Project Number: MLC AF03

Adaptive Forward Error Correction for MPEG Streaming Video

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Colin Bourassa **Mark Figura** **Michael Scaviola**

Approved:

Professor Mark L. Claypool, Advisor

Professor Robert E. Kinicki, Advisor

1. FEC
2. MPEG
3. streaming

ABSTRACT: To achieve optimal performance, streaming video requires a TCP-Friendly data rate as well as resistance to lost packets. This project implements an adaptive forward error correction (AFEC) system with temporal scaling for streaming MPEG, which repairs video frames in the event of packet loss. Data is transferred via a TCP-Friendly UDP flow which responds to network conditions to reduce congestion and optimize performance. Adaptive FEC is comparable in performance to fixed FEC, and in some test cases, outperforms it.

Table of Contents

List of Figures

1. Introduction

As the capabilities of modern computers have increased, streaming video has become ubiquitous on the Internet. With broadband and high-speed Internet connections, users are able to watch news clips, sports games, or even hold video conferences. Despite its widespread adoption, streaming video can suffer in performance due to the network conditions of the Internet.

Unlike other Internet applications, the real-time requirements of streaming video allow it to handle some loss of data at a cost to performance but make it very sensitive to network delays. TCP, the transport protocol used in most Internet applications, is not suitable. TCP does not provide a steady data rate, and while it does ensure that lost data is retransmitted, the retransmitted packets are not useful to the video stream because the retransmission delay causes the packets to arrive too late. To achieve the best performance, streaming video requires a constant, reliable data stream. Often, the UDP protocol is used, but it does not respond to network congestion as TCP does, which can exacerbate congestion. If UDP is to be used, the application has the responsibility of keeping congestion at a minimum. To do this, streaming applications should utilize TCP-friendly streams, meaning that the data rate does not exceed that of a TCP connection under the same network conditions.

One technique for controlling data rate involves scaling back the data rate of the video to use less bandwidth under congested network conditions. Three types of scaling are commonly used: spatial, quality, and temporal scaling. Spatial scaling reduces the dimensions of the video, producing a smaller data rate by using fewer pixels. Quality scaling reduces the data rate by increasing the amount of video compression. Temporal

scaling removes some of the video frames to achieve a lower data rate. Scaling may reduce congestion, and consequently, reduce the amount of lost video packets. However, lowering the data rate of the video does not address the issue of packets already lost, nor does it protect against packets being lost in the future.

Another approach to protect video from packet loss is to use Forward Error Correction (FEC). FEC comprises extra data that is sent along with a video stream and is used to repair video data that might be lost due to network conditions. For a video frame consisting of K packets, (N-K) FEC packets are added. N packets are transmitted for the frame, and if at least K packets are received, the frame can be reconstructed. When a packet is lost, the receiver can use the extra FEC data to reconstruct the missing frame. In this case, a fixed amount of FEC is applied to the video stream to compensate for any expected loss rate. The problem is that the addition of FEC increases the total data rate, so the actual video data rate must be reduced in order to avoid increased congestion, thus sacrificing video quality.

An approach designed by Huahui Wu [1] adjusts the amount of FEC added to a TCP-friendly MPEG video stream. The scheme relies on feedback from a TCP-friendly rate control protocol to predict the probability that frames will be transmitted successfully, and therefore, the amount of FEC to add. Additionally, temporal scaling is applied to optimize the playable frame rate for a TCP-friendly stream. By adding the optimal amount of FEC, there is no wasted overhead. Although a system based on this method had not yet been built, simulations have shown that this adaptive FEC approach is very effective at improving the playable framerate of streaming video.

The purpose of this project is to implement adaptive forward error correction (FEC) for streaming MPEG video, and test whether it works as simulations suggest. In particular we implemented the adaptive FEC scheme designed by Huahui Wu, which optimizes the amount of FEC data added to a TCP-friendly MPEG video stream. The project utilizes a custom-written TCP-friendly rate control algorithm which features some of the functionality of Datagram Congestion Control Protocol (DCCP) to ensure a TCP-friendly video data rate.

Our system is a set of specialized tools for Linux consisting of three major components working together: MPEG video management, adaptive FEC, and network rate control / quality-of-service measurement. The MPEG component consists of a player and an encoder that captures from a live video source or encodes from a file to produce an MPEG video stream. The MPEG component on the receiving end of the transmission has the job of dropping unplayable frames from the stream. The FEC component utilizes an algorithm (AFEC) that considers network conditions to apply the optimal amount of FEC packets to each video frame. Additionally, it applies temporal scaling to the stream to remove the least important frames, thereby reducing the data rate when necessary. The quality-of-service measurement provides packet loss probability and delay time to the FEC component while the network rate control component transmits the video stream at a TCP-friendly rate. The components are designed to work in real-time such that the system could be used for videoconferencing.

The end product will be utilized by Huahui Wu, the creator of the AFEC algorithm, for further research, performance analysis, and enhancement. We also hope that success of this project will spark additional research in the computer science

community as a whole, especially since a product like this had not yet been developed. In time, it could lead to the adoption of adaptive FEC in more mainstream video applications. This software has been designed to be as modular and reusable as possible, since each component can have a wide array of uses.

The following chapter describes each of the technologies used to implement our system. Chapter 3 provides a detailed description of the design and implementation of our system. Chapter 4 discusses the methods we used to test our system as well as performance test results. Chapters 5 and 6 state conclusions and suggest ways in which our system could be modified or extended.

2. Background and Related Work

2.1 MPEG Overview

Digital video contains a massive amount of data, on the order of tens of megabytes per second for uncompressed video. To transmit video over communications networks, a compression scheme is necessary. The MPEG (Moving Picture Experts Group) video compression standard was developed in 1992 to provide VHS-quality video at low bitrates around 1.5Mbps.

MPEG-1 uses several techniques to reduce spatial and temporal redundancy that achieve up to 100:1 compression ratios at an acceptable quality. First, the 24-bit video, which normally uses the RGB color space (in which discrete values for each of the red, green, and blue components in a color are defined), is converted to the YUV color space (in which luminance and chrominance are separated. Also known as 'YCbCr'.). Since the human eye is more sensitive to luminance, the chrominance is subsampled with surrounding pixels to achieve a 4:1 compression ratio. The resulting frames are then

divided into 16x16 macroblocks and 8x8 blocks. Each frame is encoded as either an I (intraframe), P (forward predicted), or B (bidirectional predicted) frame [2]. I frames are encoded similar to JPEG. P frames are encoded relative to the last reference frame, either an I frame or a previous P frame. Motion vectors are used to determine how the macroblocks have moved relative to the reference frame. B frames are encoded relative to the past reference frame as well as the next reference frame. The repeating pattern of frames is called a Group of Pictures (GOP). Because of the relationships between frames, there are certain frame dependencies. For example, if an I frame is not received, none of the B or P frames can be played because they rely on the I frame as a starting point. The following figure illustrates the dependencies of each frame. The AFEC system utilizes MPEG-1.

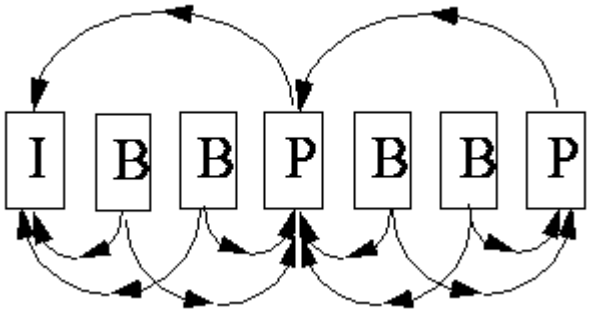

Figure 1 – Dependencies of a GOP

The MPEG-2 standard was developed in 1994 and is intended for high-quality video at higher bit rates of around 15Mbits/s. It also supports five audio channels and interlaced video, which is used in the DVD standard.

MPEG encoding is very CPU-intensive such that at the time it was developed, encoding was a long process that only achieved 1- 3 frames per second on an average PC [3]. Since then, there have been significant speed increases in computer hardware as well as the development of highly optimized code. Full-screen raw video can now be encoded

at full frame rate in real time on modern hardware, which helps to make possible the goal of live video conferencing.

ffMPEG [4] is a relatively new software package that records, converts and streams audio and video. It was developed under Linux, and consists of three programs: *ffmpeg*, *ffserver*, and *ffplay*. *Ffmpeg* is a command-line tool that can re-encode many video formats. It can also capture and encode video in real time from a TV card or other Video4Linux-compatible video source. *Ffserver* is an HTTP streaming server for live video broadcasts. *Ffplay* is a simple no-frills media player that can play all of the formats *ffmpeg* can encode. The ffMPEG package uses *libavcodec*, a leading codec library, to encode in a wide range of formats with high performance.

