaming. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6, 2010. doi:10.1109/GLOCOM.2010.5683762

[JSB$^+$09] Van Jacobson, Diana K Smetters, Nicholas H Briggs, Michael F Plass, Paul Stewart, James Thornton, and Rebecca L Braynard. VoCCN: Voice-over Content-Centric Networks. In *Proceedings of the 2009 workshop on Re-architecting the internet*, pages 1–6. ACM, 2009. doi:10.1145/1658978.1658980

[KC11] Kyungtae Kim and Young-June Choi. Performance Comparison of Various VoIP Codecs in Wireless Environments. In *Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication*. ACM, 2011. doi:10.1145/1968613.1968718

[Ker07] Kernel Newbies. Linux 2 6 14, 2007. URL: http://kernelnewbies.org/Linux_2_6_14

[Ker12] Kernel Newbies. Linux 3.2 - Linux Kernel Newbies, 2012. URL: http://kernelnewbies.org/Linux_3.2

[KNF+00] Yoshirkiro Kikuchi, Toshiyuki Nomura, Shigeru Fukunaga, Yoshinori Matsui, and Hideaki Kimata. RTP Payload Format for MPEG-4 Audio/Visual Streams. RFC 3016 (Proposed Standard), November 2000.

[KT97] Marwan Krunz and Satish K Tripathi. On the Characterization of VBR MPEG Streams. In *SIGMETRICS '97 Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 192–202. ACM, 1997. doi:10.1145/258612.258688

[LAG03] Yi J Liang, John G Apostolopoulos, and Bernd Girod. Analysis of Packet Loss for Compressed Video: Does Burst-Length Matter? In *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on*, pages 684–687 vol.5, 2003. doi:10.1109/ICASSP.2003.1200063

[Lib13] Libav. Libav, 2013. URL: https://libav.org/

[Lin13] Linphone. Linphone, an Open-Source Video SIP Phone, 2013. URL: http://www.linphone.org/

[LKP08] Aggelos Lazaris, Polychronis Koutsakis, and Michael Paterakis. A New Model for Video Traffic Originating from Multiplexed MPEG-4 Videoconference Streams. *Performance Evaluation*, 65(1):51–70, January 2008. doi:10.1016/j.peva.2007.02.004

[LL08] Y.C. Lai and Yuan-cheng Lai. DCCP Congestion Control with Virtual Recovery to Achieve TCP-Fairness. *IEEE Communications Letters*, 12(1):50–52, January 2008. doi:10.1109/LCOMM.2008.071421

[LLA+04] Mingzhe Li, Choong-Soo Lee, Emmanuel Agu, Mark Claypool, and Robert Kinicki. Performance Enhancement of TFRC in Wireless Ad Hoc Networks. In *DMS'04: Proceedings of the 10th International Conference on Distributed Multimedia Systems*, 2004.

[LM03] Olaf Landsiedel and Gary Minden. MPEG-4 for Interactive Low-delay Real-time Communication. Technical Report ITTC-FY2004-TR-23150-10, University of Kansas Information and Telecommunication Technology Center, 2003.

[LTG99] F Le Le, F Toutain, and C Guillemot. Packet loss resilient MPEG-4 compliant video coding for the Internet. *Signal Processing: Image Communication*, (15):35–56, 1999.

[LTHW10] Ying-Dar Lin, Chien-Chao Tseng, Cheng-Yuan Ho, and Yu-Hsien Wu. How NAT-compatible are VoIP Applications? *Communications Magazine, IEEE*, 48(12):58–65, 2010. doi:10.1109/MCOM.2010.5673073

[LV91] S. Low and P. Varaiya. A Simple Theory of Traffic and Resource Allocation in ATM. In *IEEE Global Telecommunications Conference GLOBECOM '91: Countdown to the New Millennium. Conference Record*, pages 1633–1637. IEEE, 1991. doi:10.1109/GLOCOM.1991.188641

[Mat04] Nils-Erik Mattsson. *A DCCP module for ns-2*. Masters thesis, Lulea Tekniska Universitet, ISSN, 2004.

[MMBK10] David Mills, Jim Martin, Jack Burbank, and William Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[Moc87] Paul Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987.

[Mov] Moving Pictures Experts Group. Advanced Video Coding — MPEG. URL: http://mpeg.chiariglione.org/standards/mpeg-4/advanced-video-coding

[Mov13] Moving Pictures Experts Group. MPEG-4 — MPEG, 2013. URL: http://mpeg.chiariglione.org/standards/mpeg-4

[MSMO97] Matt Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM Computer Communications Review*, 27(3):67–82, 1997. doi:10.1145/263932.264023

[NAT06] P Navaratnam, N Akhtar, and R Tafazolli. On the Performance of DCCP in Wireless Mesh Networks. In *Proceedings of the 4th ACM international workshop on Mobility management and wireless access*, pages 144–147. ACM, 2006. doi:10.1145/1164783.1164811

[Nat10] National Laboratory for Advanced Network Research. Iperf, 2010. URL: http://iperf.sourceforge.net/

[NHG10] Shahrudin Awang Nor, Suhaidi Hassan, and Osman Ghazali. Friendliness of DCCP towards TCP over large delay link networks. In *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, volume 5, pages 286–291, 2010. doi:10.1109/ICETC.2010.5530062

[Ori10] Emanuele Oriani. qpsnr official homepage, 2010. URL: http://qpsnr.youlink.org/

[Ost03] Shawn Ostermann. tcptrace - Official Homepage, 2003. URL: http://www.tcptrace.org/

[OWS+06] Joerg Ott, Stephen Wenger, Noriyuki Sato, Carsten Burmeister, and Jose Rey. Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF). RFC 4585 (Proposed Standard), July 2006.

[PACS11] Vern Paxson, Mark Allman, Jerry HK Chu, and Matt Sargent. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[Per99] Alan Percy. Understanding Latency in IP Telephony. Technical report, Brooktrout Technology, 1999.

[PG06] Colin Perkins and Ladan Gharai. RTP and the Datagram Congestion Control Protocol. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 1521–1524. IEEE, 2006. doi:10.1109/ICME.2006.262832

[Phe08] Tom Phelan. DCCP-TP Home Page, May 2008. URL: http://www.phelan-4.com/dccp-tp/tiki-index.php

[Pos80] Jon Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[Pos81] Jon Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981.

[PUN12] Stefan Paulsen, Tadeus Uhl, and Krzysztof Nowicki. MPEG-4/AVC versus MPEG-2 in IPTV. In *Proceedings of the International Conference on Signal Processing and Multimedia Application*, pages 27–30, Rome, 2012.

[RCPC99] Hayder Radha, Yingwei Chen, Kavitha Parthasarathy, and Robert Cohen. Scalable Internet video using MPEG-4. *Signal Processing: Image Communication*, 15(1-2):95–126, September 1999. doi:10.1016/S0923-5965(99)00026-0

[Ren07] Gerrit Renker. Linux/Documentation/networking/dccp.txt, 2007. URL: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/networking/dccp.txt?id=refs/tags/v3.2.45

[Ren11] Gerrit Renker. dccp test-tree [ANNOUNCE] dccp: DCCP-Cubic / CCID-5 subtree available. Email to Linux DCCP mailing list, 2011. URL: http://www.spinics.net/lists/dccp/msg04488.html

[RFB01] KK Ramakrishnan, Sally Floyd, and David Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001.

