High Quality Video Conferencing

Final Report for COMP420Y

Thomas Young

Department of Computer Science

Hamilton, New Zealand

December 4, 2003
Abstract

During a video conference between two parties, the delay needs to be low between one side talking and the other side receiving it in order to make it an effective means of communication. Traditionally Video applications have sacrificed latency for smooth video quality, using traditional CODECS such as MPEG over network protocols such as UDP. However these methods can put strain on the network with the lack of congestion control and can cause significant latency due to the properties of MPEG.

This report deals with a solution using a scalable intra-frame video CODEC that adds very little latency. Streaming it on top of an experimental network protocol named Datagram Congestion Control Protocol (DCCP). DCCP can use a congestion control algorithm called TCP Friendly Rate Control (TFRC). This algorithm allows the video stream to put less strain on a congested network, improving network performance. It also decides when to send packets based on an allowed rate, which will produce less jitter, and less kernel queueing time. With packets arriving more consistently and with less jitter, this allows the receiver’s play-out buffer to be significantly smaller, also lowering the latency. No previous approach has combined all these elements to produce low latency video.
Acknowledgements

I wish to acknowledge the following people for their contributions.

- Richard Nelson, for supervising me and pointing me in the right direction when I needed it.

- Sarah Hoefhamer, for her proof reading skills, and patience.

- Sam Jansen, for all the LaTeX styles and LaTeX build system I borrowed.

- The WAND Group

- Joacim Häggmark, Magnus Erixzon, Nils-Erik Mattsson, for giving me their FreeBSD DCCP implementation with TFRC before they officially released it.
# Contents

1 Introduction 1

2 Background 3
   2.1 Latency ................................................. 3
       2.1.1 Introduction .................................. 3
       2.1.2 Video Compression ............................... 3
       2.1.3 Kernel .............................................. 4
       2.1.4 The Network ...................................... 4
       2.1.5 Receive Buffer ................................. 4
       2.1.6 Decoding .......................................... 4
       2.1.7 Synchronisation ................................. 5
   2.2 Acceptable Limits ................................. 5
   2.3 Quality Of Service ................................. 5

3 Overview 7

4 Codec 9
   4.1 Background ............................................. 9
   4.2 Previous Implementations .............................. 10
   4.3 Hardware Encoding ..................................... 10
   4.4 Design .................................................. 11
       4.4.1 Origins ........................................... 11
5 Datagram Congestion Control Protocol

5.1 DCCP introduction ............................................. 17

5.2 Current Status of DCCP ........................................ 21
  5.2.1 Standards Track ........................................... 21
  5.2.2 McManus' Implementation ................................ 21
  5.2.3 Luleå's Implementation .................................... 22

5.3 TFRC on Linux .................................................. 22
  5.3.1 Compilation ................................................. 22
  5.3.2 Linking ..................................................... 22
  5.3.3 Timers ..................................................... 23
  5.3.4 Memory Allocation ......................................... 23
  5.3.5 Locking .................................................... 25
  5.3.6 Other Changes .............................................. 25
  5.3.7 DCCP Mappings ........................................... 26
  5.3.8 Debugging and Testing .................................... 27

5.4 Rate Information ............................................... 28
  5.4.1 Introduction ................................................. 28
  5.4.2 System Call ............................................... 28
  5.4.3 Userland Buffer ........................................... 29
  5.4.4 Memory Mapped Kernel Buffer ............................. 29

5.5 Packet queueing ............................................... 30
  5.5.1 Introduction ............................................... 30
  5.5.2 Non-Blocking with Queueing .............................. 30
  5.5.3 Non-Blocking Without Queueing .......................... 30
5.5.4 Memory Mapped Queues ........................................ 31
5.6 Current Status ..................................................... 31

6 Combining CODEC and DCCP .......................................... 33
6.1 Introduction ......................................................... 33
6.2 Current Status ....................................................... 33
6.3 Rate Changing ....................................................... 33
6.4 Play-Out Buffer ..................................................... 38
6.5 Limitations ........................................................ 38

7 Testing ........................................................................ 40
7.1 Hardware Vs Software ............................................... 40
  7.1.1 Introduction ...................................................... 40
  7.1.2 Test setup ......................................................... 40
  7.1.3 Admissions ........................................................ 40
  7.1.4 Results ........................................................... 41
  7.1.5 Conclusions ...................................................... 41
  7.1.6 Validation ........................................................ 41
7.2 CODEC ................................................................. 42
  7.2.1 Introduction ...................................................... 42
  7.2.2 Test Setup ......................................................... 42
  7.2.3 Admissions ........................................................ 42
  7.2.4 Results ........................................................... 43
7.3 DCCP ................................................................... 43
  7.3.1 Introduction ...................................................... 43
  7.3.2 Test Setup ......................................................... 43
  7.3.3 Admissions ........................................................ 45
  7.3.4 Results ........................................................... 45
## 8 Conclusion And Further Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Introduction</td>
<td>48</td>
</tr>
<tr>
<td>8.2 CODEC</td>
<td>48</td>
</tr>
<tr>
<td>8.3 DCCP</td>
<td>49</td>
</tr>
<tr>
<td>8.4 Together</td>
<td>49</td>
</tr>
</tbody>
</table>

## Bibliography

50
List of Figures

2.1 A diagram showing common points of latency in Video streaming over networks. .......................................................... 5

3.1 A simplified diagram illustrating basic data-flow for a half-duplex stream, between components relevant to this project. .................. 8

4.1 A Figure showing the cycle of buffers within the CODEC. ............... 14

5.1 The DCCP state diagram from the DCCP draft[10] ......................... 20
5.2 A diagram showing the interaction between the components of the DCCP implementation ............................................... 24

6.1 A graph showing the trends both resolutions and JPEG quality on the output size. ......................................................... 36
6.2 A graph showing the trends of the output size for 64x48 pixels. ....... 36
6.3 A graph showing the trends of the output size for 720x576 pixels. .... 37
6.4 A graph showing the output size against the decay in frame rate based on scaling variables. .............................................. 37

7.1 A diagram showing the number of Frames per Second the CODEC was able to produce. ................................................... 44
7.2 A diagram showing the DCCP test and debug setup using user-mode Linux. 44
7.3 A diagram showing a DCCP connection and a TCP connection fighting for bandwidth on a bandwidth constrained connection. ........ 46
This report covers the achievements of my honours project. The project involved trying to lower the latency involved in video streaming by using a scalable intra-frame CODEC over an experimental network protocol called Datagram Congestion Control Protocol (DCCP). DCCP is still in a draft stage and there are very few implementations in existence, and no known applications designed to make use of it.

Video conferencing involves capturing video and audio at one point, and reproducing it at another. For an interactive conversation this process must happen in both directions. In order to be considered high quality and an effective means of communication, there are some properties the reproduced stream needs to have.

Firstly, the images displayed at the other end should reproduce the images captured as faithfully as possible. The image quality is composed of a mixture of the effectiveness of the compression used, how compressible the image stream is and how much data the underlying network will allow to be sent.