2.2 Available Congestion Control Implementations

2.2.1 Jörg Widmer TCP-Friendly Rate-Control (TFRC) Implementation

As the oldest and most basic of these implementations, the Widmer code **[**5**]** attempted to provide functional rate control by delaying the transmission of UDP packets until they would not violate an arrived-at TCP-friendly sending rate. The distribution package was disorganized, which initially made building the code difficult. By the time the AFEC system required integration of a DCCP (Datagram Congestion Control Protocol) element, progress had been made on congestion-control protocols in general. Although the Widmer code was designed in reference to the TFRC (TCP-Friendly Rate Control) Request-For-Comments (RFC #3448), other congestion-control code packages seemed to be more viable as options.

2.2.2 User-level Berkeley Implementation

This user-level code [6], written by a team from the University of California at Berkeley, attempted to implement a code library that could be dynamically linked to programs that required this type of congestion control. The notion of a user-level DCCP implementation was initially appealing because it meant that any subsequent developers would not be required to run a custom kernel (which would require additional setup time and could lead to hardware-related complications.) Unfortunately, this implementation was rather problematic in four important respects:

- Large sections of code were commented out without any explanation as to why this was done. The code also suffered from having generally poor style and lack of descriptive comments, which reduced confidence in its robustness.
- Building the DCCP library was difficult due to dependency on a troublesome external network data-collection and timing library, support for which was minimal and updates for which are apparently infrequent.
- The output of the included test scripts (apparently used to validate the functioning of the installation) was undocumented and generally unclear.
- Some of the included test scripts failed for unknown reasons. This was also unexplained.

2.2.3 Patch Against BSD Kernel

The design of integrating DCCP functionality into the operating system kernel [7] alongside its sister protocol, UDP, was attractive as it abstracted away the technical operation of the network control, allowing the user-level applications to pursue their

original purpose of efficiently transmitting video frames. Programs could merely specify DCCP's assigned protocol number when setting up a network socket. However, because Linux was chosen as the development platform, a patch against the BSD kernel would not be directly useful.

2.2.4 Patch Against 2.4.18 Linux Kernel

The Linux kernel patch [8] has the same benefits of the patch against the BSD kernel, but was, of course, much more useful given the choice of implementation platform. The latest addition to the available archive of kernel patches was dated May 30, 2002 and was the version chosen for AFEC due to the additional fine-tuning of the code that had presumably taken place since earlier versions.

These various options for the network congestion-control segment of the system were investigated. Of the three main possibilities, the DCCP-protocol Linux kernel patch was selected for integration because of its various advantages over the rest. Perhaps the most important difference between the available options is the level at which the code runs (user-level or kernel-level). The advantage of user-level code is that no superuser privileges are required to set up and run the protocol, and the advantages to kernel-level code are that the protocol can assume its "proper" place among its sister network protocols in the kernel, and that it remains unfettered by permissions restrictions. (Note that no code or packages discussed herein should be assumed to be secure. Alternative security measures should of course be taken if exposing experimental congestion-control sockets to a non-private network.)

After obtaining and uncompressing the May 30, 2002 diff file from `http://www.ducksong.com:81/dccp/patches/`, a 2.4.18 kernel was patched successfully. However, 2.4.18 lacks support for newer hardware, most important of which were the various IDE controllers present on development machines. Specifically, without support for the HighPoint HPT374 and Promise RAID chipsets, any 2.4.18 kernel built would be unbootable on the available development hardware. Consequently, the decision was made to switch to 2.4.24 and adapt the kernel patch to that version.

With DCCP support successfully built into the kernel, user-level programs can take advantage of it as a drop-in replacement for UDP by replacing `SOCK_DGRAM` references with `SOCK_DCCP` and including the following definitions:

```
#define SOCK_DCCP                   6
#define SOL_DCCP                    269
#define SO_HANDSHAKE_DATA           1
#define SO_ALLOW_HANDSHAKE_DATA     2

struct sockaddr_dcp
{
      struct sockaddr_in in;
      unsigned int service;
};
```

Figure 2 - Example code for DCCP sockets

After the basic DCCP support was included in the kernel, modifications were made (entirely within `net/ipv4/dcp.c`) to create a proc-filesystem entry "`dccp`". Within this entry, two proc files ("`lossrate`" and "`rtt`") reflected the most-recently-calculated network loss rate and round-trip time on open DCCP sockets. This was required for the AFEC algorithm to determine the optimum FEC level to add. A more robust implementation would have involved the use of `setsockopt()` and `getsockopt()`. (This is suggested for future development, as numerous DCCP streams may be open on a given

kernel at any time. With a proc-filesystem solution, a program would not be able to determine to which stream a calculated value was relevant.)

Unfortunately, after undergoing testing, it was shown that a DCCP-enabled kernel was extremely unstable during `SOCK_DCCP`-type socket communications. Usually within 50-60KB being transferred, the kernel running on the sending side of the transmission would lock. This, of course, caused the transfer to stall. In approximately half of all tests, the receiving kernel would then also lock. It is suspected that this is due to an overflow scenario within the `dcp.o` kernel module.

In an attempt to resolve this issue, hardware supported by the 2.4.18 kernel was acquired, and a DCCP-enabled 2.4.18 kernel was booted – the theory being, that the kernel originally intended to receive the DCCP patches would behave as expected. Communication with the project maintainer and original author, Patrick McManus, suggested that this DCCP implementation had been shown to work successfully and actually interoperate with other early and experimental DCCP or TFRC implementations. The hard-lock issue persisted, which prompted the testing of both 2.4.18 and 2.4.24 kernels. The kernel versions were tested both modified and unmodified (for producing a proc-filesystem entry), with DCCP built-in, and with DCCP running as a module. These kernels were also tested in a variety of configurations, and running on a variety of hardware. These attempts were unsuccessful, and, in the interest of time and simplicity, the decision was made to implement a UDP-based module in place of the original planned DCCP module. (In improving the functioning of this AFEC system, priority should be given to identifying and resolving the issue discussed above, which is presumably the result of a bug inside `net/ipv4/dcp.c`.)

2.3 Forward Error Correction

Automatic Repeat reQuest (ARQ) is one method of transmitting data losslessly. Using this technique, when data is lost, the receiver waits until it is very unlikely that the data will come in, and then re-transmits it. ARQ can provide data integrity but this comes at the cost of latency. The receiver first has to wait a "timeout" period and then the latency to and from the sender. If latency and the timeout are too high, ARQ may become very costly or even useless for time-critical data like streaming video. However, for most general cases, ARQ is sufficient.

Forward error correction (FEC) is another technique that attempts to transmit data losslessly. FEC adds redundant data to a stream of data that is likely to become corrupted through transmission. If there is enough redundant data included in the stream, the corrupted data can be recovered. Some applications of FEC are data recovery in error-corrective memory chips for high-end server computers, data recovery from scratches on compact discs, and data transmissions to and from spacecraft [9]. Forward error correction is most useful when it is either impossible to retransmit data (such as with scratches on a compact disc) or when it is very time consuming to retransmit data (such as when sending images to Earth from a Mars rover.) The goals of forward error correction are to provide data integrity and to reduce the time it takes to receive data by removing latency from the transmission process.

Next, we will discuss some primitive and naïve forms of forward error correction, their good points, and their bad points.

2.3.1 Simple Forward Error Correction Techniques

In the examples below, we will discuss how a Mars rover might transmit image data from Mars to Earth. Transmissions through space are lossy – not all data transmitted will be received correctly. There is also a lot of latency (about 15 minutes) between Mars and Earth. Requesting retransmissions is very costly since it will take about 30 minutes round trip time (RTT) to arrive after the initial request.

The simplest type of FEC is simply transmitting data more than once. For example, a Mars rover might transmit each photograph it takes three times. Any missing or corrupted data is correctable using a "2-out-of-3 method." If the data in the first two copies does not match, the third copy is consulted. Because of the binary nature of bits, there are only two choices, so the data that is common in at least two of the three streams will be used because it is statistically more likely to be correct.

```
Byte stream1[] // the first copy
Byte stream2[] // the second copy
Byte stream3[] // the third copy
Byte output[] // the composite image being reconstructed on Earth

for i=0 to stream_length do // for each byte in the transmission
      if stream1[i] == stream2[i] // if 1 and 2 match
            output[i]=stream1[i]
      else if stream1[i] == stream3[i] // if 1 and 3 match
            output[i]=stream1[i]
      else if stream2[i] == stream3[i] // if 2 and 3 match
            output[i]=stream3[i]
```

Figure 3 – Combining repeated streams

The problems with this technique are that it is very inefficient and that the data integrity check is very rudimentary. It is inefficient because it is unlikely that all data will be lost. Therefore, it is unnecessary to transmit all data three times. It would be much better if it were possible to transmit only enough data such that the expected number of errors could be corrected. The problem of data integrity in this technique is that there is

no good way of verifying which data is the correct data. If the first bit of the picture is 0 in the first and third transmissions and 1 in the second transmission, it is more likely that the correct value is 0, but there is no way of verifying this.