[RPB$^+$08] Merhrnoush Rahmani, Andrea Pettiti, Ernst Biersack, Eckehard Steinbach, and Joachim Hillebrand. A Comparative Study of Network Transport Protocols for In-Vehicle Media Streaming. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 441–444. IEEE, 2008. doi:10.1109/ICME.2008.4607466

[SBJL08] Golam Sarwar, Roksana Boreli, Guillaume Jourjon, and Emmanuel Lochin. Improvements in DCCP Congestion Control for Satellite Links. In *2008 IEEE International Workshop on Satellite and Space Communications*, pages 8–12. IEEE, October 2008. doi:10.1109/IWSSC.2008.4656732

[SBL09] Golam Sarwar, Roksana Boreli, and Emmanuel Lochin. Performance of VoIP with DCCP for satellite links. In *Communications, 2009. ICC '09. IEEE International Conference on*, pages 1–5. IEEE, 2009. doi:10.1109/ICC.2009.5199334

[Sea04] Michael Searles. *Probe Based Dynamic Server Selection for Multimedia Quality of Service*. Masters thesis, University College Dublin, 2004.

[SF07] Arjuna Sathiaseelan and Gorry Fairhurst. Performance of VoIP using DCCP over a DVB-RCS Satellite Network. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 13–18. IEEE, 2007.

[SFCJ03] Henning Schulzrinne, Ron Frederick, Stephen Casner, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), 2003.

[Sky13] Skype Communications. Help for Skype: How much bandwidth does Skype need?, 2013. URL: https://support.skype.com/en/faq/FA1417/how-much-bandwidth-does-skype-need

[SLB07] G Sarwar, E Lochin, and R Boreli. Experimental Performance of DCCP over Live Satellite and Long Range Wireless Links. In *Communications and Information Technologies, 2007. ISCIT '07. International Symposium on*, pages 689–694. IEEE, 2007. doi:10.1109/ISCIT.2007.4392105

[Ste97] W Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001 (Proposed Standard), January 1997.

[Tcp13] Tcpdump. Tcpdump/Libpcap public repository, 2013. URL: http://www.tcpdump.org/

[The09] The Linux Foundation. dccp — The Linux Foundation, 2009. URL: http://www.linuxfoundation.org/collaborate/workgroups/networking/dccp

[The13] The FreeBSD Project. IPFW, 2013. URL: http://www.freebsd.org/doc/handbook/firewalls-ipfw.html

[TKI+05] Shigeki Takeuchi, Hiroyuki Koga, Katsuyoshi Iida, Youki Kadobayashi, and Suguru Yamaguchi. Performance Evaluations of DCCP for Bursty Traffic in Real-Time Applications. In *Applications and the Internet, 2005. Proceedings. The 2005 Symposium on*, pages 142–149. IEEE, 2005. doi:10.1109/SAINT.2005.51

[VN04] Ricardo N Vaz and Mario S Nunes. Selective Frame Discard for Video Streaming over IP Networks. In *Proceedings of the 7th Conference on Computer Networks (CRC2004)*, Leira, Portugal, 2004. doi:10.1.1.1.111.883

[VSB06] J Van Velthoven, K Spaey, and C Blondia. Performance of Constant Quality Video Applications using the DCCP Transport Protocol. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, page 511, Tampa, FL, 2006. Ieee. doi:10.1109/LCN.2006.322148

[Wai08] Wainhouse Research. Polycom's Lost Packet Recovery (LPR) Capability. Technical report, Wainhouse Research, 2008.

[Wal91] Gregory K Wallace. The JPEG Still Picture Compression Standard. *Consumer Electronics, IEEE Transactions on*, 38(1):xviii – xxxiv, 1991. doi:10.1109/30.125072

[WBSS04] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, April 2004. doi:10.1109/TIP.2003.819861

[WH06] Joerg Widmer and Mark Handley. TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification. RFC 4654 (Experimental), August 2006.

[WHZ+00] Dapeng Wu, YT Yiwei Thomas Hou, Wenwu Zhu Zhu, HJ Hung-Ju Lee, Tihao Chaing, Ya-Qin Zhang, and Jonathan H Chao. On End-to-End Architecture for Transporting MPEG-4 Video Over the Internet. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(6):923 – 941, 2000. doi:10.1109/76.867930

[WKD11] Daniel Wilson, Terry Koziniec, and Mike Dixon. Quantitative Analysis of the Effects Queuing has on a CCID3 Controlled DCCP Flow. In *Computer Applications and Industrial Electronics (ICCAIE), 2011 IEEE International Conference on*, pages 529–534. IEEE, December 2011. doi:10.1109/ICCAIE.2011.6162191

[WM08] Stefan Winkler and Praveen Mohandas. The Evolution of Video Quality Measurement: From PSNR to Hybrid Metrics. *IEEE Transactions on Broadcasting*, 54(3):660–668, September 2008. doi:10.1109/TBC.2008.2000733

[WSL00] Benjamin W Wah, Xiao Su, and Dong Lin. A Survey of Error-Concealment Schemes for Real-Time Audio and Video Transmissions over the Internet. In *Multimedia Software Engineering, 2000. Proceedings. International Symposium on*, pages 17–24. IEEE, 2000. doi:10.1109/MMSE.2000.897185

[WW07] Chungyi Wang and Qunicy Wu. Information Hiding in Real-time VoIP Streams. In *Ninth IEEE International Symposium on Multimedia*, pages 255 – 262. IEEE, 2007. doi:10.1109/ISM.2007.4412381

[WW08] Jiayu Wang and Quincy Wu. Porting VoIP applications to DCCP. In *Proceedings of the International Conference on Mobile Technology, Applications, and Systems*, number 1, pages 8:1—-8:6, New York, New York, USA, 2008. ACM Press. doi:10.1145/1506270.1506281

[Zol02] Eiman Zolfaghari. Work on the Datagram Congestion Control Protocol, May 2002. URL: http://cseweb.ucsd.edu/~tsohn/projects/dccp/

[ZXH+12] Xinggong Zhang, Yang Xu, Hao Hu, Yong Liu, Zonging Guo, and Yao Wang. Profiling Skype Video Calls: Rate Control and Video Quality. In *2012 Proceedings IEEE INFOCOM*, pages 621–629. IEEE, March 2012. doi:10.1109/INFCOM.2012.6195805

Appendix A: Linphone Modifications

A.1 DCCP Support

Adding basic DCCP support to Linphone consisted of modifying the oRTP library to utilize DCCP sockets instead of UDP sockets. This was complicated by the fact that DCCP is a connection oriented protocol, requiring *connect()* and *accept()* calls, while UDP is connectionless.

The relevant sections of code, all from oRTP/src/rtpsession_inet.c, appear below. This code is based off of oRTP revision b1590514c98d33e5464d46317fdeaec52f778de7 from git://git.linphone.org/linphone.git pulled on 2012-12-28.

