Reducing Packet Loss in Real-Time Wireless Multicast Video Streams with Forward Error Correction

Simen Fonnes

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2018

Reducing Packet Loss in Real-Time Wireless Multicast Video Streams with Forward Error Correction

Simen Fonnes

Reducing Packet Loss in Real-Time Wireless Multicast Video Streams with Forward Error Correction

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Wireless multicast suffers from severe packet loss due to interference and lack of link layer retransmission. In this work, we investigate wether the most recent Forward Error Correction (FEC) draft is suitable for real-time wireless multicast live streaming, with emphasis on three main points: packet reduction effectivity, and latency and overhead impact. We design and perform an experiment in which we simulate wireless packet loss in multicast streams with a *Gilbert* model pattern of $\approx 16\%$ random packet loss. We check all FEC configurations (L and D values) within several constraints: maximum 500 milliseconds repair window (latency impact), 66.67% overhead, and a maximum L value of 20. For all these L and D values we stream the tractor sample three times, to avoid possible outliers in the data. We show that packet loss reduction in the most recent FEC draft is effective, at most reducing from $\approx 16\%$ down to $\approx 1.02\%$. We also show that low latency streaming can be conducted, but it requires a minimum of 160 milliseconds additional latency for our stream file. The overhead for such low latency can be as high as 66.67%.

Acknowledgments

I would like to thank my supervisors, Professor Dr. Carsten Griwodz and Professor Dr. Pål Halvorsen, for providing me with excellent guidance throughout this thesis. I would like to thank my girlfriend Ina Alette Fosberg, for making illustrations and for her continuous support throughout this work. Additionally, I would like to thank my friend and co-student Fredrik Løberg for having several helpful discussions with me during the time of this thesis. Finally, I would like to thank my parents Øyvind and Anita Fonnes, for providing feedback and support throughout this thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background and Motivation

Live video broadcasting remains a popular medium for distributing activities such as sporting events or video games over the internet. From the origin of the internet and until the late 2000s, video broadcasting was mostly carried out using push-based UDP streams. UDP streams were utilized because TCP's retransmission mechanism can cause undesirable delays if there is not sufficient bandwidth (B. Wang et al. 2008). However, receivers behind firewalls and proxies had problems with connectivity because these devices often block UDP ports. Additionally, receivers with arbitrary bandwidth had issues buffering the streams because the bitrate would exceed their bandwidth. Therefore, in the late 2000s, broadcasting hosts moved from using push-based UDP streams to pull based adaptive bitrate streams over HTTP.

The current standard for adaptive bitrate streams over HTTP is DASH (Dynamic Adaptive Streaming over HTTP). DASH addresses the issues with UDP streaming by dividing videos into proportionate time slices. Each slice is firstly duplicated in various bitrates and then uploaded to a HTTP server. Receivers then pull slices with an appropriate bitrate for their bandwidth. Because the streams are carried over HTTP, they easily traverse any firewall or proxy because the data is requested, rather than pushed.

DASH solves some of the issues with UDP streaming. However, in live video broadcasting, it has its problems and limitations. Firstly, it is typical for receivers in the same live stream to pull identical slices concurrently from the HTTP server. Because DASH streams are transmitted over unicast connections, the slices have to be sent separately to each receiver. These redundant data transmissions lead to excessive workload usage, as well as excessive bandwidth consumption for the server. Secondly, a video slice cannot be made available until it is captured. [1] For example, the stream is delayed by three seconds if each slice contains three seconds of video. Third, when receivers join the DASH stream, they play the first acquired slice from the beginning regardless of when the slice was acquired. This is necessary because it gives sufficient time for the next slice to be pulled. However, this makes receivers out of sync if they receive their first acquired slice at different offsets. These offsets increases if the slice size is large. [2] Finally, it is typical for broadcasting hosts to distribute workload and bandwidth usage by using web caches. These web caches are typically located geographically close to the receivers, but they need to be synchronized. Therefore, every slice has to be sent out to all web caches before they are made available, which further increases the delay. Note that write-through caches such as *Squid* can be utilized to reduce latency.