Various techniques such as cyclic redundancy checks (CRC) or hashes (such as SHA-1 or MD5) can be used to add data integrity to the transmission. If a hash is used, when Earth receives a picture from the Mars rover, Earth will compute the hash over the data it received and then compare it with the transmitted hash. If they match, there is an extremely good chance that the data was transmitted without error. Although this technique decreases the likelihood of the data integrity problem, it worsens the efficiency problem. With this technique, we know that at least one bit is incorrect, but there is no way to tell *which* bit it is. If none of the three transmissions come through 100% correctly, all data must be thrown away and ARQ must be used. Since ARQ is what we are trying to avoid by using FEC, this is not a good solution.

A modification to this technique would be to break up each picture into many small fragments and generate a hash for each fragment. In this case, when a retransmission is needed, it is for a small fragment of data rather than a large picture. This will cut down on the total amount of data transmitted between the Mars rover and Earth, but it does not do much to solve the latency problem.

The techniques described above help to mitigate the problems of transmitting streams of data from Mars to Earth, but none of them solve the problem completely. The "2-out-of-3 method" decreases latency between Mars and Earth by not re-requesting information, but it does very little to combat data corruption, and is also inefficient since at least three copies of data must be transmitted. The hash method guarantees data

integrity, but does nothing to solve the latency problems between Mars and Earth. ARQ in the transmissions between Mars and Earth becomes a problem because the Mars rover will need to remain powered on for a long period of time after it has finished each transmission. Narrow time-windows when data can be sent to and from Earth would make transmissions incredibly slow.

The transmission technique used in AFEC needed to guarantee that ARQ was not needed and that data did not become corrupted. It is important that ARQ is not needed because of the tight timing constraints of real time video. Since most video is played at a rate of about 30 frames per second, there is only 1/30 = 33ms between video frames. On most networks, this is not enough time to timeout, request a retransmission, and wait for it to come in. Data can be buffered for a short amount of time in order to decrease the effects of latency, but for video conferencing, this will create a noticeable delay. In order to avoid buffering, extra information must be transmitted that allows missing data to be reconstructed. Since data may become corrupted along the way, it is also important that corrupted information is identifiable. Reed Solomon codes provide all of these features and are what we decided to use for AFEC.

2.3.2 Reed Solomon Erasure Codes

Reed Solomon codes are similar to a hash in that they will look through the data to be encoded and generate a set of codewords from it. In the case of a hash, these codewords can be used to determine if the data has become corrupted during transmission but can do nothing to tell where the problem is or how to fix it. By using Reed Solomon codewords, in addition to verification of data integrity, lost data can be reconstructed.

Reed Solomon codes are also much more 'adjustable' than the "two-out-of-three method." With the two-out-of-three method, all data must be transmitted three times. Reed Solomon codes allow some pieces of data to be better protected than other, less important pieces of data. For example, MPEG I frames are more important than B or P frames since an I frame must be received in order for B or P frames to be played correctly (see Figure 1). Therefore, it makes sense to add extra protection (more redundancy) to I frames.

Reed Solomon codes are a type of block-based error correcting code. Block-based error correcting codes use a notation of (N, K) where N is the number of codewords and K is the amount of data that the code will protect. Normal block-based error correcting codes can correct up to (N-K)/2 errors. If it is known where errors are located, the code is called an erasure code. An erasure code can correct up to N-K errors. For example, the erasure code (255, 239) can correct up to 255-239=16 while a normal block-based error correcting code with the same amount of redundancy can only correct 8 of the errors. In this project, we deal exclusively with lost data rather than corrupted data. Since we can determine when we lose packets through gaps in sequence numbers, we know the location of missing data and we can use an erasure code.

Two different types of block-based error correcting codes are 'short' and 'long' codes. Short codes are suitable for streams of dynamic data like video conferencing where data must be decoded on-the-fly while long codes are more suited to distribution of static data. Applications such as Digital Fountain [10] or PAR (parity archives) make use of long codes. To demonstrate the difference between these types of codes, consider the distribution of a large (700MB) movie over the Internet. A small block code would divide

the movie into many small blocks. Each block would contain redundant data that corresponds only to the original data in that block. If a piece of data in a block was lost during transmission, it could be recovered provided that data very near to it in this block was received. A large block code would also divide the movie into many blocks, but these would usually be much larger (megabytes as opposed to bytes). Each of these blocks will contain purely original data and no redundancy (A, B, and C in Figure 4). In addition to these blocks will be a number of blocks that contain only redundant data (F1 – F4 in Figure 4). This redundant data, when combined with received data, can correct data evenly spread out across the whole video file. If a block of data is lost, a certain number of any of the redundant blocks can be used to reconstruct the lost data.

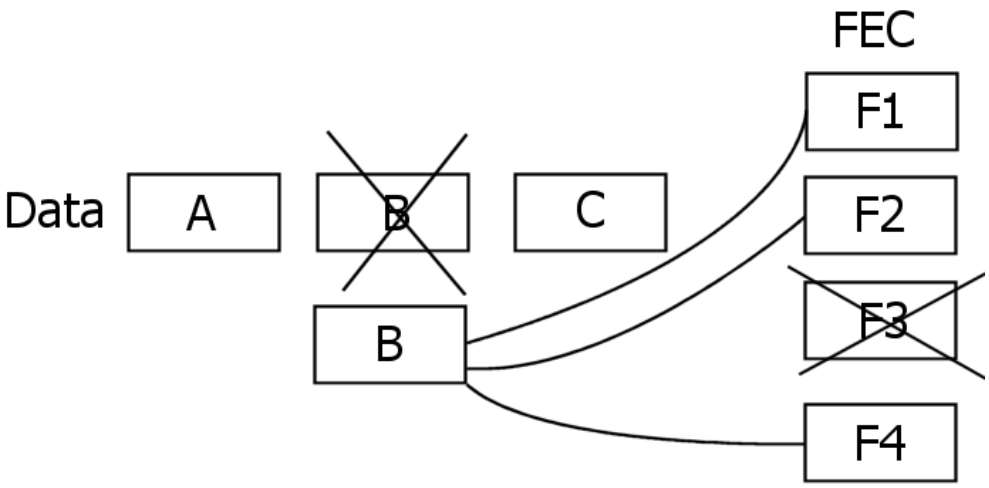

Figure 4 – Reconstruction with long FEC codes

While large block codes would be useful for recovery of a large video file being multicast on a system such as a broadcast disk [11], they are not useful for a real-time streaming video. This is because in order to create long codes, all the data must be known in advance. In addition to this, streaming video needs to be played as it is downloaded.

Streaming video needs to arrive in order so that it is playable in realtime. Short block codes are able to provide FEC as it is needed and allow realtime error correction.

2.3.3 Adaptive Forward Error Correction

TCP is one of the most popular transport layer protocols for a number of reasons. TCP guarantees that all data sent is received correctly and in the same order it was sent. It does this through ARQ. Additionally, TCP uses a sliding window which varies the rate at which it sends data based on network conditions. When a connection is first made to a remote computer, TCP's congestion window starts at two packets. This means that it must wait for acknowledgements after sending only two packets. If network conditions are good, TCP will increase the size of its window and start to send data at a faster rate, pipelining sends and receipts of acknowledgements. If network conditions worsen, TCP uses multiplicative decrease to halve its window size.

While retransmission is very good for the transfer of files which must be received losslessly such as executable files, streaming video has different QOS requirements. For streaming video, it is important that as much of the original data as possible is received, but it is even more important that new data is received in a timely fashion. Because of the tight timing constraints of streaming video, it is often not possible to wait for a retransmission. For example, NTSC video is encoded at 29.97 frames per second. This means that there is a frame played every 33ms. It is unrealistic over most connections to expect that a frame can be requested and retransmitted in less than 33ms. In Figure 5, it is detected that frame 3 was lost in transmission so it is re-requested. By the time frame 3 arrives, frame 4 has already been played, so frame 3 must be thrown away. While

intelligent buffering can help, buffering increases latency on the receiving node. In some applications such as streaming of pre-recorded videos, buffering a few seconds of video is a good solution. However, this is not a good solution for applications such as video conferencing where increased latency on the receiving node creates a noticeable end to end delay. In these types of applications, it is often better to sacrifice quality rather than increase end to end delay.