oRTP/src/rtpsession_inet.c

```
#define CCID2_REJECT_DELAY_MSEC 3000
#define CCID3_REJECT_DELAY_MSEC 1000
#define BW_SAMPLE_PERIOD_MSEC 1000
#define DCCP_SERVICE_CODE 1381257281
#define ORTP_DCCP 1
#ifndef SOL_DCCP
#define SOL_DCCP 269
#endif

#ifdef ORTP_DCCP
#include <linux/dccp.h>
/**   tfrc_tx_info    -    TFRC Sender Data Structure
 */
struct tfrc_tx_info {
  __u64 tfrctx_x;
  __u64 tfrctx_x_recv;
  __u32 tfrctx_x_calc;
  __u32 tfrctx_rtt;
  __u32 tfrctx_p;
  __u32 tfrctx_rto;
  __u32 tfrctx_ipi;
};
#endif

static bool_t try_connect(int fd, const struct sockaddr *dest,
   socklen_t addrlen){
  if (connect(fd,dest,addrlen)<0){
    ortp_warning("Could not connect() socket: %s",getSocketError
       ());
    return FALSE;
  }
  return TRUE;
```

```
}

int set_dccp_q_len(int sock, int len){
  bool_t done=FALSE;
#ifdef ORTP_DCCP
  int err;
  int val=len;
  if (len>0){
    err = setsockopt(sock, SOL_DCCP, DCCP_SOCKOPT_QPOLICY_TXQLEN,
        (void *)&val, sizeof(val));
    if (err < 0) {
      ortp_error("Failed to increase socket's transmission queue:
          %s.", getSocketError());
    }else{
      ortp_message("Setting DCCP queue length to: %i.", len);
      done=TRUE;
    }
  }
#endif
  return done;
}

static ortp_socket_t dccp_accept(int fd){
  ortp_socket_t nsock;
#ifdef ORTP_DCCP
#ifdef ORTP_INET6
  struct sockaddr_storage remaddr;
#else
  struct sockaddr remaddr;
#endif
  socklen_t addrlen;
  int err;

  do{
    addrlen=sizeof(remaddr);
    err=accept(fd, (struct sockaddr*)&remaddr, &addrlen);
    if(err<0 ){
      ortp_warning ("Error accepting on Socket: %s.",
          getSocketError());
      return err;
    }
  }while(err<0);
  nsock=err;
#endif /*ORTP_DCCP*/
  return nsock;
```

```
}

static ortp_socket_t create_dccp_send_socket(const char *addr,
   int port, int *sock_family, bool_t reuse_addr, int ccid){
  ortp_socket_t sock=-1;
#ifdef  ORTP_DCCP
  int err;
  int optval = 1;
  uint8_t ccidval;


#ifdef ORTP_INET6
  char num[8];
  struct addrinfo hints, *res0, *res;
#else
  struct sockaddr_in saddr;
#endif


#ifdef ORTP_INET6

  memset(&hints, 0, sizeof(hints));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_DCCP;
  snprintf(num, sizeof(num), "%d",port);
  err = getaddrinfo(addr,num, &hints, &res0);
  if (err!=0) {
    ortp_warning ("Error in getaddrinfo on (addr=%s port=%i): %s"
       , addr, port, gai_strerror(err));
    return -1;
  }

  for (res = res0; res; res = res->ai_next) {
    sock = socket(res->ai_family, res->ai_socktype, IPPROTO_DCCP)
       ;
    *sock_family=res->ai_family;
    if (sock==-1)
      continue;

    if (reuse_addr){
      err = setsockopt (sock, SOL_SOCKET, SO_REUSEADDR,
          (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
      if (err < 0)
      {
        ortp_warning ("Fail to set rtp address reusable: %s.",
           getSocketError());
      }
    }
#if defined(ORTP_TIMESTAMP)
    err = setsockopt (sock, SOL_SOCKET, SO_TIMESTAMP,
      (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
    if (err < 0)
    {
      ortp_warning ("Fail to set rtp timestamp: %s.",
         getSocketError());
    }
#endif

    *sock_family=res->ai_family;
    break;
  }
  freeaddrinfo(res0);
#else
  saddr.sin_family = AF_INET;
  *sock_family=AF_INET;
  err = inet_aton (addr, &saddr.sin_addr);
  if (err < 0)
  {
    ortp_warning ("Error in socket address:%s.", getSocketError()
       );
    return -1;
  }
  saddr.sin_port = htons (port);

  sock = socket (PF_INET, SOCK_DCCP, IPPROTO_DCCP);
  if (sock==-1) return -1;

  if (reuse_addr){
    err = setsockopt (sock, SOL_SOCKET, SO_REUSEADDR,
        (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
    if (err < 0)
    {
      ortp_warning ("Fail to set rtp address reusable: %s.",
         getSocketError());
    }
  }
#if defined(ORTP_TIMESTAMP)
  err = setsockopt (sock, SOL_SOCKET, SO_TIMESTAMP,
      (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
  if (err < 0)
  {
    ortp_warning ("Fail to set rtp timestamp: %s.",getSocketError
       ());
  }
#endif
#endif
#if defined(WIN32) || defined(_WIN32_WCE)
  if (ortp_WSARecvMsg == NULL) {
    GUID guid = WSAID_WSARECVMSG;
    DWORD bytes_returned;
    if (WSAIoctl(sock, SIO_GET_EXTENSION_FUNCTION_POINTER, &guid,
        sizeof(guid),
      &ortp_WSARecvMsg, sizeof(ortp_WSARecvMsg), &bytes_returned,
          NULL, NULL) == SOCKET_ERROR) {
      ortp_warning("WSARecvMsg function not found.");
    }
  }
#endif
  optval=htonl(DCCP_SERVICE_CODE);
  err = setsockopt (sock, SOL_DCCP, DCCP_SOCKOPT_SERVICE, (
     SOCKET_OPTION_VALUE)&optval, sizeof (optval));
  if (err < 0)
  {
    ortp_warning ("Fail to set DCCP service code: %s.",
       getSocketError());
  }
  ccidval=ccid;
  err = setsockopt (sock, SOL_DCCP, DCCP_SOCKOPT_CCID, (
     SOCKET_OPTION_VALUE)&ccidval, sizeof (ccidval));
  if (err < 0)
  {
    ortp_warning ("Fail to set DCCP CCID: %s.",getSocketError());
  }
#endif /*ORTP_DCCP*/
  return sock;
}

static ortp_socket_t create_dccp_accept_socket(const char *addr,
   int port, int *sock_family, bool_t reuse_addr,int ccid){
  ortp_socket_t sock=-1;
#ifdef ORTP_DCCP
  int err;
  int optval = 1;
  uint8_t ccidval;
```

```
#ifdef ORTP_INET6
  char num[8];
  struct addrinfo hints, *res0, *res;
#else
  struct sockaddr_in saddr;
#endif


#ifdef ORTP_INET6

  memset(&hints, 0, sizeof(hints));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_DCCP;
  snprintf(num, sizeof(num), "%d",port);
  err = getaddrinfo(addr,num, &hints, &res0);
  if (err!=0) {
    ortp_warning ("Error in getaddrinfo on (addr=%s port=%i): %s"
       , addr, port, gai_strerror(err));
    return -1;
  }

  for (res = res0; res; res = res->ai_next) {
    sock = socket(res->ai_family, res->ai_socktype, IPPROTO_DCCP)
       ;
    *sock_family=res->ai_family;
    if (sock==-1)
      continue;

    if (reuse_addr){
      err = setsockopt (sock, SOL_SOCKET, SO_REUSEADDR,
          (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
      if (err < 0)
      {
        ortp_warning ("Fail to set rtp address reusable: %s.",
           getSocketError());
      }
    }
#if defined(ORTP_TIMESTAMP)
    err = setsockopt (sock, SOL_SOCKET, SO_TIMESTAMP,
      (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
    if (err < 0)
    {
      ortp_warning ("Fail to set rtp timestamp: %s.",
         getSocketError());
    }
#endif

    err = bind(sock, res->ai_addr, res->ai_addrlen);
    if (err != 0){
      ortp_warning ("Fail to bind rtp socket to (addr=%s port=%i)
          : %s.", addr,port, getSocketError());
      close_socket (sock);
      sock=-1;
      continue;
    }

    *sock_family=res->ai_family;
    break;
  }
  freeaddrinfo(res0);
#else
  saddr.sin_family = AF_INET;
  *sock_family=AF_INET;
  err = inet_aton (addr, &saddr.sin_addr);
  if (err < 0)
  {
    ortp_warning ("Error in socket address:%s.", getSocketError()
       );
    return -1;
  }
  saddr.sin_port = htons (port);

  sock = socket (PF_INET, SOCK_DCCP, IPPROTO_DCCP);
  if (sock==-1) return -1;

  if (reuse_addr){
    err = setsockopt (sock, SOL_SOCKET, SO_REUSEADDR,
        (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
    if (err < 0)
    {
      ortp_warning ("Fail to set rtp address reusable: %s.",
         getSocketError());
    }
  }

  err = bind (sock,(struct sockaddr *) &saddr, sizeof (saddr));
  if (err != 0)
  {
    ortp_warning ("Fail to bind rtp socket to port %i: %s.", port
        , getSocketError());
    close_socket (sock);
    return -1;
  }

#if defined(ORTP_TIMESTAMP)