The main problems with DASH in live streaming is *scalability*, *latency*, and *synchronization*. To solve these issues, we propose the idea of using push-based multicast UDP streams. In contrary to unicast streams, data in multicast streams is only transmitted once from the server and then duplicated by the network nodes on the way to their destination. This is done in a push-based manner where every receiver subscribes to a multicast group. Because the data sent in UDP streams do not have to wait for the creation of a slice, the data can be transmitted directly after being encoded and compressed. This results in significantly reduced latency. Also, UDP streaming does not have any significant synchronization issue because the data is pushed directly to its receivers. The only exception is latency caused by geographically separated receivers.

The primary goal of this thesis is to improve live streaming in a football stadium. In this scenario, the receivers are physically present and are watching the stream on their mobile devices. The receivers connect to a server over Wi-Fi. As previously mentioned, one of the significant issues of UDP streaming is that firewalls and proxies block them. However, in this scenario, we control the communication endpoints. Therefore, we can allow the UDP streams to pass through the local wireless network. It is, therefore, a requirement that our tests should involve receivers connected over LAN Wi-Fi. Additionally, because the receivers are using their own devices, it is

1. While not beneficial, it is possible to deliver instant DASH streaming by sending H.264 slice instantly.

2. It is, however, possible to synchronize DASH participants by using Merge and Forward by Rainer et al. (2018).

vital that our solution must support the major mobile operating systems.

Media platforms such as IPTV uses multicast for video streaming services. However, there is a lack of support for multicast in most routers on the internet. Multicast Technologies (2008) measured that less than 5% of the internet support multicast. There are several reasons for this, both from a commercial and technical standpoint. Because there is no direct correlation between a receiver and host in multicast, it makes it complicated for the hosts to know who it is serving, and thus who to bill.

Additionally, from an ISPs standpoint, unicast is profitable because users can be charged higher for more bandwidth usage and therefore saving bandwidth using multicast does not make commercial sense. From a technical standpoint, multicast invokes security implications such as lack of IPSec support, in addition to constant bandwidth occupation, because it pushes data continuously through the network. For that reason, multicast is usually only used in closed environments such as local area networks or services directly from the Internet Service Provider (ISP) such as IPTV. IPTV is also often not a part of a users internet bill and thus saving bandwidth makes sense.

While wired multicast is underdeveloped and has several issues, wireless multicast has even more problems. Perkins et al. (2018) are currently working on an internet-draft discussing the most significant issues of wireless multicasting. Firstly, wireless multicasting has high error rates in congested networks. It is impossible for the host to provide retransmission of lost packets because link layer acknowledgment is turned off. Secondly, receivers in wireless networks adapt transmission rates based on the distance to the access point. However, in multicast, the transmission rate is most commonly set low to provide data to the furthest receiver. Additionally, mimo schemes cannot be applied as they are only for single receivers. Third, as a consequence of the low transmission rate, the packet transmission is taking longer compared to a higher transmission rate. Because more data is in the air, there is also more interference. Fourth, every receiver has to wake up frequently because the host is continuously pushing data through the network. Since the host never backs off this leads to high power consumption.

While push-based RTP multicast streams over Wi-Fi has several issues, in theory, it presents a better alternative to DASH regarding latency, scalability, and synchronization. It is, however, important to note that in our scenario we are streaming live video within a closed Wi-Fi environment. In this environment, issues with blocking firewalls and proxies, as well as low multicast support are no problem because we control the communication endpoints. The main problem using multicast in our scenario is handling congestion without sacrificing latency. However, even with these issues, it does not mean that multicast streams cannot be applied. Taking all of this in regard, we are left with one enclosing question: Can push-based multicast UDP streams replace pull-based unicast DASH streams?