Secondly time is a critical factor: if there is too much delay between when the video was captured and when it is finally reproduced at the other side then interactive communication is adversely affected. This is because after the first party speaks there will be the delay until the other party gets receives it, and then further delay until the first party actually gets the response to their original statement. This delayed conversation would not pose such a problem except but informal conversation won’t always follow a
back and forth pattern where it is easy to distinguish whose turn it is so speak. This leads to problems of parties talking over each other and waiting long periods to make sure the other party has not started talking.

For these reasons keeping the latency of the video stream as low as possible is critical in making video conferencing an effective method of communication.

This project is concerned primarily with attempting to produce low latency video stream capable of being streamed over the Internet. In order to simplify the problem, this project deals specifically with only one side of the connection - a half duplex video stream. Audio has been ignored as it possesses different problems than video.

The primary outcomes of the project were a scalable streaming codec and a working Linux implementation of DCCP using TRFC congestion control. A full video conferencing system using these components has been designed and appropriate optimisations to these components identified and investigated.
Chapter 2
Background

2.1 Latency

2.1.1 Introduction

Latency is gathered from many places as the data goes through the system. Since the latency is cumulative, even small amounts of delay at many points will quickly add up, especially since any delay is likely to also occur on the return path as well. Figure 2.1 shows common points where significant latency can occur.

2.1.2 Video Compression

Some CODECS use inter-frame compression, meaning that some frames only represent the differences between itself and another frame in the stream. Backwards delta compression works compressing against previous frames, this adds a small amount of latency as the previous frame has to be compressed, then decompressed then difference between that and the new frame needs to be calculated. Forwards delta compression takes the difference against frames that are later in the stream. With this method there is even more latency added as the frame can not be compressed until the one following it has.
2.1.3 Kernel

Some parts of networking stacks employ queues where packets that can’t be sent immediately for whatever reason are queued and sent later when the kernel is able to send packets again. This queueing process can lead to large amounts of data being accepted that can’t actually be sent in a short period of time and this will add latency as it takes time for the queue to be drained. The amount of latency added depends on the size of these queues.

2.1.4 The Network

This is where the bulk of latency occurs. The physical medium will add latency and little can be done to stop this. But this is not the only problem as the Internet is made up of a mesh of links, connected together by routers. Routers can add significant amounts of latency if they queue large numbers of packets. This happens when more traffic is trying to be sent over a link than the link will support and is known as congestion.

2.1.5 Receive Buffer

Because of jitter and network instability, it is necessary to keep a buffer which can be filled in times of favourable network conditions and drained during congestion. An adaptive receive buffer can significantly reduce the latency while still stopping the stream from running out of video to play most of the time. The more stable the network conditions are, the less buffer will be required.

2.1.6 Decoding

Even with the receive buffer, it can not be guaranteed that all frames have arrived. Some may have been delayed for longer than the buffer, lost completely, or may even have bit errors. If an inter-frame CODEC is being used, it will serverly hurt any frames that are compressed against this missing frame, either skipping both, waiting for a retransmission or displaying a corrupt image.
2.1.7 Synchronisation

Audio must be synchronised with video to approximately within 258msec behind or 131msec ahead of the video\textsuperscript{4}. The time required to compress audio and video streams is different and they also will have different buffer sizes when being recorded. Which ever stream is further ahead must be delayed to fit within the bounds. For this project audio has been ignored, making this requirement not an issue.

![Diagram showing common points of latency in Video streaming over networks.](image)

Figure 2.1: A diagram showing common points of latency in Video streaming over networks.

2.2 Acceptable Limits

The level at which latency is both apparent and noticeably affecting the interaction is roughly 250ms. While this is easy to achieve over a local area network (LAN), trying to stream between countries over the Internet is much more difficult, round trip time latencies can easily be over 100ms.

2.3 Quality Of Service

Traditionally the Internet has been a best-effort network - there has been no guarantee that packets will get to their destination, and certainly no promise on how long it will take until they arrive. However there are some ideas for adding Quality Of Service (QOS) guarantees on top of the best-effort network. Protocols such as RSVP\textsuperscript{2}, Diffserv\textsuperscript{3} and
MLPS[11]. It remains to be seen whether such systems will actually get the wide spread usage to make them effective, or even then whether they will be economically viable enough for everyone to use them. Many parts on the Internet today instead rely on over provisioning and the use of congestion control mechanisms on end to end protocols such as TCP. This has the advantage of being very simple, and leaving the majority of the complexity to the end users of the system, leaving the routers free to get on with routing rather than attempting to also perform accounting.

Since QOS guarantees are yet to be available on the Internet, and there is no indication of them being available anytime soon, this project deals solely with the idea of trying to reduce latency over a best effort network.
Chapter 3
Overview

The basic solution to the problem is to have a network protocol that is kind to the network and adapts to fit the current status. This would require a video CODEC that could see what the network protocol is doing and likewise adapt quickly to the changes. This would allow for a smaller receive buffer and would lower the effects of congestion on the network.

There are three fundamental aspects to the project: the CODEC, the DCCP module and software that sits in between deciding which frames and quality settings to use. Chapter 4 covers the CODEC, Chapter 5 covers the network protocol, while Chapter 6 covers the software that interacts with them both. This configuration can be seen in Figure 3.1. Video is captured by a video capture card. It is then compressed inside the CODEC. The application then takes the images and splits them up into datagrams. These datagrams are sent over a DCCP connection to the other side where they are then reformed into frames and decoded, and finally displayed.

This solution allows latency to be lowered at almost all points. The intra frame CODEC adds very little latency. The DCCP implementation can significantly lower the latency with its rate based approach, giving less jitter, and less kernel queueing time, while using congestion avoidance to put less strain on the network lowering the latency at any congested points. With less jitter by the time it arrives this allows the play-out buffer to be significantly smaller, also lowering the latency. No previous approach has combined all these elements to produce low latency video before.
Figure 3.1: A simplified diagram illustrating basic data-flow for a half-duplex stream, between components relevant to this project.
Chapter 4
Codec

4.1 Background

Motion JPEG (MJPEG) is not actually a standard or a specified format. It is merely the idea of compressing individual frames or fields with the JPEG compression algorithm. MJPEG has the advantage of being used for many years and has built up a large application base. Also frames can be decompressed independently of each other, which makes it suitable for video editing since each frame could be edited individually rather than having it effect the rest of the stream.

MJPEG is very widespread. However, its usage is slowly declining as it gets replaced by the DV format. The DV format is supported by many digital video cameras, and features a compression algorithm which features superior quality compared to MJPEG.

With reference to this project MJPEG has many distinct advantages. The fact that it is an intra-frame CODEC makes it possible to ignore the consequences on later parts of a stream from losing individual frames. Its age and wide spread use meant that there were a lot lot of implementations written by others to build off. The other advantage is that its easy to change the rate of MJPEG by changing the JPEG quality setting, or the resolution. This is not the case with DV as it has a fixed requirement of about 25MBytes S$^{-1}$.
4.2 Previous Implementations

Since MJPEG has few intellectual property issues and has been widely used there are many tools designed to be used with MJPEGs. The most advanced set of tools for the Linux platform is the Linux Audio Visual tools (LAVTOOL)[1]. The LAVTOOLs are a set of console programmes designed for capturing video from hardware, allowing other programmes to edit it, and then more applications to compress it into MPEG or another CODEC thats more suitable for video file distribution. The LAVTOOLs are designed to work with either hardware JPEG compression boards based off the Zoran chip-set, or a standard capture card using the Video 4 Linux (V4L) API.