  err = setsockopt (sock, SOL_SOCKET, SO_TIMESTAMP,
      (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
  if (err < 0)
  {
    ortp_warning ("Fail to set rtp timestamp: %s.",getSocketError
        ());
  }
#endif
#endif
#if defined(WIN32) || defined(_WIN32_WCE)
  if (ortp_WSARecvMsg == NULL) {
    GUID guid = WSAID_WSARECVMSG;
    DWORD bytes_returned;
    if (WSAIoctl(sock, SIO_GET_EXTENSION_FUNCTION_POINTER, &guid,
        sizeof(guid),
      &ortp_WSARecvMsg, sizeof(ortp_WSARecvMsg), &bytes_returned,
          NULL, NULL) == SOCKET_ERROR) {
      ortp_warning("WSARecvMsg function not found.");
    }
  }
#endif

  optval=htonl(DCCP_SERVICE_CODE);
  err = setsockopt (sock, SOL_DCCP, DCCP_SOCKOPT_SERVICE, (
     SOCKET_OPTION_VALUE)&optval, sizeof (optval));
  if (err < 0)
  {
    ortp_warning ("Fail to set DCCP service code: %s.",
       getSocketError());
  }
  ccidval=ccid;
  err = setsockopt (sock, SOL_DCCP, DCCP_SOCKOPT_CCID, (
     SOCKET_OPTION_VALUE)&ccidval, sizeof (ccidval));
  if (err < 0)
  {
    ortp_warning ("Fail to set DCCP CCID: %s.",getSocketError());
  }
  err=listen(sock,5);
  if(err<0){
    ortp_warning ("Error listening on Socket:%s.", getSocketError
       ());
    close_socket (sock);
    return -1;
  }
  set_non_blocking_socket(sock);
#endif /*ORTP_DCCP*/
  return sock;
}

static ortp_socket_t create_and_bind_dccp_random(const char *
   localip, int *sock_family, int *port,bool_t send, int ccid){
  int retry;
  ortp_socket_t sock = -1;
  for (retry=0;retry<100;retry++)
  {
    int localport;
    do
    {
      localport = (rand () + 5000) & 0xfffe;
    }
    while ((localport < 5000) || (localport > 0xffff));
    /*do not set REUSEADDR in case of random allocation */
    if(send){
      sock = create_dccp_send_socket(localip, localport,
         sock_family,FALSE, ccid);
    }else{
      sock = create_dccp_accept_socket(localip, localport,
         sock_family,FALSE, ccid);
    }
    if (sock!=-1) {
      *port=localport;
      return sock;
    }
  }
  ortp_warning("create_and_bind_random: Could not find a random
     port for %s !",localip);
  return -1;
}

/**
 *rtp_session_set_local_addr:
```

```
 *@session:   a rtp session freshly created.
 *@addr:    a local IP address in the xxx.xxx.xxx.xxx form.
 *@rtp_port:    a local port or -1 to let oRTP choose the port
    randomly
 *@rtcp_port:   a local port or -1 to let oRTP choose the port
    randomly
 *
 *  Specify the local addr to be use to listen for rtp packets or
     to send rtp packet from.
 *  In case where the rtp session is send-only, then it is not
    required to call this function:
 *  when calling rtp_session_set_remote_addr(), if no local
    address has been set, then the
 *  default INADRR_ANY (0.0.0.0) IP address with a random port
    will be used. Calling
 *  rtp_sesession_set_local_addr() is mandatory when the session
    is recv-only or duplex.
 *
 *  Returns: 0 on success.
**/

int
rtp_session_set_local_addr (RtpSession * session, const char *
   addr, int rtp_port, int rtcp_port)
{
  ortp_socket_t sock1;
  ortp_socket_t sock2;
  int sockfamily=0;
  int port;

  if (session->rtp.s_socket!=(ortp_socket_t)-1 || session->rtp.
     a_socket!=(ortp_socket_t)-1){
    /* don't rebind, but close before*/
    rtp_session_release_sockets(session);
  }
  /* try to bind the rtp port */
  if (session->rtp.is_dccp){
    if (rtp_port>0){
      sock1=create_dccp_accept_socket(addr,rtp_port,&sockfamily,
         session->reuseaddr,session->rtp.dccp_ccid);
    }else{
      sock1=create_and_bind_dccp_random(addr,&sockfamily,&
         rtp_port,FALSE,session->rtp.dccp_ccid);
    }
    sock2=create_and_bind_dccp_random(addr,&sockfamily,&port,TRUE
       ,session->rtp.dccp_ccid);
    set_dccp_q_len(sock2, session->rtp.dccp_q_len);
  }else{
    if (rtp_port>0)
      sock1=create_udp_socket(addr,rtp_port,&sockfamily,session->
         reuseaddr);
    else
      sock1=create_and_bind_udp_random(addr,&sockfamily,&rtp_port
         );
    sock2=sock1;
  }

  if (sock1!=-1 && sock2!=-1){

    if(session->rtp.is_dccp){
      set_socket_sizes(sock1,session->rtp.snd_socket_size,session
         ->rtp.rcv_socket_size);
      set_socket_sizes(sock2,session->rtp.snd_socket_size,session
         ->rtp.rcv_socket_size);
      session->rtp.s_socket=sock2;
      session->rtp.a_socket=sock1;
    }else{
      set_socket_sizes(sock1,session->rtp.snd_socket_size,session
         ->rtp.rcv_socket_size);
      session->rtp.s_socket=sock1;
      session->rtp.r_socket=sock1;
      session->rtp.a_socket=sock1;
    }
    session->rtp.sockfamily=sockfamily;
    session->rtp.loc_port=rtp_port;
    /*try to bind rtcp port */
    if (rtcp_port<0) rtcp_port=rtp_port+1;
    sock1=create_udp_socket(addr,rtcp_port,&sockfamily,session->
       reuseaddr);
    if (sock1!=(ortp_socket_t)-1){
      session->rtcp.sockfamily=sockfamily;
      session->rtcp.socket=sock1;
    }else{
      ortp_warning("Could not create and bind rtcp socket.");
    }

    /* set socket options (but don't change chosen states) */
    rtp_session_set_dscp( session, -1 );
    if(!session->rtp.is_dccp){
      rtp_session_set_multicast_ttl( session, -1 );
      rtp_session_set_multicast_loopback( session, -1 );
    }

    return 0;
  }
  ortp_debug("Could not bind RTP socket on port to %s port %i",
     addr,rtp_port);
  return -1;
}

#define IP_UDP_OVERHEAD (20+8)
#define IP6_UDP_OVERHEAD (40+8)
#define IP_DCCP_OVERHEAD (20+30)
#define IP6_DCCP_OVERHEAD (40+30)

static void update_sent_bytes(RtpSession*s, int nbytes){
  int overhead;
#ifdef ORTP_INET6
  if(s->rtp.is_dccp){
    overhead=(s->rtp.sockfamily==AF_INET6) ? IP6_DCCP_OVERHEAD :
       IP_DCCP_OVERHEAD;
  }else{
    overhead=(s->rtp.sockfamily==AF_INET6) ? IP6_UDP_OVERHEAD :
       IP_UDP_OVERHEAD;
  }
#else
  if(s->rtp.is_dccp){
    overhead=IP_DCCP_OVERHEAD;
  }else{
    overhead=IP_UDP_OVERHEAD;
  }
#endif
  if (s->rtp.sent_bytes==0){
    gettimeofday(&s->rtp.send_bw_start,NULL);
  }
  s->rtp.sent_bytes+=nbytes+overhead;
}

static void update_recv_bytes(RtpSession*s, int nbytes){
  int overhead;
#ifdef ORTP_INET6
  if(s->rtp.is_dccp){
    overhead=(s->rtp.sockfamily==AF_INET6) ? IP6_DCCP_OVERHEAD :
       IP_DCCP_OVERHEAD;
  }else{
    overhead=(s->rtp.sockfamily==AF_INET6) ? IP6_UDP_OVERHEAD :
       IP_UDP_OVERHEAD;
  }
#else
  if(s->rtp.is_dccp){
    overhead=IP_DCCP_OVERHEAD;
  }else{
    overhead=IP_UDP_OVERHEAD;
  }
#endif
  if (s->rtp.recv_bytes==0){
    gettimeofday(&s->rtp.recv_bw_start,NULL);
  }
  s->rtp.recv_bytes+=nbytes+overhead;
}

int
rtp_session_rtp_send (RtpSession * session, mblk_t * m)
{
  int error;
  int i;
  rtp_header_t *hdr;
  struct sockaddr *destaddr=(struct sockaddr*)&session->rtp.
     rem_addr;
  socklen_t destlen=session->rtp.rem_addrlen;
  ortp_socket_t sockfd=session->rtp.s_socket;

  hdr = (rtp_header_t *) m->b_rptr;
  /* perform host to network conversions */
  hdr->ssrc = htonl (hdr->ssrc);
  hdr->timestamp = htonl (hdr->timestamp);
  hdr->seq_number = htons (hdr->seq_number);
  for (i = 0; i < hdr->cc; i++)
    hdr->csrc[i] = htonl (hdr->csrc[i]);

  if(session->rtp.is_dccp && session->rtp.s_connected==FALSE){
    if (connect(sockfd,destaddr,destlen)<0){
      if(errno!=EISCONN){
        ortp_warning("Could not connect() socket: %s",
           getSocketError());
        return -1;
      }
    }
    set_non_blocking_socket(sockfd);
    session->rtp.s_connected=TRUE;
    destaddr=NULL;
    destlen=0;
  }

  if (session->flags & RTP_SOCKET_CONNECTED) {
    destaddr=NULL;
    destlen=0;