1.2 Problem Statement

Before we can provide a sustainable multicast video stream, we must first address the issues with multicasting video. One of the significant problems is packet loss as a result of no link layer retransmission as well as high interference. Nayarasi (2012) performed a multicast video streaming test over Wi-Fi and reported the stream as being not viewable. Additionally, (Perkins et al. 2018) state that it is not uncommon for packet loss rates of 5% while streaming over Wi-Fi using multicast. Therefore, we limit our focus in this thesis to reduce packet loss, which we consider a reasonable step towards replacing unicast with multicast streams.

The most common way to handle error rates in wireless multicast networks is to apply multicast over unicast. Multicast over unicast transforms multicast packets to unicast at the wireless endpoints. By modifying the packets to unicast, each packet yields the benefit of retransmission as well as lower power consumption. However, it comes at the price of higher bandwidth consumption.

Another approach for reducing packet loss is Forward Error Correction (FEC). In environments where retransmission is expensive or impossible such as satellite transmission or video streaming it is typical to use FEC. By applying FEC, we proactively send control data to rebuild packets lost in transmission. The amount of FEC protection is dependent on the interference in the network. In severely congested networks, FEC may duplicate entire streams. In less congested networks, however, a single parity product may protect multiple of the original packets.

At this time, Zanaty et al. (2018) are constructing a new standard for generic FEC in RTP. This FEC draft addresses scalability issues with the earlier specifications including RFC 2733, RFC 5109 and SMPTE 2022-1, and offers several improvements. Therefore, we will implement this FEC draft and test its effect on reducing packet loss. To the best of our knowledge, there have not been conducted any tests regarding packet loss in wireless multicast video streams using this draft.

In summation, our problem statement comes down to if we can reduce packet loss in wireless multicast streams using FEC. We address this by breaking the problem statement into the following questions:

- How effective is FEC at reducing packet loss in Wi-Fi Multicast Networks?
- Can FEC maintain low latency in real-time video streams?
- Can FEC preserve acceptable bandwidth consumption?

1.3 Approach

We approach the problem statement in the three following parts:

- Determine why and how packet loss effects wireless video streams.
- Design and execute an experiment where we investigate the effects of the most recent FEC draft in multicast video streaming.
- Evaluate the results of the experiment.

In the first part, we discuss in depth of how video streaming works with an emphasis on RTP and wireless multicast. We consider the problems with wireless multicast and how we can make the problems tolerable.

In the intermediate part, we design an experiment where we capture data from a congested multicast video stream with FEC protection. This experiment design contains the packet loss patterns during the stream, as well as the different FEC configurations we apply. We collect the packet loss pattern from related literature. Our goal is to make the experiment as reliable as possible by testing as many FEC configurations as possible.

In the final part, we evaluate the experiment as mentioned above by comparing the captured video before and after it is passed through the FEC decoder. In the comparison, we examine the effects of the FEC decoder in regard to inflicted packet loss and latency, as well as overhead.

1.4 Scope

The scope of this thesis covers the effects of the most recent FEC draft in congested and delay sensitive WLANs. Due to our limited time and resources, we will not compare our implementation with a unicast DASH solution. To see the true benefit of multicast vs. DASH, we would need a large test environment with multiple access points, and numerous clients per access point. We, therefore, consider this future work.

Our thesis is structured around improving bandwidth consumption for multiple clients watching live video over a congested WLAN in a football stadium. In addition to video streaming, it is also important for clients to be able to rewind the video stream, either to custom points in time or points in time provided by the server. These timestamps provided by the server are manually marked by employees which correspond to significant points in time (e.g., goals and fouls) which in turn yields high re-watch rate for clients in contrary to random rewinds. On-demand multicast approaches have been studied extensively such as Pull Patching by Jacobsen et al. (2010), which we discuss in Chapter 2. However, in this thesis, we only focus on improving the primary video stream and thus do not study this portion of the scenario.