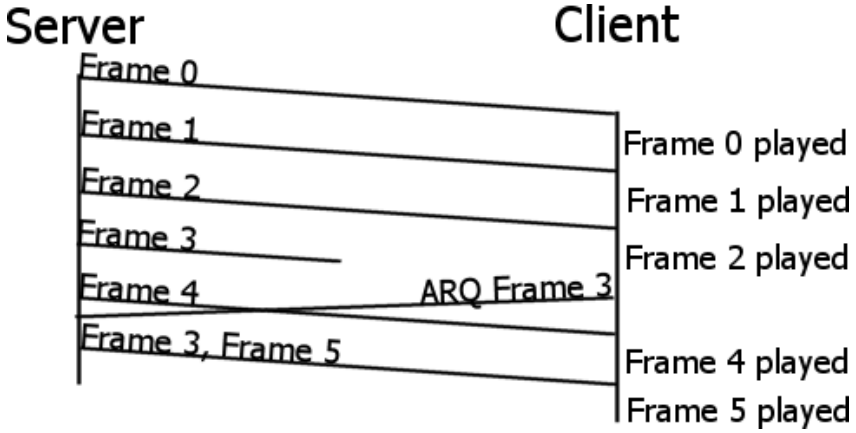

Figure 5 – Problems with ARQ and realtime video

Many current streaming video solutions use "best effort" protocols such as UDP. UDP does not automatically request retransmission of lost packets because it has no mechanism for detecting packet loss. UDP is unresponsive to packet loss and congestion so already congested routers can be flooded with lots of undeliverable traffic. TCP Friendly Rate Control (TFRC) provides the rate control of TCP while allowing application-level control over retransmission. Since retransmissions are undesirable for streaming video, forward error correction can be used to provide data integrity. The one remaining problem is, given the TCP-friendly rate, to determine the optimal amount of FEC to add to the data. If too much FEC is added, it is wasted – a higher quality video could have been sent in its place. If too little FEC is added, there will not be enough

redundancy to repair the lost data. Since it is an all-or-nothing situation, this is also a waste. Wu's algorithm helps in this regard by predicting the optimal amount of FEC based on network conditions, which allows us to achieve the highest throughput of usable data.

2.3.4 Existing FEC Solutions

While researching forward error correction, a number of existing FEC implementations were uncovered. Some were applicable to AFEC while others gave us a broader understanding of FEC in general. This section discusses the merits of the most influential papers and implementations.

2.3.4.1 Luigi Rizzo

Luigi Rizzo of the University of Pisa has written two very useful papers [12, 13] on his implementation of a Vandermonde matrix based Reed Solomon erasure code. Both papers offer a very good overview of forward error correction and offer some insight into the mathematics behind his implementation. Perhaps more useful is Rizzo's implementation of Reed Solomon erasure codes. While Rizzo's code is written in a fairly clean style, it is still very difficult to follow since there are few comments. We were unable to determine if changing the amount of FEC was possible. Since adjusting the amount of FEC on a packet-by-packet basis is a central idea of AFEC, we decided against using Rizzo's code.

2.3.4.2 Michael Luby

Michael Luby of Digital Fountain is the leading contributor to many papers [10,14] about large block codes, some of which lay the foundation for Digital Fountain's products. Although we have shown that large block codes are not useful for AFEC, Luby's papers served as useful background on FEC and the differences between short and large block codes. No implementations of Luby's algorithms could be found, perhaps because they are in use commercially.

2.3.4.3 Phil Karn

Although no FEC-related papers written by Phil Karn of Qualcomm Inc were found, he has been cited as a helpful source of information by many papers we read. Moreover, he has written, and made freely available, a very flexible implementation of Reed Solomon based erasure codes [15]. The flexibility of Karn's implementation led to our choice of incorporating it into AFEC.

3. System Implementation

This section explains the implementation of the AFEC system. Each software component is discussed in detail, as well as the design decisions made during the development process.

3.1 System Overview

The three main components required were the MPEG processor, FEC encoder/decoder, and rate-controlled network sender / quality-of-service measurement. To improve flexibility, ease testing, and facilitate future development, the components were designed to interoperate by passing data on STDOUT and receiving on STDIN. In this way, they can be chained together on the command line using pipes and redirection operators. Also, the simple communication protocol between the components allows for one or more components in the chain to be replaced without the others requiring any changes. The following diagram shows all of the components in our system. Each block represents an executable program, each of which is explained in detail in this section.

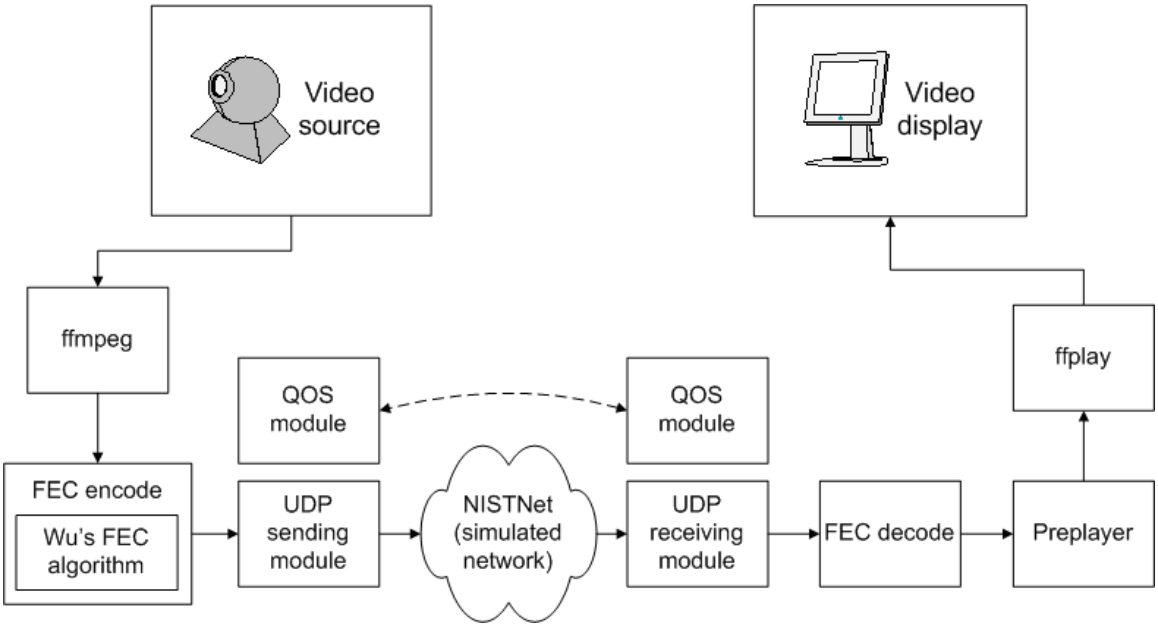

Figure 6 - Block diagram showing data flow between components

The following diagram shows how MPEG video passes through our system. It shows the data format at each step of the data flow in Figure 6. Again, this will be explained in more detail in this section.