The programmes are modularised such that the functionality is performed in an independent shared library and the specific task is performed by the executable file.

4.3 Hardware Encoding

It had to be established whether a hardware encoder was required or if, instead a software compressor on a modern CPU, it was able to perform compression in real time at high resolutions. Since testing (see Chapter 7.1 on page 40 ) was unable to provide a definite answer as to whether a modern CPU could do real time MJPEG encoding, a Pinnacle DC10+ hardware MJPEG card was purchased. The hardware features a Zoran chip which provides hardware JPEG compression [16] . The card provides real-time encoding of an analogue video source connected to the card. The drivers for Linux, which are provided as part of the MJPEG tools package and are also included in the official Linux kernel source distribution, provide an extension to the V4L API to allow programmes to receive the compressed streams. The hardware captures from an analogue source (either RCA video or s-Video), using one of the television systems, PAL, SECAM or NTSC it is limited with these features in mind. This meant that it was primarily designed for capturing in an interlaced mode. The interlaced focus meant that the card was limited by the drivers or the hardware itself to capturing at no more than 288 pixels in height in progressive mode. This made the hardware capturing less useful as the software encoder
can easily surpass 288 pixels in height even on the highest quality modes.

4.4 Design

4.4.1 Origins

The CODEC is based off the LAVTOOL libraries. Some parts of the libraries had to be rewritten while others had to be modified slightly in order to cope with the demands for streaming. The decoding end had serious issues with interlacing and changing rates.

4.4.2 Interlacing

Interlacing is a way of increasing the quality of an analogue broadcast. Instead of sending only 25 full frames a second, 50 fields are sent. Each field only consists of either the odd lines or the even lines. The television will then display the fields combined, overwriting the old field. This method allows more complicated motion can be captured and displayed more smoothly, without requiring extra bandwidth. Progressive modes just send the entire frame. Interlacing can cause visual problems if motion becomes too rapid. While progressive would display frames that do not fit together well, interlaced video would feature visual artifacts from the fields not appearing to line up correctly.

It is arguable whether interlacing or progressive is better. While in theory interlacing gives the ability to display motion more fluidly than progressive frames, this argument is somewhat lost for video conferencing where motion tends to be minimal anyway. Interlacing creates many difficulties with respect to this project such as decreasing the digital compressibility of the images, producing twice as many images, and the extra complexity of attempting to perform rate changing with interlaced images and handling progressive inputs.

Compression

JPEG compression works by smoothing over images slightly, removing information that human eyes can not really see anyway. When trying to do JPEG compression twice on
the image, once for even lines and once for odd lines some of the ability to compress is lost since it is now trying to smooth between two lines without the line in between them which would instead reside in the next field. Interlacing also produces twice as many images resulting in more bandwidth being lost to JPEG header information. At smaller bit-rates, it is possible that this header information may comprise a significant part of the total transmission.

**Complexities**

With progressive streaming it is easy to send individual frames to the other side and they can be decoded immediately. However with interlaced images, the two fields must be kept together and be marked whether they are the upper or lower field, and then decoded when both have arrived.

Most new technology is slowly shifting to the progressive frames since digital compression can achieve superior results in both quality and bit-rate to analogue interlacing.

It is less than desirable to artificially interlace video that has been digitised in a progressive format. Deinterlacing video that has been interlaced video is not such a problem (assuming that the output device does not require interlacing) as the video will have to be deinterlaced at some point, in order to be displayed.

For reasons of compression, complexity, and future compatibility, I have chosen to use a progressive picture, using LAVREC to perform the deinterlacing if needed.

Actually removing the interlacing was not an easy task, as there were many places that assumed interlacing, and the API of the library in some places allowed the choice of progressive mode and in others it did not. Rather than spent too much time attempting to learn the internals of the library more and export the options in a more general way, I opted to change the places where interlacing was hard-coded on, to hard coding it to a progressive mode. Fortunately deinterlacing was already performed correctly by LAVREC.
4.4.3  Rate changing

Lavrec, since it is designed for capturing movies from analogue sources for later editing and or re-compression to a more suitable CODEC for movie files does not intentionally have any dynamic rate abilities. I had to investigate how and where these could be inserted into the LAVREC code.

JPEG Quality

Changing the JPEG Quality turned out to be the easiest to perform - in software compression at least. The software CODEC in the library reads the value straight from the structure that the application and the library share, each time it attempts to compress a frame. Thus simply changing it in that data structure will produce a change in the rate, effective of the next frame being compressed. The latency for this would be directionally proportional to the amount of time it took to compress a frame.

The quality is a setting between 0 and 100. However pictures encoded with below 30 feature so many visual artifacts that the picture becomes unrecognisable. Setting the value to over 90 produces extremely large images which have a virtually identical appearance to those at 90.

Resolution

Changing the resolution proved more difficult as this had to be instructed to the hardware itself. The resolution is made up of the width and height property in pixels. The aspect ratio between these is fixed arbitrarily at PAL’s 1.25 horizontal pixels for every vertical. A requirement of JPEG compression is that both dimensions are divisible by 16. This gives 33 steps between the minimum resolution the cards can capture at of 64 by 48 pixels, up to PAL’s maximum of 720 by 576 pixels.

For non-hardware compression, the Video4Linux (V4L) calls request a memory mapping for each frame, passing the resolution to the driver each time. This differs from the hardware compression where, this is only done once when the device is opened. Having to close the device and reopen it is likely to lead to missing multiple frames as buffers
are destroyed rather than drained. For this reason and the limitation of 288 pixels in height, hardware compression was abandoned for this project.

In order to perform resolution changing in software, three copies of the resolution settings had to be maintained, with primitive locking to avoid the likeliness of race conditions. This process is required because the CODEC works by passing buffers around in a circle, between the capture card, the CODEC and the application. This can be seen in Figure 4.1 the buffers are passed between each part and will queue up in between steps if one process takes longer than the others and there are spare buffers. The rate must be change must be initiated on one buffer, and the change must cycle through the system with that buffer. The process begins when the client application decides to change the rate. It modifies the copy in the shared lavinfo structure. When the library next attempts to synchronise a frame with the video capture card, it updates the copy of the resolution that is passed to the hardware. It also sets the buffer number of which it is just about to fill. Then when the encoder notices that the hardware resolution is different from the the other internal copy, it checks if the buffer number matches the one being compressed. If it does not, then the frame is still at the old resolution. Otherwise it is assumed the frame is at the new resolution, and is compressed accordingly. This process, without real atomic synchronisation measures does feature the possibility of a race condition. However any race condition will only result in one corrupt frame, and is

Figure 4.1: A Figure showing the cycle of buffers within the CODEC.
very statistically unlikely to occur. In my opinion this makes it not worth the cost of using truly atomic locking.

A better system would be to store the resolution information with each buffer, but this would require substantial changes to LAVREC.