1.5 Outline

This master's thesis is structured in the following order:

- **Chapter 2 — Wireless Streaming in Dense Places**
  This chapter presents an overview of video streaming with emphasis on multicast wireless streaming.
- **Chapter 3 — Generic Forward Error Correction in RTP**
  In this chapter, we discuss how generic forward error correction in RTP has evolved over the years. Additionally, we provide an in depth discussion of the most recent FEC draft.
- **Chapter 4 — Design**
  This chapter presents the design of our experiment, which includes both the environmental setting and preliminary testing, as well as the details of the experiment itself.
- **Chapter 5 — Implementation**
  This chapter presents our C++ implementation of the most recent FEC draft in the Live555 Media Streaming Library. We utilize this implementation during the experimentation described in the previous chapter.
- **Chapter 6 — Evaluation**
  In this chapter, we evaluate the experiment results which includes a detailed analysis as well as a thorough discussion.

- **Chapter 7 — Conclusion**
  In the final chapter, we summarize our findings and contributions, as well as provide possible directions for future work.

Chapter 2

Wireless Streaming in Dense Places

In this chapter, we discuss how video streaming is carried out today and what problems it faces, especially with live and wireless streaming. In Section 2.1 we discuss how video streaming is carried out today. In Section 2.2 we discuss the Real-time streaming protocol and how it has developed over the years. In Section 2.3, we provide an overview of the inner workings of IP multicast. Finally, in Section 2.4, we summarize and conclude the chapter.

2.1 Video Streaming

Consumption of video is an important part of human life in 2018. eMarketer (2017), estimates that a United States adult spends, on average in 2018, 5 hours and 13 minutes watching video, every day. They also show that the total time of video consumption has more or less stayed the same. TV delivers the majority of video, but in the recent years, more time has shifted over to consumption from *digital video devices*. This shift has made Video on Demand and Live streaming companies such as Youtube and Twitch more significant.

As mentioned in the introduction, video streaming is primarily carried out using unicast connections. Unicast is a one to one connection between a server and a client and does not scale when redundant data is sent. Therefore, because we consume more and more video with higher and higher quality, the bandwidth requirement has also increased both for the providers and consumers of video.

2.1.1 Dynamic Adaptive Streaming over HTTP

DASH (Dynamic Adaptive Streaming over HTTP) is an adaptive streaming technique proposed by Carmel et al. (2002). Briefly explained, DASH separates a media resource into small segments and encodes these segments in different qualities; ranging from low to high. When a client requests a specific resource, and receives it, the flow of segments is measured. Based on this measurement, the next segment to pull is chosen if the bandwidth can handle the potential new flow of segments. In other words: The quality of the segment is chosen based on the client's bandwidth, which results in stability for the clients.

To account for the multiple redundant streams in DASH, web caches are typically used. This would be, in simple terms, multiple small servers placed in different areas containing popular streaming content. This means that a participant would not have to retrieve the stream from the main server, if the stream was stored in the web cache. This web cache will be geographically closer than the main server to the client (e.g, same country or city). This effectively reduces the main server's bandwidth requirement, and provides less latency, due to its geographic location. An example of web cache usage would be Netflix' streaming service, which as of 2017, uses EVCache. This is a RAM based cache, with several micro services around the world (Siddiqi 2016).

2.1.2 UDP Streaming

UDP video streaming is a technique used when low latency is more important than maintaining the integrity of the information sent. UDP streaming streams a resource continously without waiting for any acknowledgement, in contrary to DASH. However, there is usually a need for metadata in UDP videostreaming. Therefore, protocols such as SDP and RTSP is used to receive metadata and negotiate terms for the stream. These protocols are carried over TCP. After this exchange has occured, the stream is sent continously. UDP doesn't have any built in error recovery, and therefore in congested networks, it can be affected by high loss rates. It also does not support any adaption of quality out of the box. However, research ideas for these exists such as pull patching.