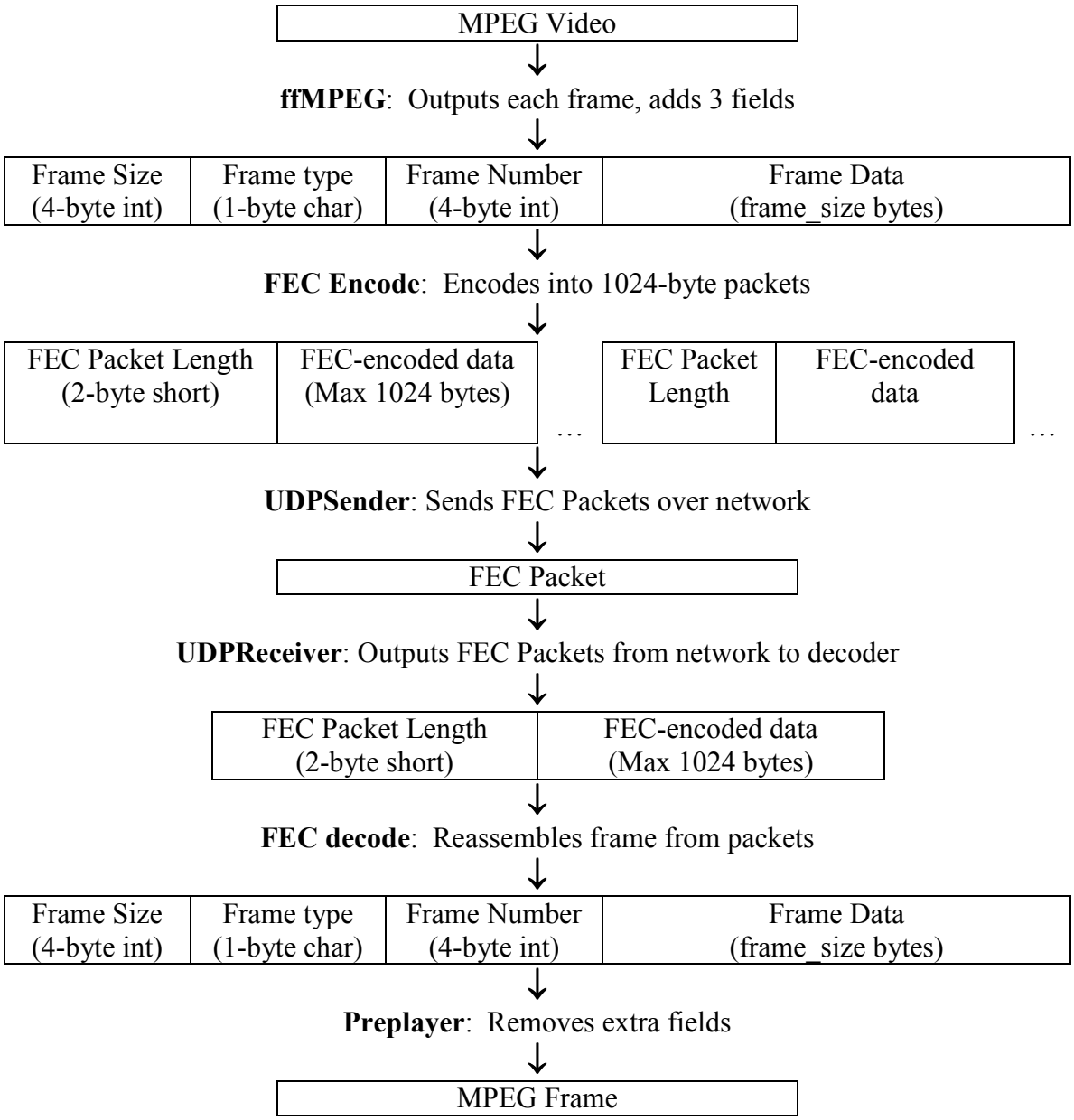

Figure 7 - Data flow packet diagram

3.2 MPEG Component

When deciding on the video component for our system, a major concern was about the ability to capture video and encode it in real time. The Berkeley MPEG tools, ffMPEG, and the codec from MPEG Software Simulation Group were all considered as solutions. The MPEGSSG codec was ruled out because it lacked a player for decoded video and the code hadn't been updated in the good part of a decade. While the Berkeley MPEG tools are popular for academic research, it also contains old code, and required some modification to compile in Linux. Also, encoding performance is questionable and might not be suitable for a real time application.

The current version of ffMPEG (0.4.8) proved to be a perfect match for our system. ffMPEG solves all of the problems associated with the other software we tried. It is a currently active project with a versatile command-line encoder and minimalist player. It also supports video capture with real time encoding, providing in a single program two critical elements to this project. The video capture can use any Video4Linux-compatible device. The test setup uses it with an ATI TV Wonder card, which has the popular BT878 (Brooktree) chipset. On a 1600Mhz Athlon XP, real time capture and encoding to MPEG-1 utilizes only 20-30% of the CPU. This leaves plenty of CPU time for the other components.

3.2.1 ffMPEG Modification

The AFEC algorithm requires GOP information, including the type and size of each frame as well as the GOP size. This information is contained within an MPEG stream, and the goal was to pass this information along to the algorithm in a simple and efficient manner. Initially, an attempt was made using the *mpeg-stat* tool from Berkeley

MPEG tools. Mpeg-stat reads in an MPEG stream, processes it, and logs all of the stream information. The idea was to pipe the output of the ffmpeg encoder through mpeg-stat, to extract each frame's information and pass it along with the frame to the AFEC component. However, the mpeg-stat code was difficult to modify, and writing custom software to parse an MPEG stream was judged to be too difficult for the scope of this project.

Instead of adding a component, a better solution would be to modify the *ffmpeg* source. In the portion of the code where ffmpeg handles an encoded video frame received from the codec, some code was inserted to write a custom stream out to a separate file. The modified ffmpeg encoder adds the frame size, type, and number before each frame. (Figure 6) The modified output file is actually a named pipe, which the AFEC component can read in real time while the encoder is running. The reason the default output of the encoder was left unchanged is that it is handled mostly by the codec, and it would be unwise to modify the codec as well. So, when using ffmpeg for the system, unmodified MPEG output is simply redirected to /dev/null.

A shell script invokes the encoder to either capture from a video source, or encode a specified video file. In either case, encoding is processed at the native input frame rate so that the video will be streamed and played back at the proper speed. The default output video size is 320x240 with a default bit rate of 1.5Mbps is used, and a size 12 GOP having the pattern IBBPBBPBBPBB. These parameters can be easily modified by changing the command-line arguments that the shell script passes to the encoder. Changing the GOP pattern requires the additional step of modifying GOP values in the

other components. When initiated, the encoder waits until the AFEC module opens the other end of the pipe, effectively synchronizing the operation.

On the receiving end of the connection, the modified MPEG stream is output frame-by-frame from the FEC decoder module. This cannot be fed straight to the player because of the extra headers added, so they must be stripped off. Some frames may have been lost despite AFEC's efforts to save them. A frame is deemed lost if it either did not make it through the network cloud or the FEC decoder was unable to recover it. The MPEG component does not require notification of a lost frame, since frame numbers are tracked. Some B and P frames may be dependent on a lost I or P reference frame, and thus cannot be played. Such frames must be thrown away.

A *preplayer* module that works between the FEC decoder and *ffplay* was developed, allowing us to solve these problems without modifying the player. The preplayer strips off the size, type, and frame number data added, and calculates whether each frame can be played. If a frame is dependent on a reference frame that has been lost, the preplayer simply drops the frame and does not pass it on to the player. Ffplay has the ability to play all frames regardless of missing reference frames. Of course, this results in poor quality because the video looks incorrect when a frame is played without its reference frames, and MPEG was not designed to behave in this way. When the preplayer drops dependent frames, ffplay still handles the stream properly, waiting for the next frame.

To aid in debugging and performance evaluation, the preplayer also performs comprehensive custom logging that is separate from any MPEG stream statistics provided by the ffMPEG package. It calculates the playable framerate, percentage of

playable frames, number of missing frames, and number of frames of each type dropped due to missing reference frames. All timing used in the logs is calculated using the *gettimeofday()* function, which on most systems has a resolution of at least 1/100 second. The logfile contains the number, size, and type of each frame received, as well as notification of a dropped frame. When the program terminates, it writes a summary of the stream.

3.3 Network Measurement and Congestion Control

3.3.1 Congestion Control Integration

The first UDP-based module, developed for testing purposes and simpler than the equivalent DCCP module, took several command-line parameters as follows:

- IP address of the machine that will be receiving the transmission
- Round-trip time in milliseconds
- TCP retransmit timeout in milliseconds
- Packet-loss probability

These parameters remained static during operation, contrary to the functioning of the DCCP module (in which dynamic network measurement would update them.) Using these parameters and the mathematical model developed to calculate the TCP-friendly sending rate, the UDP module added a calculated amount of delay before sending each packet, such that the average data rate would not exceed the TCP-friendly rate for the specified network conditions. When running, this module received data on STDIN (where the first two bytes contained a C-style `short` representing the number of bytes in the packet to follow, as shown in Figure 6), and passed data to its sister UDP module

waiting on the receiving end of the connection. The receiving module, in turn, output the data on STDOUT (where the receiving-end FEC module would read and decode it.)

Because of its inflexibility and lack of faithfulness to the original DCCP integration plan, it was decided that this static-state UDP module would be replaced by a custom-written module which, although still communicating using UDP, was able to actively measure network capabilities and continuously update the relevant parameters (round-trip time, TCP retransmit timeout, and packet-loss probability). This is done by continuously sending "control" packets to the receiver. If the receiver echoes these packets back within a reasonable amount of time, the RTT and instantaneous packet loss (0%) is recorded in a running average. If the packet never comes back, it is recorded as lost (100% instantaneous packet loss) in the running average. As in Wu's paper [1], RTO is calculated as 4 * RTT. Work on this module was completed and the system was successfully tested. Error-correction data was effectively used and tweaked as the NIST Net simulated network dropped and delayed packets at varying rates. The format of the data that this UDP module receives and outputs is identical to that of its predecessor.

3.4 FEC Implementation

3.4.1 Packetizing the Data

Although it was the best option available, Karn's FEC implementation was not exactly what was needed for AFEC. While Karn's implementation of Reed Solomon erasure codes was complete, it is not extremely straight forward how to use the codes to guarantee better data delivery.

A naïve approach to sending data and its corresponding FEC over the network would be to send, in a streaming fashion, data followed directly by its corresponding redundancy. This could be done in a number of ways as illustrated by Figure 9 below. The problems arise because of the way that most networks like the Internet work. Instead of transferring data a byte at a time, the smallest piece of data receivable by the AFEC application is a packet. Because of the way lower-level network protocols work, if any data is lost or becomes corrupted during transmission, the entire packet that the data was in must be thrown out. Because of this limitation of the lower network-layers, if data along with its redundancy is transmitted across the network in a single packet, the redundancy can never be useful. If a packet is lost, the redundancy that was supposed to rebuild that data is also lost. A slightly better idea is to send data in one packet followed by a packet of redundant information. While it is less likely that both data and redundant information will be lost, if the data packet is lost, it will take a same-size packet of redundant information to "reconstruct" the original data. As shown in Figure 8, if packet 1 is dropped in Method 1, no amount of redundancy in packet 1 can recover its data because it is also lost. Method 2 is inefficient because it takes a whole packet of redundancy to correct a lost packet of data.