**Frame Rate**

The final method for rate changing is frame-rate. In low motion video such as that associated with video conferencing a frame rate as low as 15 frames a second is generally acceptable. For high motion streams the lack of frames will result in a noticeably jagged and unnatural representation of what should be smooth motion. The current mechanism for adjusting frame rate is to aim for the the 25 frames a second that deinterlaced PAL video produces and drop frames accordingly to meet the desired frame rate. For video conferencing this is a perfectly acceptable method, but streams with more action would require a more refined method.

### 4.4.4 Display

One whole layer of the display code had to be rewritten in order to be more stream-able friendly. The application should be pushing frames to the display library as they frames become available, rather than the library requesting them. The main limitation was that the displaying library had no support whatsoever for changing rates via the resolution, but also that it had an issue with the progressive mode.

LAVPLAY library worked by initialising the library to a new thread, and then it would request a new frame every 1/25th of a second (for PAL). This is not a good method for streaming as the frames arrival times will feature large amounts of jitter. What makes this even worse is that the LAVPLAY code relied on copying the entire image into one of its buffers which is unneeded overhead, since the two copies are exactly the same and there is no need for the client application to delete the original buffer.

Another issue with LAVPLAY is that it would not cope with different sized images to its overlay window.
The new design works using a Simple Direct Layer (SDL) Luminance Crominance (YUV) overlay window. Whenever an image is attempted to be displayed, the library makes a very simple parse of the JPEG headers, extracting the resolution. If the resolution has changed then it will destroy the current overlay and create a new one to accommodate the new number of pixels. The overlay gives the advantage of allowing the graphics card to perform the CPU intensive YUV resizing to present a constantly sized image rather than one that grows and shrinks along with the rate. This also removes the need for software YUV to Red Green Blue (RGB) conversion as this is done in the graphics card’s hardware. Unfortunately there is a small limitation when changing resolution, swapping to the new overlay creates a small blue flicker in the output.

The library starts a new thread in order to be able to update the display even if the application has not requested a frame recently. This thread handles requests to display an image from the application. The image is then decompressed directly from the applications receive queue to the overlay’s buffer. Once finished the overlay is then updated onto the screen, producing video without any flicker. This makes one less copy than the original LAVPLAY.

4.4.5 Scaling

The will not scale to a given rate, instead this behaviour is left to the client applications. It will however respond to requests to change either the resolution or the JPEG quality setting.
Chapter 5
Datagram Congestion Control Protocol

5.1 DCCP introduction

Datagram Congestion Control Protocol (DCCP) is a new protocol being ratified by the Internet Engineering Task Force (IETF). It is designed to fulfil the needs of an emerging traffic type.

Until recently the majority flows across the Internet have fit into one of the following categories:

- Simple file transfer (E.G. HTTP, FTP, E-Mail)
- Low bandwidth Interactive session (E.G. Telnet, SSH)
- Simple query (E.G. DNS)

All these types of flows have been a perfect fit for the predominant transport protocols of Transmission Control Protocol (TCP) [15] and User Datagram Protocol (UDP)[14]. However, new flow types are originating that TCP and UDP can not fully provide for. Interactive high bandwidth flows tend to want to avoid the latency and bandwidth costs of a reliable in order transport protocol. As more live multimedia content is streamed
through the Internet these streams will require a standard congestion control algorithm to avoid congestion.

Most of these current applications such as games and Voice Over IP (VoIP) use UDP since it doesn’t add the latency that TCP does. This creates a problem because UDP has no form of congestion control as it was designed originally as a protocol for simple queries [14].

While it is possible for each application writer to write congestion control into their application layer protocol, this is not a good solution. Congestion control is too important to leave up-to individual writers. Not only must the algorithm know when to send at a slower rate in times of congestion, it is imperative that it does this in a similar and compatible way to the other flows in order to fairly distribute the available bandwidth between flows[5].

**Founding Ideas Behind DCCP**

DCCP’s basis is an unreliable protocol designed for high bandwidth tasks that require lower latency than TCP offers. It resembles TCP but lacks the reliability and also offers a datagram service rather than a stream. Like TCP, a connection can be thought of as being comprised of two separate half connections with a unidirectional data flow for each half connection. This can be seen in Figure 5.1 with the state from closed to request after a request packet is sent. The other side then transitions from listen to respond sending a respond packet. Finally the client sends an acknowledgement bringing the state to open.

DCCP also makes use of a service code in its connection identifier, allowing different services to operate on the same port. This is in addition to the Internet Protocol (IP) numbers and port identifiers that make up a TCP connection.

**Negotiation**

Connection initiation requires a three way handshake between the end hosts. Peers also use these packets and the ones following it to negotiate options for the connection.
These negotiations continue until both sides agree on the options for each half connection. Only then are application datagrams are allowed to be transmitted. But rather than just having one congestion control algorithm to determine when a packet is allowed to be sent, the hosts decide which algorithm to use during the negotiation phase. The algorithms are refereed to by Congestion Control Identification (CCID)[10].

**DCCP’s Congestion Control Algorithms**

There are currently two defined congestion control algorithms, CCID 2:TCP-like [6] and CCID 3:TCP Friendly Rate Control (TFRC) [7]. TCP-like is as close to TCP’s congestion as possible for an unreliable protocol. It makes use of a congestion window, shrinking and widening it in the same way TCP would.

**Tcp Friendly Rate Control**

TFRC is designed for live streaming of multimedia by regulating the time between sent packets. It achieves this by calculating a rate using information gained from feedback packets from the receiver. It then tells its client application how much bandwidth it may use. This rate is calculated in such a way that it ends up sending a similar amount to what TCP would allow under the same conditions. This has the advantage for the streaming client of knowing the rate it can set its CODEC’s at to keep within the confines of the congestion control algorithm. Limiting based on a given rate also allows for a more constant stream of packets helping to minimise the jitter and bursting behaviour of TCP. This effect happens because TCP and TCP-Like make use of a window, of how many packets or bytes are allowed to be unacknowledged in the network at any one time, this leads to the behaviour of sending as many packets as possible then waiting for acknowledgements to arrive before sending another burst. TFRC on the other hand allows a given number of packets per be sent a second, encouraging them to be sent at regular intervals. TFRC works by the receiver sending three variables back to the sender for every feedback packet. These variables are the rate at which it has received data, the packet loss event number, and the window counter. The sender then takes these
variables and works out a new rate using its equation. However, it places a bound on it such that it will never send at a race more than twice what the receiver has actually reported to have received and that it will never fall below one packet every 64 seconds [9].
5.2 Current Status of DCCP

5.2.1 Standards Track

The protocol hasn’t changed significantly in the past year. The option negotiation had to be changed as it was flawed under certain conditions. Most other modifications to the draft documents have been on improving the phrasing of the specification. The main thing holding it back from being standardised are the lack of complete implementations and real world evidence that it solves the problems.

5.2.2 McManus’ Implementation

There is currently only one publicly released implementation and it is now quite outdated and features many problems. The patch by Patrick McManus is for the Linux kernel 2.4.18. It is currently not being actively maintained, has no documentation and is incomplete. [12].

The major limitation of the implementation is that it only supports CCID 2 TCP like congestion control. All through-out the source code, it just assumes that it is using CCID 2, and because of this does not feature any way to easily add in other congestion control algorithms. It also misuses one variable to store the CCID for both sides of the connection and this is is contrary to the draft standard where both half connections are supposed to be able to feature different CCIDs.