Pull Patching

Pull patching is an idea by Jacobsen et al. (2010) that combines the Patching idea mentioned in Section 3.1.2 with DASH. In this article, we learn that the majority of the logic is placed with the client rather than the host. They argue that the client has the most accurate information about it's own bandwidth condition, which is the reason for this architectural choice. We can think of the client as the consumer and the host as the distributor. A client asks for a certain resource and is delegated to a multicast stream for the main media content, and a patching stream to compensate for potential packet loss, and receiving the beginning of the stream. The multicast stream is distributed in four different qualities (DASH), from which the clients selects one, according to their current bandwidth condition.

2.1.3 Live Streaming

Before digital video stremaing we used analog video streaming. The signals were sent as scanned lines and broadcast directly to consumers using radio. This yielded very low latency, basically the speed of light minus overhead. When the internet became popular, there was live streaming using UDP. This introduced latency problems because the video had to be compressed befrore transmitted to its destination, to save bandwith. Additionally, UDP streaming has several issues. In many consumers firewalls it is blocked and therefore we have to use SDP or RTSP but these are not sucessfull if blocked. Therefore, several companies started to use DASH for live streaming. Since DASH uses TCP as its undelying protocol it is not blocked.

DASH yields several benefits, but also issues in live streaming. Because of its native adaptivity, the users gets the best quality for their bandwith. In addition, DASH makes it easy to rewind, and pause, because the client simply pulls its desired segments. That is, unless the segments are deleted. However, DASH divides videos into segments. This adds an additional delay of the length of the segment, because it has to be created before it is made available. Another problem is syncronization. When clients pulls segments at different times they always play the segment from start. This adds additional delay. Also, if another client is playing the middle of that segment the two will not be syncronized. When using web caches etc, additional buffer time might be added for the segments to be transmitted to these caches. This further increases the delay.

2.2 Real-time Transport Protocol

RTP is a protocol designed to transport real time and diagnostic data, by Schulzrinne et al. (1996). The real time and diagnostic data are transported using two sub protocols: RTP and RTP Control Protocol (RTCP), respectively. It is important to note that RTCP does not handle packet loss or other issues with the participants; it only provides the data needed to solve these issues. In the end, the responsibility to handle these issues lies with the developers. In RTP, different payload formats are described in RFCs.

2.2.1 RTP

The RTP protocol uses RTP packets, which consist of a header and a payload. The first two bytes of the header consist of fields describing: the version of RTP used, if padding is used, if the packet is extended, how many participants there are, marker and the payload type. A sequence number is defined for the next two bytes. This is incremented for each packet sent, which makes it simple to detect packet loss. In the next word, a timestamp is provided. This is used to play the content of the payload relative to each other, even if packet loss has occurred. One can imagine it as two clocks on each side of the connection that the timestamps are compared to. Next, we have the SSRC, which is the ID for the host in the current stream. Finally, we have the contributing sources (CSRC identifiers). This is a list containing the sources that contribute to the given packet, merged by a mixer. A mixer is a mechanism to merge streams (e.g, to save bandwidth for participants with restricted bandwidth). The RTP payload consists of the data being transported, with some additional padding if it was defined in the padding header field.

2.2.2 RTCP

RTCP serves two main purposes: Provide diagnostic data among the participants, and inform the participants of the different host identifiers (CNAMEs). In RTCP, five different packets are used: *Sender report*, *Receiver report*, *Source description*, *Goodbye* and *Application-specific message*. These packets can be sent together in a *compound packet*, to reduce packet overhead. The sender and receiver reports contain the actual diagnostic data, either sent from an active sender or from a receiver. A compound packet should always include one of these reports along with a source description, because these fulfill the main purposes of RTCP. To inform the other participants if a participant is leaving the conversation, a Goodbye packet is sent. This packet also includes a description of why the participant is leaving. The Application-specific message is a packet used for developers to create custom RTCP packets.

2.2.3 RTP discussion & mechanisms

Several principles and ideas for mechanisms were proposed in the RFC 1889, as there are many issues to handle with varied bandwidth and a variable number of participants. All the discussion and mechanisms in this Section are derived from Schulzrinne et al. (1996).