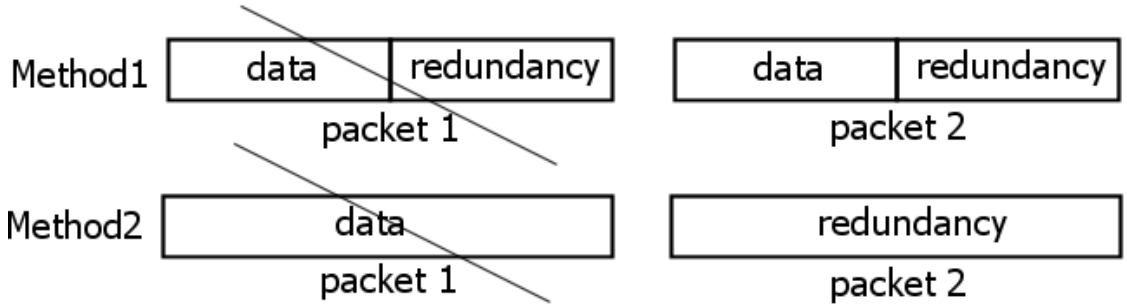

Figure 8 – Packetizing data

This is not the desired behavior. AFEC must be able to control exactly how much redundancy is included with its corresponding data. With the problem described above, the best option is to include an equally sized redundancy packet for each data packet sent. Although it is not explained in detail, Rizzo mentions [12] that he uses an algorithm to evenly distribute data across multiple packets in order to avoid such a problem but does not go into detail. Unfortunately, Karn's code did not do anything like this, so it had to be added in our implementation.

The solution was to break the stream up into *sequences* of packets. It was decided there would be a sequence for each MPEG frame. When a frame is being processed before being sent over the network, it is broken up into at least P_{SMax} packets which is the maximum of P_S and the number of packets required to fit all the data. P_S is the minimum number of packets with which data can be recovered in the proportion expected by the Reed Solomon code in use. In the equation below, B_{Data} / $B_{Redundancy}$ is equivalent to the FEC code's `N / (N - K)`. P_S is calculated in Figure 9 below.

P_S = ceil(L * (B_{Data} / $B_{Redundancy}$))

P_S: Number of packets in the sequence
L: the number of expected lost packets
B_{Data}: bytes of data in the sequence
$B_{Redundancy}$: bytes of redundancy in the sequence

Figure 9 – Calculation of P_S

If the size of the frame is larger than the maximum size of the packet (1024 bytes currently) times P_S, then the number of packets in the sequence is the number of packets required to hold all frame data.

Once the number of packets in a sequence (P_{SMax}) is known, data can be distributed between them. This is done in such a way that if any packet is lost, data loss will be distributed evenly throughout the sequence. If this it the case, then the redundant data can correct errors because there will be only up to 1 / P_{SMax} of each piece of the sequence missing for each packet lost. The algorithm to distribute data is below.

```
Packet orig_packets[P_SMax] // original packets
Packet dist_packets[P_SMax] // distributed packets (which
                            // we're creating)

c=0
for p from 0 to P_SMax
  for i from 0 to original_packets[i].length
    dist_packets[c mod P_SMax][c/P_SMax] = orig_packets[p][i]
    c=c+1
```

Figure 10 – FEC distribution algorithm

Once data has been distributed amongst the packets with the above algorithm, the data is sent over the network. On the other end, the data is put back into the original order. If any packets were lost (detected by gaps in a sequence number), this information is noted and given to the Reed Solomon decoder later as erasure data. Since the data was evenly distributed all through the sequence, any lost information will also be distributed evenly through the sequence. By assuring that the sequence length is at least P_{SMax}

packets long, there will be enough redundancy to correct at least L missing packets with the error correcting code in use.

3.4.2 Integrating Wu's Algorithm

Wu's algorithm is the most important part of AFEC. It computes the optimal amount of FEC to add for each frame of an MPEG stream based on RTT, P, and the sizes of the MPEG frames. An attempt at a TFRC UDP client and server along with a QOS monitoring program were implemented. These two sets of programs allowed us to update RTT and P with a reasonable level of accuracy in realtime and gave us the ability to pass this information on to Wu's algorithm.

How to best describe to Wu's algorithm the size of the frames is an issue that remains unresolved. Wu's algorithm requests the size of frames based on the number of packets they take up. Because of the enforcement of a minimum number of packets per sequence (P_{SMax}), it is often the case that most frames will use the same number of packets. Consider a high motion video where I frames are 16KB, P frames are 10KB, and B frames are 5KB. Also consider that the FEC codec currently in use is (255, 239) (a code that can correct up to 16 erasures). If P_{SMax} is computed for packets with a maximum size of 1KB, it can be shown that a sequence for an I, P, or B frame will be 16 packets long. Each of the I frame's packets will be completely full, the B frame's packets 5/8 full and the B frame's packets only 5/16 full. These are very different amounts of data being sent across the network, but the number of packets is the same. Because of this inconsistency, it was unclear what should be given to Wu's algorithm. In place of the true size of each frame, standard sizes as presented in Wu's paper were used. In the

implementation, constant values of 25 packets, 8 packets, and 3 packets are assigned for I, P, and B frames respectively. In the future, this should be worked out. This should be straightforward, as separate functions are provided for computing the size of I, P, and B frames. The solution can be implemented without having to dig through other code.

Despite this problem, Wu's algorithm adjusts the amount of FEC in an intelligent way based only on RTT, P, and the expected ratio of I, P, and B frame sizes. Although the algorithm does not compute the optimal amount of FEC to add because of the missing information, the results of our tests indicate that if the I, P, and B frame sizes were specified correctly, Wu's algorithm would work very well.

4. Testing and Performance Evaluation

4.1 Testbed Setup

This section describes the hardware used to develop, debug, and test the AFEC system. The 'server' is the system that performs the video encoding, adds FEC, temporally scales the video, and sends the data over the network. The 'client' receives, decodes, repairs or discards broken frames, and plays the video. The following component list is provided as a basis for comparison, since the AFEC system will likely be used on different PC hardware in the future.

Server Specifications
AMD Athlon XP 1900+
1GB PC2100 DDR RAM
ATI TV Wonder PCI (BT878 Brooktree chipset)
Realtek 8139 10/100 NIC

Client Specifications
Dual AMD Athlon MP 1800+
512MB PC2100 DDR RAM
NVIDIA Geforce 2
3Com SOHO 10/100 NIC

Both systems currently run Gentoo Linux 1.4 with kernel version 2.6.5. (Version 2.6 or later of the kernel is actually required on the sending side of the system, as the network measurement code requires a high-resolution timer based on the CPU clock.) The FEC and MPEG components were developed using kernel version 2.4.24, so they are tested to work in that and any later versions. The networking component was developed entirely in the 2.6 series. The 2.4 kernel may not have the timing capabilities required to run the UDPSender, so it is recommended that the AFEC system should be used with the 2.6 series kernel.

4.1.1 NIST Net

A local area network does not experience enough delay or packet loss to perform controlled tests in which high levels of delay and packet loss are required to prove proper functionality. For this reason, a Linux PC router running the NIST Net[16] software package was used.

NIST Net is a network emulator tool that allows controlled, reproducible experiments to be performed. It operates at the IP level and is implemented as a kernel module extension to Linux. When installed on a Linux router, it allows for network traffic passing through the router to be manipulated. For this project, the main concern regards delay (round-trip-time), capacity, and packet drop percentage, although NIST Net provides many other options.

Our Linux router is a Pentium II 450 MHz with 128MB RAM running a fresh installation of Red Hat Linux 8.0. The system runs the 2.4.18-14 kernel. The only difference from the default kernel is that the Enhanced Real-Time Clock (RTC) has been compiled into the kernel, as it is required by NIST Net. Packet forwarding has also been enabled, so the computer acts as a router.

NIST Net version 2.0.12 was installed. One tricky problem encountered was that running the emulator caused the system to become unresponsive or crash. The only solution was to rename /dev/rtc (Can be renamed to anything; this system uses /dev/rtcbackup). Compilation problems involving the RTC were even more severe on other Linux distributions that were tried (including Mandrake 8.0 and 9.1), so this was an acceptable solution.

One side-effect of using NIST Net in a router configuration is that when settings such as delay or loss are specified for 'default' source and destination, they are applied to both interfaces. Since all packets pass through each interface, this effectively doubles the specified delay or loss. To prevent the values from doubling, specify the source and destination IPs specifically (in both directions) and apply loss, delay or bandwidth constraints to only one of the two entries.