I found the potential for a denial of service attack against any host with an open port. Sending a packet with a malformed option will put the kernel into an infinite loop, quickly rendering the machine useless on a single processor system. Interestingly this same packet will also crash the ethtool patch for DCCP.

Another minor issue, that will only effect interoperability is that the checksum does not include a virtual IP header in its calculation. This is most likely because the implementation was written prior to this being included in the specification.
5.2.3 Luleå’s Implementation

A group at Luleå University in Sweden are working on a full implementation for FreeBSD 5.0, which was recently released, they gave me a patch of their current version. They also released the Ethereal patch for DCCP that they used to help develop their version. Luleå’s implementation is mostly complete, with the only major missing parts being IPv6, mobility and some of the security aspects. It is designed in a modular way, defining a programming interface through which CCID’s can perform their tasks without major modifications to the bulk of the DCCP code base. It provides separate interfaces for the sending side and the receiving side of the same host connection. This allows for each half connection to have a different congestion control scheme, unlike McManus’ implementation.

5.3 TFRC on Linux

The project required a DCCP implementation that featured the TFRC congestion control algorithm that was based on the Linux kernel. For this reason I decided to make use of McManus’ Linux patch and try to incorporate the TFRC algorithm from Luleå into it.

5.3.1 Compilation

The first thing to do was to make the TFRC code from Luleå compile under Linux. This required copying some of the FreeBSD kernel headers onto Linux. The major one was queue.h, a simple generic queue implementation in C. This did not present many challenges.

5.3.2 Linking

The next stage was to try and make it link against the Linux kernel and McManus’ DCCP patch. The main part of this was implementing functions that map parts of the FreeBSD kernel that TFRC relied on to the Linux kernel, and to map the parts of
Luleå’s DCCP to McManus’ version. While FreeBSD and Linux share similar history and the basic idea of a UNIX like kernel. Unfortunately this likeness do not flow into the design of the internal kernel calls. The US Navy found while making an IPV6 and IPSec implementation that was portable between BSD OS, FreeBSD, OpenBSD, NetBSD, and Linux that between 20 and 40% had to be specifically written for each system [13].

The basic layout for how the separate code-bases interact and how they had to be modified can be seen in Figure 5.2.

### 5.3.3 Timers

Timers proved especially problematic and essentially required rewriting all timer parts of the code in the TFRC implementation. The FreeBSD implementation of kernel timers works by having a set of methods for creating and modifying timers that return a pointer to a structure which has been allocated by the functions. In comparison, the Linux functions take an already allocated structure and use it. All pieces of code that used timers were commented out using a C precompiler conditional that is defined to compile the code on Linux. The FreeBSD functionality was then written using Linux.

### 5.3.4 Memory Allocation

The kernel memory allocation (MALLOC) functions were almost the same on both kernels. However the FreeBSD malloc() provides a method using the M.ZERO flag where it requests that the memory be zeroed first, as Linux provided no option similar to this in its kmalloc(). So the MALLOC call had to be mapped via an intermediate function that checked for the M.ZERO flag and write zeros to the memory after calling kmalloc().

A small translation was also required between FreeBSD’s flag to request that malloc doesn’t sleep to make it compatible with Linux’s.
Figure 5.2: A diagram showing the interaction between the components of the DCCP implementation.
5.3.5 Locking

Some data structures which can be accessed by the kernel from both the user context and in an interrupt context need locking to avoid race conditions. FreeBSD makes use of a Mutual Exclusion (MUTEX) structure for locking. Linux has many different structures for locking: semaphores, spin locks, and read-write versions of both. Since the code needs to protect from interrupt handlers and be locked in interrupt handlers, spin locks seemed the best choice. Spin locks are a quite primitive form of atomic locking. They consist of an integer, and when a thread wishes to grab the lock, then it disables interrupts and does an atomic subtract, and checks on the value. If the result is zero then the thread has the lock. If however, it is greater than one then the thread keeps checking until it acquires the lock. While not as efficient as a mutex, it has the advantage that it will never sleep and it can be optimised away in uniprocessor machines to just disable interrupts.

One possible problem is that places in the TFRC code request a recursive MUTEX. A recursive MUTEX allows the same thread that is already in possession of the lock to re-request the same lock without it deadlocking. There is no suitable Linux kernel mechanism for a recursive lock, although some form of thread local storage should make it possible. Recursive locks are a poor design choice as they can encourage less care and understanding in using locks.

A distinct probability for the programmers inclusion of a recursive lock is that TFRC calls dccp_output(), which can in turn call TFRC functions. This would lead to a deadlock otherwise. My use of TFRC does not follow this behaviour. No deadlocks have been encountered from these recursive locks. There is also no indication in the code that the recursive feature is actually used within TFRC.

5.3.6 Other Changes

McManus’s patch had to be modified to actually make use of TFRC. When packets are received it must tell both the TFRC sending congestion control algorithm and the receiver (assuming that both half connections are using TFRC). Ideally, this should not
happen in the interrupt context since interrupts are disabled and should re enabled as quickly as possible. However, McManus’ patch drops feedback packets before they reach a bottom half. Since there was little documentation, it was hard to know how much work would be involved in shifting this compared to what would be required to make TFRC interrupt context safe. Since TFRC didn’t appear to feature any functions that might sleep in its tfrc_send_packet_recv and tfrc_recv_packet_recv the latter option was chosen.

TFRC also had to be asked if data packets could be sent. This was achieved by calling tfrc_send_packet, which returns 1 if a packet is allowed to be sent and 0 otherwise. This function also allocates necessary structures to allow TFRC to keep track of packets that have been sent.

Another call had to be made after a packet was actually sent. This allowed TFRC to know what packets had been sent and hence when combined with the acknowledgement information, know which were packets were outstanding.

Although CCID negotiation is rigged, TFRC is still not initialised until after the negotiation has been confirmed by both sides. When the connection is closed then TFRC must likewise be destroyed.

5.3.7 DCCP Mappings

TFRC depends on being able to get the feedback options from the other side and send the other side feedback.

Option reading was achieved by copying Luleå’s code into the code base, since it is only parsing a string rather than any kind of internal structure. Allowing TFRC to set outgoing options was a little more challenging. McManus’ patch includes a small buffer for options to be appended to the next packet, and a counter of how many bytes have been used so far. It was important to check for a buffer overflow in the case that the new operations would write past the end of the buffer. Then the options were added to the buffer and the counter was incremented appropriately to reflect the new size. If there was not enough room in the buffer, then the function could return false leaving
TFRC to deal with this fact appropriately.

Having added feedback options, TFRC required the ability to actually request a feedback packet be sent using the dccp_output() function. Since this call often comes from an interrupt context, it is not allowed to sleep. McManus’ acknowledgement code does have the possibility of sleeping so instead of calling it directly, the map makes use of the Linux kernel’s task scheduling and adds a task to be scheduled on the next context switch.

5.3.8 Debugging and Testing

Once linked, focus moved onto testing. Due to the stability issues that are inherent in untested kernel code, a strong development platform had to be chosen. User Mode Linux (UML) was selected for its fast performance, ease of use and strong ability to be debugged with a source level debugger.