Scalability

With RTP packets, scalability should not become a problem, as it is not expected for more than two participants to speak simultaneously in a conference call. In RTCP on the other hand, the amount of control packets would increase linearly to the participants without some control mechanism. We resolve this growth by limiting the control packet flow to a fraction of the session bandwidth. The senders are given more headroom, so new participants can receive the CNAME as soon as possible. Describing the algorithm itself is not within the scope of this essay.

Mixers and translators

RTP also introduces the terms *mixers* and *translators*. A mixer is a mechanism to handle bandwidth variations, among the participants. Simply put, mixers multiplex signals from participants with higher bandwidth, to a single signal. This signal is then compressed, based on the bandwidth of the lower bandwidth participants; and finally sent to these participants. This will provide participants with the quality their bandwidth can handle.

A translator is a mechanism that yields several responsibilities in RTP Multicast. When passing RTP/RTCP packets through a firewall, the firewall may drop some of the packets, if the firewall's criteria are not matched. In that case two translators are used, one outside and one inside the firewall. The translator outside the firewall tunnels the packets through a secure channel. When the packets are inside the firewall, the other translator delegates the packets into the original multicast stream. A translator can also keep track of the original CSRC when combining packets, IPv4 to IPv6 conversion etc.

SSRC identifier collision

If a SSRC identifier is not unique, we get a collision and looping of packets might occur. An example of looping could be if a mixer receives data, and sends it back to the same multicast group. Before the duplication is handled, the receivers should discard packages from one of the senders, until the problem is fixed. Duplicate SSRC identifiers are the sender's responsibility to fix, and are fixed by leaving the chat using a Goodbye packet, followed by generating a new random SSRC identifier and rejoining the conversation.

Security

As mentioned, in both RTP and RTCP headers, there is a padding count field to add padding at the end of the packet. This is used for block ciphers that require a specific static length to be encrypted. Only the payload is encrypted, because that is where the sensitive information lies. When encrypting, the padded amount of bits is added at the end of the packet, and when decrypting the process is reversed. By the time the RFC 1889 paper was published, DES (NIST 1995) was the default encryption algorithm. As of 2017, AES (NIST 2001) is the default encryption algorithm, but is also a block cipher which makes it fully compatible.

Don't mix audio and video

Originally in RTP, audio and video were to be sent in separate packets. One of the reasons for this separation is the fact that: If a participant only needs one entity of either video or audio, it is a simple procedure to provide one of these entities when the packets are separate, for obvious reasons. Examples of this can be if a participant only wants to receive the audio in a conference call, and if the audio track of a movie is dubbed in multiple languages. Other reasons describe issues with synchronization if video and audio is combined, such as sequence number handling. This is however not relevant using MPEG streams because audio and video consist of the same format, which is discussed further in Section 3.1.1.

2.2.4 Development of the RTP-standard

Development of the RTP-standard is defined by a set of *Request for Comments (RFC)*. The first standardization of RTP was presented in the RFC 1889 document, which is what Section 2 of this document describes. In Section 3, we discuss how RTP has developed as a protocol over the years, from its origin in the 1990s and to today.

Prior to the RFCs, there were Internet Engineering Notes (IENs). The internet stream protocol (ST) by Forgie (1979), shows early ideas for audio conferences, and was later converted to RFC 1190, by Topolcic (1990). However, we will only concern ourselves with RFCs directly attached to RTP in this Section.

1990s

The world wide web's popularity grew rapidly during the 1990s, and with it came the possibility of performing Internet conferences, streaming media etc. With low bandwidth being a major issue, the engineers had to look towards multicast solutions, as unicast solutions were too demanding for the bandwidth. However, there was a necessity to time-align packets as well as multicast diagnostic information about the participants in the multicast group. As a result; in 1996, H. Schulzrinne, GMD Fokus & S. Casner presented RTP, combining a fast, scalable, jitter handling protocol, with bandwidth saving multicast.