4.1.2 Network Interface Configuration

The NIST Net router was configured to route between the 192.168.1.0/24 and 192.168.0.0/24 subnets. The server was configured with a 192.168.0.0 subnet IP, using the NIST Net router's 192.168.0.1 interface as a default gateway. The client used the router's other interface, 192.168.1.2, as its default gateway. It is important that both systems use the router as their gateway, and that any other interfaces are properly configured or turned off.

4.2 Testing Procedure

To make our tests fair and repeatable, live video capture was not used, opting instead for prerecorded footage. Three clips approximately 30 seconds long were chosen to keep testing times manageable. Each clip is classified into one of the following categories:

Low Motion – Stationary camera, infrequent or subtle movement ("Torx": human subject sitting stationary; movement in scene is mainly of object held in subject's hands.)

Medium Motion – Mostly stationary camera, high movement of subjects in video ("Bball": several basketball players engaged in a game; camera pans slightly to follow the players.)

High Motion – Camera pans, intense subject movement ("Rat": Chromatically simple background; extremely fast panning and motion of radio-controlled electronic rat.)

Aesthetically, high motion video appears to suffer more seriously from lost frames than low motion video. However, aesthetics are difficult to quantify, and our tests are focused primarily on the playable frame rate regardless of video type. The reason three levels of motion were tested is that despite a constant bit rate, the I, B, and P frames of each video have different average sizes. The chart below shows the average size of each frame type for the three test videos.

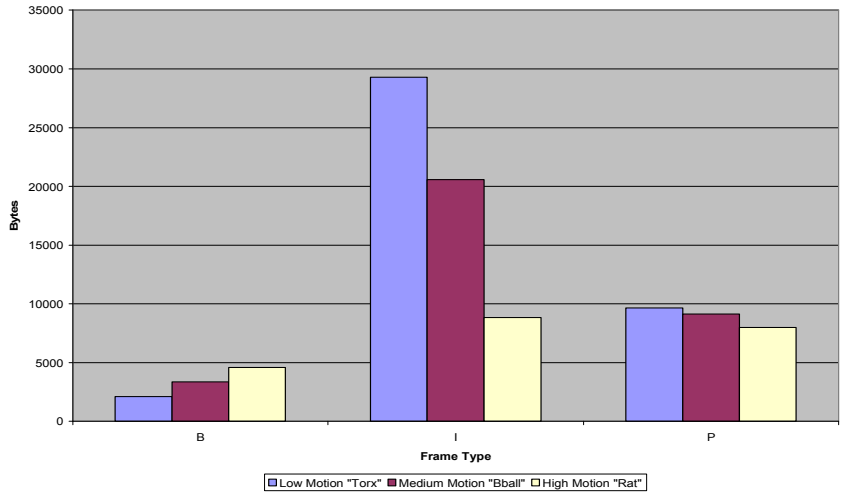

Figure 11 - Average size of I, B, P frames for each test video

In high motion video, I, P, and B frames are all nearly the same size. So, for a given bit rate, high motion video will be more resistant to packet loss than low motion because the chances of losing a packet of an I frame (most detrimental to frame rate) are less because I frames make up a smaller percentage of video packets. For low motion

video, I frames tend to be very large compared to B frames. Because most of the video data is weighted in the I frames, the loss of one packet is more likely to occur in an I frame. This results in a lower frame rate for a given packet loss, especially since each lost I frame causes the loss of an entire GOP.

ffMPEG was configured to encode clips at 1500kbps, 320x240 resolution, and a GOP of IBBPBBPBBPBB. For each test, the desired values in NIST Net were set and QOS modules were started. The client first runs "`./udpreceiver++ | ./fec -decode | ./preplayer mpegout.mpg log.txt`" and "`ffplay mpegout.mpg`". This puts the client into a waiting mode, and it will begin as soon as it receives the first video frame. The server runs "`./encode_file.sh <filename>`" to ready the encoder and "`fec -encode | ./udpsender++ <ip>`" to start the session. After the video is complete, CTRL-C terminates the client and appends a summary to the log.

Tests were performed with three encoding methods: No FEC, Fixed FEC at 10%, and Adaptive FEC with temporal scaling (Wu's adaptive model). A fixed FEC value of 10% means that 10% of the video's original data rate is added as error correction overhead. Only the adaptive FEC tests utilize the temporal scaling scheme from Wu's model. For every set of tests, including the tests with no FEC, the video passed through all of the components as shown in our system diagram. The only configuration difference is the command-line argument passed to the FEC module. There were many possible variables to choose from when designing an extensive series of tests. The packet loss percentage was varied, since it has a clear effect on all three encoding methods in our system.

4.3 Test Results

The following graphs show the effect of packet loss on playable frame rate for each of the three videos using each encoding method. Each data point represents the average playable frame rate from one test run of the video, as reported by the preplayer log file.

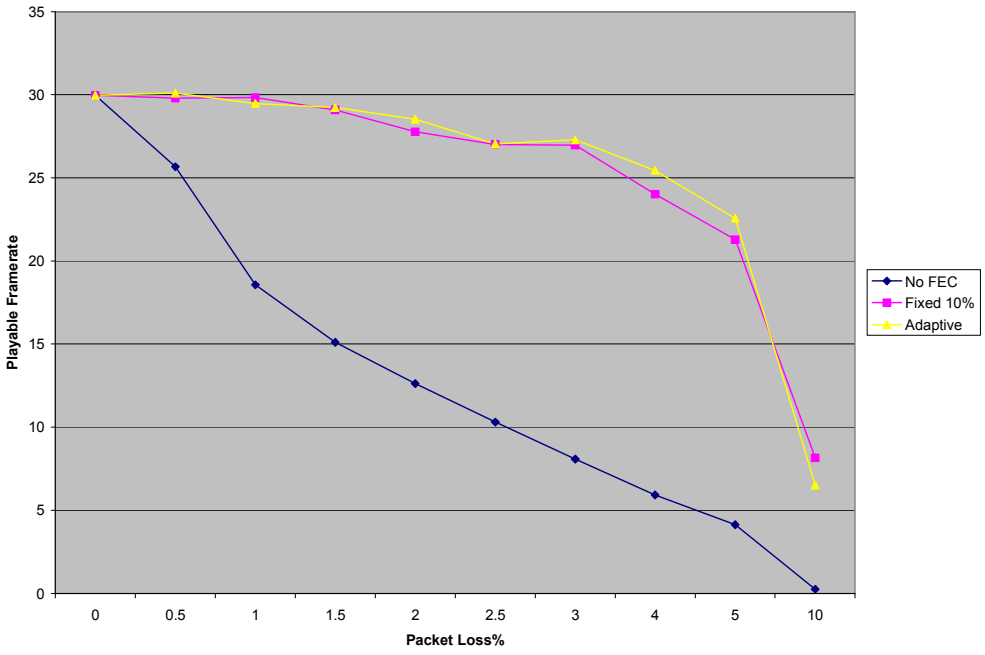

Figure 12 – Effect of packet loss on playable frame rate for low motion video (Torx)

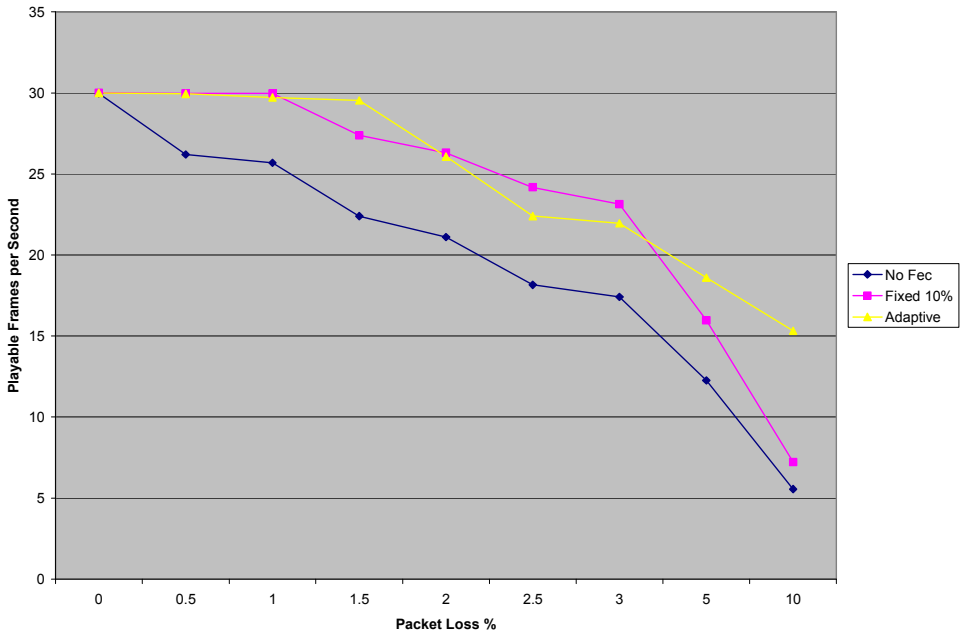

Figure 13 – Effect of packet loss on frame rate for medium motion video (Bball)