  }

  if (rtp_session_using_transport(session, rtp)){
    error = (session->rtp.tr->t_sendto) (session->rtp.tr,m,0,
       destaddr,destlen);
  }else{
#ifdef USE_SENDMSG
    error=rtp_sendmsg(sockfd,m,destaddr,destlen);
#else
    if (m->b_cont!=NULL)
      msgpullup(m,-1);
    error = sendto (sockfd, (char*)m->b_rptr, (int) (m->b_wptr -
       m->b_rptr),
       0,destaddr,destlen);
#endif
  }
  if (error < 0){
    if (session->on_network_error.count>0){
      rtp_signal_table_emit3(&session->on_network_error,(long)"
         Error sending RTP packet",INT_TO_POINTER(
         getSocketErrorCode()));
    }else ortp_warning ("Error sending rtp packet: %s ; socket=%i
       ", getSocketError(), sockfd);
    session->rtp.send_errno=getSocketErrorCode();
    if(errno!=EINTR && errno!=EAGAIN && errno!=EWOULDBLOCK){
        session->rtp.s_connected=FALSE;
    }
    if(session->rtp.is_dccp && (errno==EAGAIN || errno==
       EWOULDBLOCK) && session->eventqs!=NULL){
        session->rtp.stats.rejected++;
        OrtpEvent *ev;
        OrtpEventData *evd;
        struct timeval tm;
        long msec;
        gettimeofday(&tm,NULL);
        msec=(tm.tv_sec- session->rtp.last_reject.tv_sec)*1000;
        msec+=(tm.tv_usec- session->rtp.last_reject.tv_usec)
           /1000;
        if( (session->rtp.dccp_ccid == 3 &&  msec >
           CCID3_REJECT_DELAY_MSEC) ||
            (session->rtp.dccp_ccid == 2 && msec >
              CCID2_REJECT_DELAY_MSEC) ){
          gettimeofday(&session->rtp.last_reject,NULL);
          ev=ortp_event_new(ORTP_EVENT_SEND_REJECTED);
          evd=ortp_event_get_data(ev);
          evd->packet=dupmsg(m);
          rtp_session_dispatch_event(session,ev);
        }
        struct timeval now;
        gettimeofday(&now,NULL);
        struct tm *timeinfo;
        char buffer[80];
        timeinfo=localtime(&now.tv_sec);
        strftime(buffer,80,"%Y-%m-%d_%H:%M:%S",timeinfo);
        ortp_message("sjero-info: Rejected Send. time=%s.%06d
           session=%p",buffer, (int)now.tv_usec, session);

    }
  }else{
    update_sent_bytes(session,error);
#ifdef ORTP_DCCP
    if(session->rtp.is_dccp && session->rtp.dccp_ccid==3){
      struct timeval tm;
      long msec;
      gettimeofday(&tm,NULL);
      msec=(tm.tv_sec- session->rtp.last_bw_update.tv_sec)*1000;
      msec+=(tm.tv_usec- session->rtp.last_bw_update.tv_usec)
         /1000;
      if(msec >= BW_SAMPLE_PERIOD_MSEC){
        gettimeofday(&session->rtp.last_bw_update,NULL);
        struct tfrc_tx_info ifo;
        socklen_t len=sizeof(ifo);
        if(getsockopt(session->rtp.s_socket,SOL_DCCP,
           DCCP_SOCKOPT_CCID_TX_INFO,&ifo,&len)>=0){
          OrtpEvent *ev;
          OrtpEventData *evd;
          ev=ortp_event_new(ORTP_EVENT_BANDWIDTH);
          evd=ortp_event_get_data(ev);
          evd->info.bandwidth=(ifo.tfrctx_x>>6)*8;
          rtp_session_dispatch_event(session,ev);
        }
      }
    }
#endif
  }
  freemsg (m);
  return error;
}

int rtp_session_rtp_recv (RtpSession * session, uint32_t user_ts)
{
  int error=0;
  ortp_socket_t sockfd=session->rtp.r_socket;
  ortp_socket_t new;
#ifdef ORTP_INET6
  struct sockaddr_storage remaddr;
#else
  struct sockaddr remaddr;
#endif
  socklen_t addrlen = sizeof (remaddr);
  mblk_t *mp;

  if ((session->rtp.a_socket==(ortp_socket_t)-1) && !
     rtp_session_using_transport(session, rtp)) return -1;  /*
     session has no sockets for the moment*/

  while (1)
  {
    bool_t sock_connected=!!(session->flags &
       RTP_SOCKET_CONNECTED);

    if (session->rtp.cached_mp==NULL)
       session->rtp.cached_mp = msgb_allocator_alloc(&session->
          allocator,session->recv_buf_size);
    mp=session->rtp.cached_mp;

    if(session->rtp.is_dccp){
      if (session->rtp.r_connected==FALSE){
        if((new=dccp_accept(session->rtp.a_socket))<0){
          return -1;
        }
        sockfd=session->rtp.r_socket=new;
        set_non_blocking_socket(sockfd);
        session->rtp.r_connected=TRUE;
      }else{
        if (rtp_session_using_transport(session, rtp)) {
          error = (session->rtp.tr->t_recvfrom)(session->rtp.tr,
             mp, 0,
              (struct sockaddr *) &remaddr,
              &addrlen);
        } else { error = rtp_session_rtp_recv_abstract(sockfd, mp
           , 0,
              NULL,NULL);
        }
      }
    }else{
      if (sock_connected){
        error=rtp_session_rtp_recv_abstract(sockfd, mp, 0, NULL,
           NULL);
      }else if (rtp_session_using_transport(session, rtp)) {
        error = (session->rtp.tr->t_recvfrom)(session->rtp.tr, mp
           , 0,
            (struct sockaddr *) &remaddr,
            &addrlen);
      } else { error = rtp_session_rtp_recv_abstract(sockfd, mp,
         0,
            (struct sockaddr *) &remaddr,
            &addrlen);
      }
    }
    if (error > 0){
      if (session->rtp.is_dccp==FALSE && session->use_connect){
        /* In the case where use_connect is false, symmetric RTP
           is handled in rtp_session_rtp_parse() */
        if (session->symmetric_rtp && !sock_connected){
          /* store the sender rtp address to do symmetric RTP */
          memcpy(&session->rtp.rem_addr,&remaddr,addrlen);
          session->rtp.rem_addrlen=addrlen;
          if (try_connect(sockfd,(struct sockaddr*)&remaddr,
             addrlen))
            session->flags|=RTP_SOCKET_CONNECTED;
        }
      }
      mp->b_wptr+=error;
      if (session->net_sim_ctx)
        mp=rtp_session_network_simulate(session,mp);
      /* then parse the message and put on jitter buffer queue */
      if (mp){
        update_recv_bytes(session,mp->b_wptr-mp->b_rptr);
        rtp_session_rtp_parse(session, mp, user_ts, (struct
           sockaddr*)&remaddr,addrlen);
      }
      session->rtp.cached_mp=NULL;
      /*for bandwidth measurements:*/
    }
    else
    {
      int errnum;
      if (error==-1 && !is_would_block_error((errnum=
         getSocketErrorCode())) )
      {
        if (session->on_network_error.count>0){
          rtp_signal_table_emit3(&session->on_network_error,(long
             )"Error receiving RTP packet",INT_TO_POINTER(
             getSocketErrorCode()));
        }else ortp_warning("Error receiving RTP packet: %s, err
           num  [%i],error [%i]",getSocketError(),errnum,error);
#ifdef __ios
        /*hack for iOS and non-working socket because of
           background mode*/
        if (errnum==ENOTCONN){
          /*re-create new sockets */
          rtp_session_set_local_addr(session,session->rtp.
             sockfamily==AF_INET ? "0.0.0.0" : "::0",session->rtp
             .loc_port,session->rtcp.loc_port);
        }
#endif

        if(session->rtp.is_dccp){
          close_socket(session->rtp.r_socket);
          session->rtp.r_connected=FALSE;
        }
      }else{
        /*EWOULDBLOCK errors or transports returning 0 are
           ignored.*/
        if (session->net_sim_ctx){
          /*drain possible packets queued in the network
             simulator*/
          mp=rtp_session_network_simulate(session,NULL);
          if (mp){
            /* then parse the message and put on jitter buffer
               queue */
            update_recv_bytes(session,msgdsize(mp));
            rtp_session_rtp_parse(session, mp, user_ts, (struct
               sockaddr*)&session->rtp.rem_addr,session->rtp.
               rem_addrlen);
          }
        }
      }
      /* don't free the cached_mp, it will be reused next time */
      return -1;
    }
  }
  return error;
}
```