RFC 2038 In Section 2.4.5, we discussed the idea to never mix audio and video in the same RTP packets, which was promoted in the RFC 1889. However, in RFC 2038, optimization for sending MPEG1/MPEG2 video was presented (Hoffman et al. 1996). MPEG is combining both audio and video into a single format, using the system streams for MPEG 1 and transport streams for MPEG 2; and thus sending both audio and video in the same RTP packets. This is nonetheless not a replacement of the original idea, but rather an expansion. This document was quite significant for RTP, as we still use MPEG1/MPEG2 to this day. This includes services such as IPTV, which is discussed in Section 4.2.1.

2.2.5 2000s

In summary, the 1990s were the origin of RTP, which yielded the first standards of the protocol. As we move into the 2000s, we see several RFCs being presented, expanding the standard in many different areas. These areas include video streaming, text and music communication, etc.

RFC 3190 RFC 3190 describes the sampling of 12-bit nonlinear, 20-bit linear, and 24-bit linear audio, and how to pack it into the RTP payload. The main motivation behind this RTP expansion, by Kobayashi et al. (2002), was to transport high quality audio over the Internet. It was intended for media such as DAT (digital audio tape) and DV (digital video), which were relevant in the 2000s. Workarounds could have been used, such as to convert 12-bit nonlinear audio into 16-bit linear, sending it over RTP and then decode it back. This would use 33% more of the network capacity than needed, which was also a key motivation.

RFC 3550 RFC 3550 is the current standard for RTP, proposed by Schulzrinne et al. (2003). This document explains that RFC 3550 obsoletes RFC 1889. It is mostly identical, except for improvements to rules and algorithms used in the protocol. There is no change in the packet structure, nor the protocol principles. Schulzrinne et al. (2003., 1) explains it as:

> “The biggest change is an enhancement to the scalable timer algorithm for calculating when to send RTCP packets in order

> to minimize transmission in excess of the intended rate when many participants join a session simultaneously.”

RFC 3640 In RFC 3640, standardization of transporting MPEG-4 streams over RTP was proposed by van der Meer et al. (2003). In this document, we learn that MPEG-4 streams combine both audio and video in Access units. Some of the MPEG-4 elementary streams contain bits that prevent the stream from being playable if lost. In this case, it was proposed that this can be handled by using either access points or TCP. This is a generic streaming technique, so every MPEG-4 stream can be sent, no matter the encoding. However, for this to work, the modes given in RFC 3640 needs to be followed.

RFC 4103 In RFC 4103, real-time text (RTT) was introduced by Hellstrom and Jones (2005). This document explains the design of RTT, which is a sending *letter by letter* approach in conferences. It can be used simultaneously with other media, but the document suggest that separate RTP streams should be used. In Linux, *talk* was an application that used RTT. When every keystroke is seen by the people in the conference, it can provide a real-time feel. This actually turned out to be a major problem as the participants saw all the letters when someone was typing. If a person misspelled a lot it could become embarrassing and the feel of being monitored would frequently occur.

RFC 4175 In RFC 4175, Gharai and Perkins (2005) describe a standard for packing uncompressed video into the RTP payload, for transport. Uncompressed video is typically used in television and was therefore the main motivation behind this document. The formats chosen were recommended by ITU, and include standard and high quality formats. These video formats use scan lines in either progressive or interlaced scanning. In *progressive scanning*, a frame is displayed using the scanned lines in a top-down fashion. In *interlaced scanning*, the frames are used in pairs, where the first contains the even scan lines and the second the odd scan lines.

RFC 4696 RFC 4696 describes a recommended way of streaming MIDI over RTP by Lazzaro and Wawrzynek (2006). We learn that the main motivation behind this document is for musicians to play together, without the need to be in the same room. A digital keyboard instrument’s keys can be mapped directly to MIDI, which further can be sent over the Internet and played on the receiver’s side. In other words, we are sending the inputs, not the actual sound, as these are reproduced on the endpoints of the conversation.