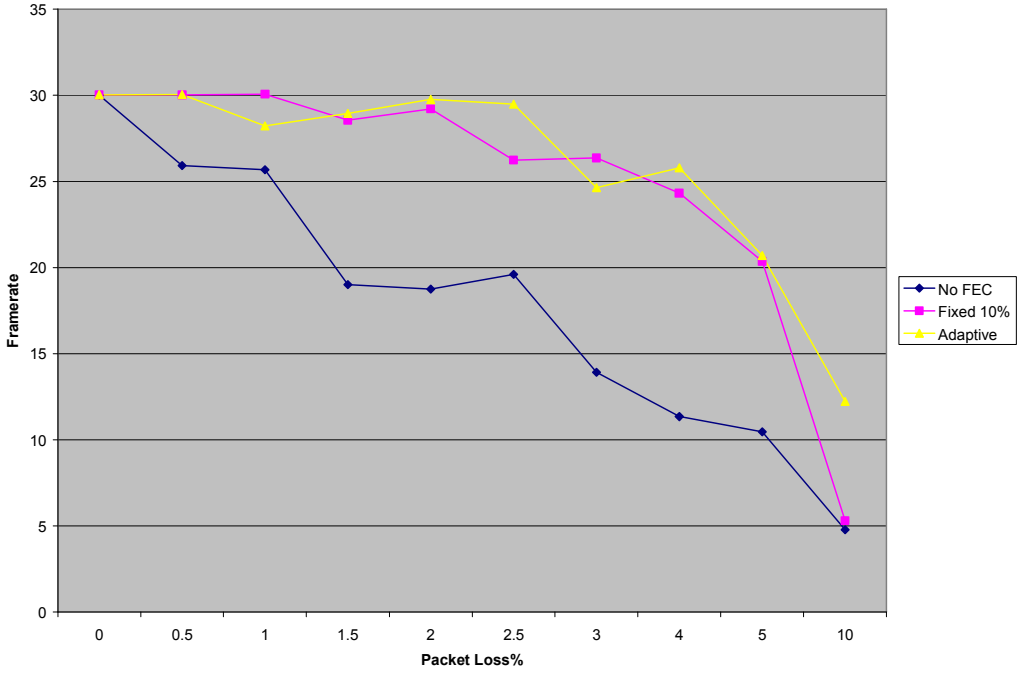

Figure 14 – Effect of packet loss on frame rate for high motion video (Rat)

4.4 Analysis of Results

It is clear that a small amount of packet results in a significant degradation in frame rate when no error correction is used. The graphs above show that FEC keeps frame rates consistently higher for increasing loss. Adjusted FEC slightly outperforms the 10% fixed FEC, but for packet loss alone, the difference is not dramatic. Before the testing sessions, a 10% fixed FEC was used, and it was unclear that its performance would appear similar to that of the adaptive FEC. Smaller values for fixed FEC would result in an overall decrease in playable frame rate as loss increases. Larger amounts of fixed FEC would improve playable frame rate at higher loss until the data rate exceeds the maximum allowed by the congestion control, after that point the frame rate would decline.

The relative size of MPEG frames makes each class of video perform differently in our system. The high motion video has a higher overall frame rate than the low motion video due to the effect explained with Figure 11. Consequently, a given amount of FEC has a greater effect on high-motion video. Adding one packet of FEC to an 8-packet I frame increases its chances of being transmitted successfully more than adding one FEC packet to a 28-packet I frame (as in low-motion video).

With packet loss alone, the system's integration of the rate control module does not display the full range of its capabilities. When NIST Net was used to limit bandwidth, results clearly favored the adaptive FEC model. Unfortunately, these results did not produce useful data, for the following reasons. The QOS module quickly discovered the bandwidth limitation and limited the transfer rate. Wu's algorithm applies temporal scaling on the fly, and a choppy, lower-bandwidth video emerges on the client side.

When this same test was performed using the Fixed FEC and No FEC methods, there was no temporal scaling (since it is exclusive to the adaptive FEC model). This resulted in all video frames being transferred over the rate-controlled link, causing the video to play back at a fraction of its original speed. The video *appears* better because motion is more fluid but in reality it is slow-motion and would never be acceptable for a live video application. Wu's model with temporal scaling reduces the delay to near realtime by reducing the amount of data that must be transferred.

Packet delay also affected frame rate. The table below shows the frame rate using the adaptive FEC with a 0.5% packet loss and varying delay (rtt).

Delay (ms)	25	50	100
Framerate (fps)	30	23.6	9

Figure 15 – Framerate with adaptive FEC at various latencies

In the test, temporal scaling was applied in the face of delay, lowering the frame rate.

5. Conclusion

Development of the AFEC system was successful. Our system implements Huahui Wu's adaptive forward error correction model with temporal scaling. It has the ability to capture and stream live or prerecorded video, and data is transferred over the custom TCP-Friendly congestion control module.

We were pleased to have met all major objectives of the project, especially after the problems encountered with DCCP. Our implementation of adaptive forward error correction shows promise as a new way to improve performance of streaming MPEG video. The system has been designed with flexibility and modularity in mind, and will make a great tool for others to use.

Though testing the system yielded a large amount of log data, our evaluations have been somewhat limited. Test results over the NIST Net network show that our implementation of adaptive FEC performs at least as well as 10% fixed FEC. Analysis suggests that in real-life network conditions, adaptive FEC would have higher performance and efficiency than fixed FEC, and the congestion control would allow other traffic, including multiple AFEC streams, to achieve improved performance.

6. Suggestions for Future Work

There are several interesting aspects of the system that can be explored, given the time and resources. When a standard DCCP or TFRC implementation is released, it would make sense to incorporate it into the AFEC system (such an implementation could entirely replace our UDP/QOS components). Our system's modular design makes it easy to modify or replace components while still keeping the simple interfaces intact.

The system currently does not incorporate audio. The adaptive FEC model was designed only for video, so our system currently ignores audio when encoding an MPEG stream. The ffMPEG code allows for extraction of audio in the same way video frames are extracted, and the FEC component is capable of encoding any data, including audio. Audio could be sent as a separate data flow, interleaved, or even encoded with FEC.

Nothing in the system's design prevents it from operating in a bi-directional video conferencing setup. One instance of the system would need to be initiated in each direction. However, the hardware used to develop the system lacks the processing power to provide full frame rate in two instances of the system. If it is not already possible with today's fastest desktop processor, it will be soon.

References

[1] H. Wu, M. Claypool, and R. Kinicki, “Adjusting Forward Error Correction with Temporal Scaling for TCP-Friendly Streaming MPEG”, Worcester Polytechnic Institute, 2003.
[2] D. Le Gall, “MPEG: A Video Compression Standard for Multimedia Applications”, Communications of the ACM, April 1991.
[3] Berkeley MPEG Tools, http://bmrc.berkeley.edu/frame/research/mpeg/
[4] FFMPEG Multimedia System, http://ffmpeg.sourceforge.net/, September 2003.
[5] J. Widmer, “Implementation of the TCP-Friendly Rate Control Protocol“, http://www.icir.org/tfrc/code/
[6] T. Sohn, E. Zolfaghari, A. Evlolgimenos, K Hao Lim, and K. Lai, “Work on the Datagram Congestion Control Protocol“, http://www.cs.berkeley.edu/~laik/projects/dccp/
[7] M. Erixzon, N. Mattsson, J. Häggmark: http://www.freebsd.dccp.org/
[8] P. McManus, “DCCP Linux Kernel Patches”, http://www.ducksong.com:81/dccp/
[9] D. R. Shier, K. T. Wallenius (Eds.), “Some error-correcting codes and their applications”, Chapter 19 of “Applied Mathematical Modeling: A Multidisciplinary Approach”, CRC Press, Boca Raton, FL, 1999.
[10] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, “A digital fountain approach to reliable distribution of bulk data”. In Proc. ACM Sigcomm '98, pp. 56-67, Vancouver, Canada, September 1998.
[11] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments," Proceedings of ACM SIGMOD'1995.
[12] L. Rizzo, “On the Feasibility of Software FEC” University of Pisa, 1997.
[13] L. Rizzo, “Effective Erasure Codes for Reliable Computer Communication Protocols.” University of Pisa, 1997.
[14] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. “Practical Loss-Resilient Codes.” International Computer Science Institute, 1998.
[15] P. Karn, “Forward Error Correction Codes”, http://www.ka9q.net/code/fec/
[16] M. Carson, D. Santay, “NIST Net – A Linux-based Network Emulation Tool”, National Institute of Standards and Technology, 2002.