One problem that occurred was the method used for timing. My initial implementation of FreeBSD’s microtime() call just used the standard Linux timing mechanism of jiffies, which is only updated every 10 microseconds. This proved problematic later on if the receiver attempted to send two feedback packets in a 10 microsecond time-frame. It resulted in the time delta being zero and this meant in the feedback packet reported the bandwidth received as 0 bytes a second. This caused a very harsh response from the sender that is bound to send no more than twice the rate that the receiver reports as receiving.

This was fixed by instead basing the timing off a kernel function that checks the Real Time Clock (RTC). While a little more CPU intensive, at least it doesn’t break the feedback.

User-mode Linux (UML) is a port of the Linux kernel to Linux. It allows Linux kernels to run under a real Linux system running the kernel in the user space. This essentially allows one physical machine to pose as both development machine and multiple testing machines, without having to have a full machine reboot every time a testing kernel crashes.
A gain from working under UML was the ability to debug the kernel using a high level source code debugger such as the GNU Debugger (GDB).

A further gain from using UML was the ability to use network emulation tools and packet capture on virtual networks between multiple UML instances to emulate interesting network properties and ensure protocol correctness from packet captures.

Most development was done with two UML instances communicating with each other using tun/tap devices on Linux. Latency and router queues were added using NISTNET, in order to test behaviour to various network conditions (see Chapter 7.3 for more details).

5.4 Rate Information

5.4.1 Introduction

Ideally rate information from TCP Friendly Rate Control (TFRC) should be made available to the user-land application. In the video conferencing context, this information helps the application know what rate the video encoding processes should be set to. Most other uses of DCCP and particularly TFRC could make use of the information as well. There are various mechanisms in the Linux kernel to pass information from the kernel to a user-land process. The information should be passed on a per socket basis. Each different DCCP with TFRC socket should get its own rate information. The makes non socket operation methods such as the /proc interface less ideal as they are not easily based on working with individual socket descriptors. This leaves three main options, an ioctl system call to return the information, a user-land memory area that is written to by the kernel, or a region kernel memory to be exported to application using the memory map system call on the socket.

5.4.2 System Call

A standard ioctl system call certainly has its advantages. It is very easy to code at both the kernel level and the user-land process. However every system call has a distinct time
penalty. If the application wants to check the value often through polling to get quick responses to changes then this could lodge a measurable time penalty.

5.4.3 Userland Buffer

Another solution would be to have a setsockopt system call that passes a user-land buffer that has been allocated by the application to the kernel, which it can continually copy the information into. This is a little more complicated than the IOCTL, but avoids the polling issue. However the kernel has no way of guaranteeing the user-land buffer will be loaded in the physical ram, rather than paged out to the swap space. This could introduce a performance problem as the thread that is attempting to update the information may have to sleep while the page is loaded into ram. This has serious issues since in most cases the rate gets changed in an interrupt context.

5.4.4 Memory Mapped Kernel Buffer

Instead, my implementation of DCCP allocates a kernel page of memory for the buffer. It requests that the VM mark this page to always remain in main memory and hence is never paged out to disk. The client application accesses the buffer by mapping the memory with an memory map call on the socket descriptor. This removes the polling problem and the sleeping problem. It does however place an extra demand on four kilobytes of memory being constantly in RAM.

Currently the system uses shares a single rate value with 128 bit floating point precision. It would be trivial to use other parts of the memory page to share more information that clients might be interested in such as packet loss information and estimated Round Trip Time (RTT).
5.5 Packet queueing

5.5.1 Introduction

With the aim of decreasing overall latency there arises the problem of whether or not to queue packets that the application wishes to send but that the congestion control algorithm isn’t currently allowing DCCP to send.

5.5.2 Non-Blocking with Queueing

Both McManus’ implementation and Luleå’s have the notion of queueing packets and sending them later when the congestion control allows it. Ideally such decisions should be left up to the application as these can be latency inducing operations. The decision is whether to use a standard queue or instead, telling the application the packet can’t be sent yet. This comes down to a question of whether the increased overhead of calling system calls to attempt to send packets that can’t be sent is preferential to the latency incurred in queueing packets where it may be more preferential to drop them. There exists another solution though, that differs from the conventional socket interface.

5.5.3 Non-Blocking Without Queueing

My current implementation returns EWOULDBLOCK on any attempt to write to the socket where the congestion control algorithm disallows it. It is then up to the application to decide what to do. This is a performance problem though as the send system call currently has quite a lot of cost to it. My tests (see Section 7.3 running under User Mode Linux (UML) showed that the call took between 0.0001 and 0.001 seconds. This makes it a very expensive and slow polling operation, and is unlikely to be effective in its current state for high throughput situations.

However since the TFRC code remains mostly unchanged and was designed with queueing in mind, there most likely exist many optimisations that could be put in place to remove the overhead of the polling if it is obvious to the congestion control algorithm that it is not yet time to send a packet.
5.5.4 Memory Mapped Queues

Even with an efficient polling method, it is still polling through a system call which will still have a large impact on performance. A much better solution is to have a queue that removes overhead of polling for testing if a packet can be sent. The difference from previous implementations is that instead of a regular network kernel buffer, have a queue that can be modified by the application right up till a packet is sent. A group at ICSI Centre for Internet Research (ICIR) have designed an API that allows this and are said to be working on an actual implementation. This API would have to deviate from the traditional socket operations of send and recv. Instead, the application could memory map a kernel buffer in a similar fashion to the rate information in Section 5.4. An implementation would have to be very careful about race conditions between user land and kernel land, especially with symmetric multiprocessing and the modern generation of preemptable kernels.

5.6 Current Status

The implementation is complete feature wise, and appears to behave according to the draft specification for congestion control. However, many things lacking in McManus’ patch are still not implemented. For example it still doesn’t handle negotiation of congestion control algorithms correctly, while the situation has been improved, there still exist parts where the code assumes that it must use CCID 3 for both sides of the half connection. Other omissions include

- Mobility
- Oscillation Prevention
- IPv6
- ECN support
- Path MTU discovery
The DCCP kernel module does, unfortunately, have kernel panic bugs still in the code. Testing has found that there remains at least two major bugs. One bug relates to corrupting the Kernel’s timer list by somehow injecting an invalid pointer into the list. This causes an attempt to dereference the invalid pointer inside the kernel timer running procedure. Despite making use of list and slab poisoning, I have been unable to trace what causes the corruption. It is difficult to know when the list gets corrupted as it is only noticeable when it actually attempts to execute that timer. All timer operations are either performed in the interrupt context, or with interrupts disabled so there is little possibility of a race condition.

The other bug which is less frequent, is a kernel panic on a null pointer inside an optimised version of the memory copy function. This has been difficult to track down, as it must be a side effect of my code rather than a direct problem as the memcpy function is only ever called in McManus’ code.
Chapter 6

Combining CODEC and DCCP

6.1 Introduction

DCCP and the CODEC can be used together to produce a high quality video stream. The components used to perform this can be seen in the striped boxes in Figure 3.1. The basic purpose is to get the feedback information from DCCP and decide how to adjust the CODEC to fit in with this feedback. It also must deal with splitting frames up into multiple packets, and putting them together again at the other side. Another aspect that needs to be dealt with is the play-out buffer.