A.2 Bitrate Control

Full DCCP support also required modifying Linphone's bitrate control code to handle feedback from DCCP. We present our modifications below.

Linphone's bitrate control code is divided into three files. qosanalyzer.c takes feedback information and determines if the connection is still in an acceptable state. bitratecontrol.c contains the bitrate control state machine, which takes the recommendations from qosanalyzer.c and makes a decision about what to do. It also handles gradually increasing the bitrate under good network condtions. Finally, bitratedriver.c handles the actual interaction with the video codec to achieve the requested bitrate change.

The relevant sections of code appear below. This code is based off of mediastreamer2 revision 22f54d4038fd4ba2897e506be776fe3c0956dd3d from git://git.linphone.org/linphone.git pulled on 2012-12-28.

mediastreamer2/src/qosanalyzer.c

```
#define STATS_HISTORY 3

static const float unacceptable_loss_rate=10;
static const int big_jitter=10; /*ms */
static const float significant_delay=0.2; /*seconds*/

typedef struct rtpstats{
  uint64_t high_seq_recv; /*highest sequence number received*/
  float lost_percentage; /*percentage of lost packet since last
     report*/
  float int_jitter; /*interrarrival jitter */
  float rt_prop; /*round trip propagation*/
  bool_t send_refused;
}rtpstats_t;

typedef struct _MSSimpleQosAnalyser{
  MSQosAnalyser parent;
  RtpSession *session;
  int clockrate;
  rtpstats_t stats[STATS_HISTORY];
  int curindex;
  bool_t rt_prop_doubled;
  bool_t send_was_refused;
  bool_t pad[2];
}MSSimpleQosAnalyser;

static bool_t rt_prop_doubled(rtpstats_t *cur,rtpstats_t *prev){
  //ms_message("AudioBitrateController: cur=%f, prev=%f",cur->
     rt_prop,prev->rt_prop);
  if (cur->rt_prop>=significant_delay && prev->rt_prop>0){
    if (cur->rt_prop>=(prev->rt_prop*2.0)){
      /*propagation doubled since last report */
      return TRUE;
    }
  }
  return FALSE;
}

static bool_t rt_prop_increased(MSSimpleQosAnalyser *obj){
  rtpstats_t *cur=&obj->stats[obj->curindex % STATS_HISTORY];
  rtpstats_t *prev=&obj->stats[(STATS_HISTORY+obj->curindex-1) %
     STATS_HISTORY];

  if (rt_prop_doubled(cur,prev)){
    obj->rt_prop_doubled=TRUE;
    return TRUE;
  }
  return FALSE;
}

static bool_t simple_analyser_process_rejected_send(MSQosAnalyser
    *objbase){
  MSSimpleQosAnalyser *obj=(MSSimpleQosAnalyser*)objbase;
  rtpstats_t *cur;

  cur=&obj->stats[obj->curindex % STATS_HISTORY];
  cur->send_refused=TRUE;
  obj->send_was_refused=TRUE;
  return TRUE;
}
```

```
static bool_t simple_analyser_process_rtcp(MSQosAnalyser *objbase
   , mblk_t *rtcp){
  MSSimpleQosAnalyser *obj=(MSSimpleQosAnalyser*)objbase;
  rtpstats_t *cur;
  const report_block_t *rb=NULL;

  if(obj->session->rtp.is_dccp){
    return TRUE;
  }

  if (rtcp_is_SR(rtcp)){
    rb=rtcp_SR_get_report_block(rtcp,0);
  }else if (rtcp_is_RR(rtcp)){
    rb=rtcp_RR_get_report_block(rtcp,0);
  }
  if (rb && report_block_get_ssrc(rb)==rtp_session_get_send_ssrc(
     obj->session)){

    obj->curindex++;
    cur=&obj->stats[obj->curindex % STATS_HISTORY];

    if (obj->clockrate==0){
      PayloadType *pt=rtp_profile_get_payload(
         rtp_session_get_send_profile(obj->session),
         rtp_session_get_send_payload_type(obj->session));
      if (pt!=NULL) obj->clockrate=pt->clock_rate;
      else return FALSE;
    }

    cur->high_seq_recv=report_block_get_high_ext_seq(rb);
    cur->lost_percentage=100.0*(float)
       report_block_get_fraction_lost(rb)/256.0;
    cur->int_jitter=1000.0*(float)
       report_block_get_interarrival_jitter(rb)/(float)obj->
       clockrate;
    cur->rt_prop=rtp_session_get_round_trip_propagation(obj->
       session);
    cur->send_refused=FALSE;
    ms_message("MSQosAnalyser: lost_percentage=%f, int_jitter=%f
       ms, rt_prop=%f sec",cur->lost_percentage,cur->int_jitter,
       cur->rt_prop);
  }
  return rb!=NULL;
}
```

```
static void simple_analyser_suggest_action(MSQosAnalyser *objbase
    , MSRateControlAction *action){
  MSSimpleQosAnalyser *obj=(MSSimpleQosAnalyser*)objbase;
  rtpstats_t *cur=&obj->stats[obj->curindex % STATS_HISTORY];
  /*big losses and big jitter */
  if (cur->lost_percentage>=unacceptable_loss_rate && cur->
     int_jitter>=big_jitter){
    action->type=MSRateControlActionDecreaseBitrate;
    action->value=MIN(cur->lost_percentage,50);
    ms_message("MSQosAnalyser: loss rate unacceptable and big
       jitter");
  }else if (rt_prop_increased(obj)){
    action->type=MSRateControlActionDecreaseBitrate;
    action->value=20;
    ms_message("MSQosAnalyser: rt_prop doubled.");
  }else if (cur->send_refused){
    action->type=MSRateControlActionDecreaseBitrate;
    action->value=30;
    ms_message("MSQosAnalyser: send refused, cut bitrate");
    cur->send_refused=FALSE;
  }else if (cur->lost_percentage>=unacceptable_loss_rate){
    /*big loss rate but no jitter, and no big rtp_prop: pure
       lossy network*/
    action->type=MSRateControlActionDecreasePacketRate;
    action->value=20; /*For video, this is equivalent to
       Decreasing bitrate*/
    ms_message("MSQosAnalyser: loss rate unacceptable.");
  }else{
    action->type=MSRateControlActionDoNothing;
    ms_message("MSQosAnalyser: everything is fine.");
  }
}

static bool_t simple_analyser_has_improved(MSQosAnalyser *objbase
    ){
  MSSimpleQosAnalyser *obj=(MSSimpleQosAnalyser*)objbase;
  rtpstats_t *cur=&obj->stats[obj->curindex % STATS_HISTORY];
  rtpstats_t *prev=&obj->stats[(STATS_HISTORY+obj->curindex-1) %
     STATS_HISTORY];

  if(cur->send_refused==TRUE){
    goto end;
  }
  if (prev->lost_percentage>=unacceptable_loss_rate){
    if (cur->lost_percentage<prev->lost_percentage){
      ms_message("MSQosAnalyser: lost percentage has improved");
      return TRUE;
    }else goto end;
  }
  if (obj->rt_prop_doubled){
    if(cur->rt_prop<prev->rt_prop){
      ms_message("MSQosAnalyser: rt prop decrased");
      obj->rt_prop_doubled=FALSE;
      return TRUE;
    }else goto end;
  }

  if (obj->send_was_refused){
    if(cur->send_refused==FALSE){
      obj->send_was_refused=FALSE;
      return TRUE;
    }else goto end;
  }

end:
  ms_message("MSQosAnalyser: no improvements.");
  return FALSE;
}
```

mediastreamer2/src/bitratecontrol.c

```
#define CCID2_ADJUSTMENT_WAIT_MSEC 3000
#define CCID3_ADJUSTMENT_WAIT_MSEC 1000
static const int probing_up_interval=10;

struct _MSBitrateController{
  MSQosAnalyser *analyser;
  MSBitrateDriver *driver;
  enum state_t state;
  int stable_count;
  int probing_up_count;
  struct timeval last_change;
  int adjust_wait;
};

MSBitrateController *ms_bitrate_controller_new(MSQosAnalyser *
   qosanalyser, MSBitrateDriver *driver, RtpSession *session){
  MSBitrateController *obj=ms_new0(MSBitrateController,1);
  obj->analyser=ms_qos_analyser_ref(qosanalyser);
  obj->driver=ms_bitrate_driver_ref(driver);
  obj->last_change.tv_sec=0;
  obj->last_change.tv_usec=0;
  if(session->rtp.is_dccp){
    if(session->rtp.dccp_ccid == 3){
      obj->adjust_wait=CCID3_ADJUSTMENT_WAIT_MSEC;
    }else if(session->rtp.dccp_ccid == 2){
      obj->adjust_wait=CCID2_ADJUSTMENT_WAIT_MSEC;
    }
  }else{
    obj->adjust_wait=0;
  }
  return obj;
}

static void state_machine(MSBitrateController *obj){
  MSRateControlAction action;
  switch(obj->state){
    case Stable:
      obj->stable_count++;
    case Init:
      ms_qos_analyser_suggest_action(obj->analyser,&action);
      if (action.type!=MSRateControlActionDoNothing){
        execute_action(obj,&action);
        obj->state=Probing;
      }else if (obj->stable_count>=probing_up_interval){
        action.type=MSRateControlActionIncreaseQuality;
        action.value=10;
        execute_action(obj,&action);
        obj->state=ProbingUp;
        obj->probing_up_count=0;
      }
    break;
    case Probing:
      obj->stable_count=0;
      if (ms_qos_analyser_has_improved(obj->analyser)){
        obj->state=Stable;
      }else{
        ms_qos_analyser_suggest_action(obj->analyser,&action);
        if (action.type!=MSRateControlActionDoNothing){
          execute_action(obj,&action);
        }
      }
    break;
    case ProbingUp:
      obj->stable_count=0;
      obj->probing_up_count++;
      ms_qos_analyser_suggest_action(obj->analyser,&action);
      if (action.type!=MSRateControlActionDoNothing){
        execute_action(obj,&action);
        obj->state=Probing;
      }else{
        /*continue with slow ramp up*/
        if (obj->probing_up_count==2){
          action.type=MSRateControlActionIncreaseQuality;
          action.value=10;
          if (execute_action(obj,&action)==-1){
            /* we reached the maximum*/
            obj->state=Init;
          }
          obj->probing_up_count=0;
        }
      }
    break;
    default:
    break;
  }
  ms_message("MSBitrateController: current state is %s",
     state_name(obj->state));
}



void ms_bitrate_controller_process_rtcp(MSBitrateController *obj,
    mblk_t *rtcp){
  struct timeval tm;
  long msec;
  gettimeofday(&tm,NULL);
  msec=(tm.tv_sec- obj->last_change.tv_sec)*1000;
  msec+=(tm.tv_usec- obj->last_change.tv_usec)/1000;
  if(msec <= obj->adjust_wait){
    return;
  }
  if (ms_qos_analyser_process_rtcp(obj->analyser,rtcp)){
    state_machine(obj);
  }
}

void ms_bitrate_controller_process_rejected_send(
   MSBitrateController *obj){
  struct timeval tm;
  long msec;
  gettimeofday(&tm,NULL);
  msec=(tm.tv_sec- obj->last_change.tv_sec)*1000;
  msec+=(tm.tv_usec- obj->last_change.tv_usec)/1000;
  if(msec <= obj->adjust_wait){
    return;
  }
  if(ms_qos_analyser_process_rejected_send(obj->analyser)){
    state_machine(obj);
  }
}

void ms_bitrate_controler_process_bandwidth_update(
   MSBitrateController *obj, int value){
  MSRateControlAction action;
  struct timeval tm;
  long msec;
  gettimeofday(&tm,NULL);
  msec=(tm.tv_sec- obj->last_change.tv_sec)*1000;
  msec+=(tm.tv_usec- obj->last_change.tv_usec)/1000;
  if(msec <= obj->adjust_wait){
    return;
  }

  obj->state=Stable;
  obj->stable_count=0;
  action.type=MSRateControlSetBitrate;
  action.value=value;
  execute_action(obj,&action);
}
```