6.2 Current Status

Without a usable TFRC implementation on Linux, the majority the interesting features of this programme could not be tested. However, it currently streams video over UDP and will allow keyboard input to manually adjust the rate.

6.3 Rate Changing

Background

The CODEC (See Chapter 4) does not itself provide a way of setting the rate to a certain number of bytes per second. Instead it exports methods for adjusting the compression
and size of the image outputted. The programme must then decide how to adjust these to meet DCCP’s requirements.

**Current Implementation**

The current implementation is an adaptive method that looks at the current rate produce and the desired rate. If the desired rate is less, a heuristic attempts to lower the rate. Otherwise it attempts to increase it. Since the CODEC allows for smooth scaling having to change it frequently is not an issue.

**Adjustable Variables**

There are three software controlled variables in the encoding process, resolution, JPEG quality and frame-rate. There is also one major non-software controllable variable, and this the compressibility of the image being captured. Because of this uncontrollable variable, it is important that the rate scaling be a dynamic model that looks at how it can adjust settings to affect the rate. Rather than the alternative of a static lookup table where a particular rate is encoded using a predetermined setting from the table.

**Trends in Scaling**

Tests were performed to consider what effects each variable has on the scaling (see Chapter 7.2 for details of how this was performed)

Increasing the resolution increases the output rate exponentially this trend can be seen in Figure 6.2 and Figure 6.3. Increasing the JPEG quality setting also leads to an exponential increase in file size which is illustrated in figure 6.4. The possible rates of output from combining the variables together can be seen in 6.1. This graph shows that there are a multitude of ways of making up a particular rate. Especially when considering the frame rate as a variable also. Since image size adjustments require changing hardware parameters on both sides of the connection causing a blue flicker to be produced, resolution swapping should be used as a last resort. Dropping frames
should also not be used regularly as this will severely degrade high motion sequences. This leaves the quality setting as the most preferential attribute to modify.

**Proposed Algorithm**

My proposed algorithm would primarily adjust the quality setting to attempt to achieve the rate. However, if it attempts to scale above a quality setting of 90, then it should instead change resolutions and drop the quality back down to 55. It is imperative that when selecting a new resolution, the selected resolution does not exceed DCCP’s given rate. Altering the resolution will still produce four frames at the old rate until there is a frame at the new rate. If it was decided when the first frame was produced at the new resolution, that the resolution was too high, and had to be changed. Then this would leave two options: attempt to send the produce four frames produced at this resolution before getting it right. It is likely that one or more of these frames would have to be dropped in order to not meet DCCP’s rate restrictions. This will result in a resolution change lasting between one and four frames. This is not a good solution as it will produce multiple blue flickers. The alternative would be dropping 4 frames in a row which will also severely affect the perceived quality of the stream by not showing any motion that occurred during those frames.

It is also important to not set the resolution excessively bellow what it should be, this will lead to a drop in quality and more blue flickers if the resolution has to be increased again.

Based of the testing data, it would appear that switching between the resolutions tested would be enough. A jump from 64x48 pixels at a quality of 95 produces more or less the same rate as 176x144 at a quality of 50. Following this 176x144 at quality 95 yields a similar rate to 384x304 at a quality setting of 50.

**Responding to rate halving**

Under harsh congestion, it is very likely that DCCP will request that the CODEC immediately half its sending rate.
Figure 6.1: A graph showing the trends of both resolutions and JPEG quality on the output size.

Figure 6.2: A graph showing the trends of the output size for 64x48 pixels.
Figure 6.3: A graph showing the trends of the output size for 720x576 pixels.

Figure 6.4: A graph showing the output size against the decay in frame rate based on scaling variables.
6.4 Play-Out Buffer

The play-out buffer is currently implemented, in a way that it will form a queue of all frames not yet played and will clear all frames that are older than the currently displayed frame. For an effective play-out buffer, there needs to be communication between the streamer and the listener about the status of the network. If there are few disturbances in the network and frames arrive at a consistent rate rather than following a bursting behaviour, then there is no need for a play-out buffer at all and frames can be played as soon as they arrive.

Any play-out system really needs to be carefully adjusted since it is trying to archive a balance between the added latency added by the buffer and any smoothness lost from draining the buffer and running out of frames to display temporarily - it is something that is hard to measure without human observation.

6.5 Limitations

One of the biggest limitations is attempting to use lines which feature large and regular amounts of packet loss. The CODEC will only process frames that have been fully completed sending. At higher rates the image sizes will grow to be over 200KBytes. Assuming an Ethernet MTU of 1500 bytes, then 133 packets will be required to be sent to transmit a single frame (ignoring the small overhead from the application protocol information). With even a small amount of packet loss almost all frames will not be able to be displayed even though the majority of the data is available.

While not attempted in this project, if the lost data was part of the image rather than the JPEG header then padding with random data make make the majority of the frame viewable while corrupting a tiny part of the image.

There would be many more effective approaches such as including some form of redundancy, or attempting to guess at missing portions by looking at the rest of the image. These would add a lot of complexity to the CODEC and would require a very strong knowledge of the mathematic compression algorithms such as Discrete Cosine
Transformation (DCT) that are prevalent in JPEG.
Chapter 7
Testing

7.1 Hardware Vs Software

7.1.1 Introduction

In order to decide whether a hardware decoder card was required tests were performed using the JPEG library[8]. In these tests, an image comprised of randomly coloured pixels (a worst case scenario for compression) was compressed using JPEG compression.

7.1.2 Test setup

To avoid testing bottlenecks in the process other than software encoding, the image was loaded into memory as it would be if it was captured from a video device. The programme would then attempt to encode the image repeatedly and would report the total execution time to encode the image 100 times. Compressed output was written to a memory. The encoder would then start over, re-encoding the image again.

7.1.3 Admissions

The testing is not entirely accurate, as it was using the same image each time and could lead to parts of the image being stored in the CPU cache for the subsequent encodings. It is possible that the randomised image randomly formed an image that was not close to a true worst case scenario. Given that the purpose of the test was to form an indication
of whether software was an acceptable method to use, these risks are unlikely to have a 
huge effects on the validity of the conclusions.

7.1.4 Results

<table>
<thead>
<tr>
<th>JPEG Quality</th>
<th>Resolution</th>
<th>Frames per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>720x576</td>
<td>25.0</td>
</tr>
<tr>
<td>10</td>
<td>720x576</td>
<td>23.1</td>
</tr>
<tr>
<td>20</td>
<td>720x576</td>
<td>23.1</td>
</tr>
<tr>
<td>30</td>
<td>720x576</td>
<td>23.1</td>
</tr>
<tr>
<td>40</td>
<td>720x576</td>
<td>23.1</td>
</tr>
<tr>
<td>50</td>
<td>720x576</td>
<td>23.1</td>
</tr>
<tr>
<td>60</td>
<td>720x576</td>
<td>23.1</td>
</tr>
<tr>
<td>70</td>
<td>720x576</td>
<td>21.0</td>
</tr>
<tr>
<td>80</td>
<td>720x576</td>
<td>20.0</td>
</tr>
<tr>
<td>90</td>
<td>720x576</td>
<td>18.75</td>
</tr>
<tr>
<td>100</td>
<td>720x576</td>
<td>12.0</td>
</tr>
</tbody>
</table>

7.1.5 Conclusions

The results were inconclusive as to the purpose of the test. It showed that a fast CPU 
could probably handle most resolutions and quality settings but would have problems 
keeping up at the higher resolutions and quality settings.

7.1.6 Validation

Final implementation and use of the software CODEC proved these results to be accurate 
and validated that the software encoder dropped very dramatically in frame-rate at upper 
limits. Figure 7.1 shows the decline.
7.2 CODEC

7.2.1 Introduction

The CODEC had to be tested to ensure that it could perform encoding at high rates, it also had to prove that it could scale the output rate and also that it was stable enough.

7.2.2 Test Setup

Frames per Second and Rate changing

The tests were run using an application that used the CODEC to capture from an analogue video camera plugged into a Brook Tree 878 based video capture card. Five resolutions (64x48, 176x44, 384x304, 576x464, and 720x576) and five quality settings (30,50,65,80,95) were selected. The selections were designed to concentrate around the extreme cases and also resolutions and quality settings that would be most useful. Any quality setting under 30 produces output that is visually useless, and anything over 95 is exceptionally larger without being any better in detectable visual quality making these numbers the upper and lower bounds. The application is programmed to generate statistics such as the number of frames per second and the total size of frames produced during that period. The application was manually told to begin recording results and also manually told to stop.

7.2.3 Admissions

The timing of 20 seconds was performed manually, but based off watching the computers timer. Results were taken at the end of this, though this timing does not directly affect the results, it was simply the measure of time that the rate had to converge to its average performance. The tests were performed on a desktop system where other applications were running, and hence individual results may be questionable, but overall trends will still be valid.

The CODEC was attempting to compress a live analogue source that would not be returning exactly the same frame each capture. It is probable that lighting conditions
changes very slightly between captures.

The timing within the application was only accurate to the nearest second.

7.2.4 Results

Frames per Second

Like the initial results seen in Section 7.1 the results appear to show that the CODEC can perform in most resolutions but begins to struggle at the high end. The graph of the results shown in Figure 7.1 seems to be a mirror of the rate output seen in Figure 6.1. Unfortunately the upper bound on the encoding is always $25 \text{ FS}^{-1}$ as that is all the BT878 card will deliver. By combing the results it can be seen that the CODEC is able to keep the FPS at a usable level until the output rate reaches around 100KBytes.

Output Rate

The graph of the output rates (Figure 6.1) shows that there is a good selection of rates available and multiple of ways of obtaining most rates.

7.3 DCCP

7.3.1 Introduction

In order to continue any further work integrating video and networking together, DCCP had to be tested to ensure it was stable, had acceptable performance and followed the DCCP draft documents, except where already noted otherwise.

7.3.2 Test Setup

The setup for testing my DCCP implementation was using two User-Mode Linux virtual machines networked together through a tun/tap virtual Ethernet device in the host machine. This network could be monitored by Ethereal using the DCCP patch produced by Luleå and could have network emulation applied using NISTNET.
Figure 7.1: A diagram showing the number of Frames per Second the CODEC was able to produce.

Figure 7.2: A diagram showing the DCCP test and debug setup using user-mode Linux.
Testing was undertaken by running a server on one UML machine that would wait for a connection. The other UML machine would connect to it and then after a five second delay, begin sending data as fast as the congestion control algorithm would allow it. The server would accept this data and then do no further processing on it.

A verbose amount of debugging information could be logged through the kernel system logging mechanisms. This could be read later and used to determine irregular behaviour.

7.3.3 Admissions

All tests were performed on the UML setup described in 7.3.2 and shown in Figure 7.2. UML introduces a performance penalty of at least 30%, making any conclusions about real world performance problematic. UML is also a recent development and it is possible that it will exhibit different properties than the kernel running on a real machine.

7.3.4 Results

Stability

Testing concluded with there still existing at least three introduced kernel bugs.

- Kernel timer list corruption, causing kernel fault during timer processing.
- DCCP memory corruption, causing segmentation fault during memory copy.
- Failure to free memory on receive side, causing a kernel fault when unloading module.

TCP Fairness and Congestion Control

DCCP and TCP appear to play more or less fairly. Figure 7.3 shows two applications trying to send as much data as possible, one using a DCCP socket, the other TCP. It can be seen that they establish an equilibrium with each other eventually, rather than one application continuing to take up more and more of the connection. It can also be
seen that DCCP responds less harshly to the congestion, with a lower amplitude in its oscillations. This would be even less with TFRC’s oscillation prevention implemented.

![DCCP Bandwidth Vs Time](image)

**Figure 7.3:** A diagram showing a DCCP connection and a TCP connection fighting for bandwidth on a bandwidth constrained connection.

**System Call Performance**

System call performance on the send call was relatively fast with debugging messages turned off. It was as low as 0.0001 seconds, averaging around 0.0005 seconds. This would allow for a maximum of around 2000 packets a second. This will be made a lot worse by UML as it has to trace system calls.

**Bandwidth Performance**

Bandwidth performance is still well below that of TCP, and appears to be CPU bound. The bandwidth peak was measured by NISTNET at approximately 300MBytes.S⁻¹ in
contrast to iperf which managed 1300MBytes$\text{s}^{-1}$. This follows since the packets being sent were 300 bytes, and the result of 2000 packets a second above would translate to a theoretical maximum of 600MBytes$\text{s}^{-1}$

**Rate reporting**

Manual observation proved the the reported rate never grew above twice the possible rate, and tended to be around the rate that was actually being sent.
8.1 Introduction

In this project, I have written a scalable CODEC and produced a working version of DCCP with TFRC on Linux. I have investigated different methods for implementing aspects of DCCP. I have also investigated methods of how to build a video conferencing system based on available rate feedback.

8.2 CODEC

I produced a low latency scalable MJPEG CODEC. The CODEC proved to be highly scalable, being able to change rates from as low as 15 KB/s to as high as 4.5 MB/s. The codec will currently stream over a UDP connection.

I have found that interlacing is not an efficient method for digital compression, especially low motion video conferencing, it removes some of the information that is needed for easy compression and makes many things more complicated.

I also found that while hardware allows the higher end rates to be captured, it places too many unneeded constraints such as interlacing on the images to achieve this, making it not suitable for such purposes.

Future work could look at implementing per frame forward and backwards error correction, using delta compression on acknowledged frames and using multiple areas...
for quantisation to produce higher quality in the more important areas, similar to DV’s algorithm.

8.3 DCCP

I modified McManus’ patch for Linux to incorporate TFRC. This is the first implementation for Linux with TFRC support.

I also investigated methods for passing rate information to client applications. Memory mapping is the most effective way as it never has to sleep and can be both updated and checked with very little processing overhead. This method was implemented in my DCCP implementation.

Future work would entail completing the implementation, removing bugs and implementing a high performance queueing mechanism.

8.4 Together

I investigated methods for scaling the CODEC based on the rate from TFRC.

Future work could look at implementing my proposed algorithm for rate changing. It would also be useful to compare it to other ideas, possibly some involving machine learning. It would also be possible to look at the addition of a scalable receive buffer.