mediastreamer2/src/bitratedriver.c

```
static const int min_video_bitrate=64000;
static const float increase_ramp=1.1;

typedef struct _MSAVBitrateDriver{
  MSBitrateDriver parent;
  MSBitrateDriver *audio_driver;
  MSFilter *venc;
  int nom_bitrate;
  int bitrate_limit;
  int cur_bitrate;
  int set_hold;
}MSAVBitrateDriver;
```

```
static int dec_video_bitrate(MSAVBitrateDriver *obj, const
    MSRateControlAction *action){
  int new_br;

  ms_filter_call_method(obj->venc,MS_FILTER_GET_BITRATE,&obj->
      cur_bitrate);

  if(obj->nom_bitrate > obj->cur_bitrate){
    obj->nom_bitrate=obj->cur_bitrate;
  }
  if(obj->nom_bitrate*10 < obj->cur_bitrate){
    obj->nom_bitrate=obj->cur_bitrate/2;
  }
  new_br=((float)obj->nom_bitrate)*(100.0-(float)action->value)
      /100.0;
  if (new_br<min_video_bitrate){
    ms_message("MSAVBitrateDriver: reaching low bound.");
    new_br=min_video_bitrate;
  }
  obj->nom_bitrate=obj->cur_bitrate=new_br;
  ms_message("MSAVBitrateDriver: targeting %i bps for video
      encoder.",new_br);
  ms_filter_call_method(obj->venc,MS_FILTER_SET_BITRATE,&new_br);
  return new_br==min_video_bitrate ? -1 : 0;
}

static int inc_video_bitrate(MSAVBitrateDriver *obj, const
    MSRateControlAction *action){
  int newbr;
  int ret=0;

  ms_filter_call_method(obj->venc,MS_FILTER_GET_BITRATE,&obj->
      cur_bitrate);
  if(obj->nom_bitrate*10 < obj->cur_bitrate){
    obj->nom_bitrate=obj->cur_bitrate/2;
  }
  if(obj->nom_bitrate > obj->cur_bitrate*10){
    obj->nom_bitrate=obj->cur_bitrate*2;
  }
  newbr=(float)obj->nom_bitrate*(1.0+((float)action->value/100.0)
      );
  if(newbr<min_video_bitrate){
    ret=-1;
    newbr=min_video_bitrate;
  }
  if(newbr > obj->bitrate_limit){
    newbr=obj->bitrate_limit;
    ret=-1;
  }
  obj->nom_bitrate=obj->cur_bitrate=newbr;
  ms_message("MSAVBitrateDriver: increasing bitrate to %i bps for
       video encoder.",obj->cur_bitrate);
  ms_filter_call_method(obj->venc,MS_FILTER_SET_BITRATE,&obj->
      cur_bitrate);
  return ret;
}

static int set_video_bitrate(MSAVBitrateDriver *obj, const
    MSRateControlAction *action){
  int ret=0;
  int proposed_bitrate;

  proposed_bitrate=action->value*0.9; //leave 10% for
      fluctuations
  obj->set_hold++;
  if(proposed_bitrate < min_video_bitrate){
    proposed_bitrate=min_video_bitrate;
  }
  if(proposed_bitrate > obj->bitrate_limit){
    proposed_bitrate=obj->bitrate_limit;
    ret=-1;
  }
  if(obj->set_hold > 4 &&
       ((obj->cur_bitrate - proposed_bitrate > obj->cur_bitrate
          *0.1) ||
        (proposed_bitrate - obj->cur_bitrate > obj->cur_bitrate
           *0.2))  ){
    obj->nom_bitrate=obj->cur_bitrate=proposed_bitrate;
    obj->set_hold=0;
    ms_message("MSAVBitrateDriver: setting bitrate to %i bps for
        video encoder.", obj->cur_bitrate);
    ms_filter_call_method(obj->venc,MS_FILTER_SET_BITRATE,&obj->
        cur_bitrate);
  }
  return ret;
}

static int av_driver_execute_action(MSBitrateDriver *objbase,
    const MSRateControlAction *action){
```

```
  MSAVBitrateDriver *obj=(MSAVBitrateDriver*)objbase;
  int ret=0;
  if (obj->bitrate_limit==0){
    ms_filter_call_method(obj->venc,MS_FILTER_GET_BITRATE,&obj->
        bitrate_limit);
    if (obj->bitrate_limit==0){
      ms_warning("MSAVBitrateDriver: Not doing adaptive rate
          control on video encoder, it does not seem to support
          that.");
      return -1;
    }
    obj->nom_bitrate=obj->bitrate_limit;
  }

  switch(action->type){
    case MSRateControlActionDecreaseBitrate:
      ret=dec_video_bitrate(obj,action);
    break;
    case MSRateControlActionDecreasePacketRate:
      if (obj->audio_driver){
        ret=ms_bitrate_driver_execute_action(obj->audio_driver,
            action);
      }
      ret=dec_video_bitrate(obj,action);
    break;
    case MSRateControlActionIncreaseQuality:
      ret=inc_video_bitrate(obj,action);
    break;
    case MSRateControlActionDoNothing:
    break;
    case MSRateControlSetBitrate:
      ret=set_video_bitrate(obj,action);
    break;

  }
  return ret;
}
```

Appendix B: DCCP CCID 3 Patch

In the process of conducting this research, we identified a bug in DCCP CCID 3's handling of loss intervals, which are used to compute the loss event rate.

The bug is that CCID 3 does not update the length of the second loss interval until the loss starting the third interval. According to the TFRC standard [FHPW08], it should be updating the length and recomputing the loss event rate after each packet in the loss interval. As a result, the loss event rate stays constant throughout the entire second loss interval. We have observed connections maintaining extremely low sending rates for many minutes as a result of this bug.

We submitted a patch for this issue in [Jer13] and had it accepted into the Linux DCCP testing tree. We expect this patch to make it into the mainline kernel eventually, but, as of this writing, we are uncertain when that will happen.

We also backported this patch to the 3.2.0-39 kernel used in our testing. That patch is included below. It applies cleanly to the mainline 3.2 kernel as well as the Ubuntu 3.2.0-39 kernel used in our tests.

DCCP CCID 3 Second Loss Interval Patch

```
--- a/net/dccp/ccids/lib/loss_interval.c  2012-04-05 14:03:32.000000000
    -0400
+++ b/net/dccp/ccids/lib/loss_interval.c  2013-04-16 12:27:44.712156108
    -0400
@@ -153,9 +153,20 @@
  new->li_ccval   = tfrc_rx_hist_loss_prev(rh)->tfrchrx_ccval;
  new->li_is_closed = 0;

- if (++lh->counter == 1)
+ if (++lh->counter == 1) {
    lh->i_mean = new->li_length = (*calc_first_li)(sk);
- else {
+   new->li_is_closed = 1;
+   new = tfrc_lh_demand_next(lh);
+   if (unlikely(new == NULL)) {
+     DCCP_CRIT("Cannot allocate/add loss record.");
+     return false;
+   }
+   ++lh->counter;
+   new->li_seqno   = tfrc_rx_hist_loss_prev(rh)->tfrchrx_seqno;
+   new->li_ccval   = tfrc_rx_hist_loss_prev(rh)->tfrchrx_ccval;
+   new->li_is_closed = 0;
+   new->li_length    = 1;
+ } else {
    cur->li_length = dccp_delta_seqno(cur->li_seqno, new->li_seqno);
    new->li_length = dccp_delta_seqno(new->li_seqno,
          tfrc_rx_hist_last_rcv(rh)->tfrchrx_seqno) + 1;
```