2010s

In the 2010s, as media streaming becomes progressively more used, we see a few RFCs that expand the different video formats the protocol can transport, along with an update of RFC 3550.

RFC 6184 RFC 6184 is a description of transporting H.264 encoded video using RTP by Y.-K. Wang et al. (2011). In this document, it is discussed that H.264 produces an Internet friendly format, which separates the payload into packets. In these packets, the first byte contains the type of Network Abstraction Layer (NAL) unit used, and the rest of the bytes are the payload. NAL units are (put easily) segments of frames, which combined produces the actual frames of the stream. RTP can contain one or more of these formats, but as every packet is independent, a generic transportation of different data is achieved. This makes it easy to adapt to different formats.

RFC 7587 RFC 7587 describes the payload format for Opus-encoded speech and audio data by Spittka et al. (2015). In this document we learn that the Opus encoding was designed to be lightweight and scalable with bitrate and complexity. It is also adaptive, e.g if there is silence in a conversation a lower bit-rate of data can be sent (Discontinuous Transmission). As for the RTP payload, a varying number of frames are placed in Opus packets, which are then placed in the RTP payload. However the RTP payload should only contain one Opus packet, while Opus packets can contain any number of frames. The first byte of the Opus packet describes the encoding used in the packet. The Opus packet is an octet of bytes, which means no padding is needed in the RTP payload.

RFC 7656 RFC 7656 was presented as a further explanation of RFC 3550 by Lennox et al. (2015). The reason behind this update was due to critics claiming RFC 3550 was confusing and inconsistent. Therefore, RFC 7656 digs deeper into the core fundamentals of the protocol.

2.2.6 The history of RTP based research ideas & applications

In Section 3, we discussed the viewpoint of how the RTP standard has developed over the years. Taken this development into account, we discuss some of the research ideas and applications which use these standards, through the years.

1990s

As mentioned in Section 3, the world wide web's popularity grew rapidly during the 1990s, and with it came the possibility of performing Internet conferences, streaming media etc. With RTP being a candidate to handle the increasing bandwidth demand for media streaming, a few ideas emerged to use RTP Multicast for this task. Among these were Patching and Receiver driven layered multicast (RLM). More specifically, these ideas were invented around media streaming services called Video on Demand (VoD), which is the ability for clients to receive a video any time, while not being limited by a broadcast. VoD had its first trials in the early 1990s.

Patching When using RTP multicast to perform VoD services, there are many obstacles to overcome, among them are participants watching the same content at slightly different times. Patching by Hua et al. (1998) presents a solution to this problem by letting clients buffer an initial multicast stream, while simultaneously watching an additional provided unicast patch stream, containing the missing content. This is maintained until the client has caught up to the original multicast stream, in which case the patching stream is dropped. If the original stream is too far ahead, e.g the client doesn't have enough space to buffer the multicast stream, a new multicast stream can be created (graceful patching), or a new patching stream can be created, while buffering from the original multicast stream (greedy patching).

When using graceful patching, late joining participants will always use the latest multicast stream as the original stream. In the original paper, experiments were run where Patching was tested, with both graceful and greedy patching. In most cases, graceful patching resulted in more shared resources than greedy patching did, because the expense of creating additional multicast streams, using graceful patching, was significantly low.

Receiver driven layered multicast RLM is a solution for handling varied network speeds among the participants, in the multicast stream, proposed by McCanne et al. (1996). RLM divides the stream into different layers, with each stream building on top of the other, with the combination of layers providing the highest quality. This is called a cumulative model. In the original paper, there is also mention of an independent model. This model includes separate streams containing all the data, but in different qualities. However, this paper focuses on cumulative sessions, as it is argued that these make a more effective use of the bandwidth. While the cumulative model is in focus, compatibility towards the independent model is maintained.

The logic is moved from the server to the client, as it is the client's responsibility to change between the different layers. The logic behind the switches are join experiments, which is a trial and error approach: Attempt to upgrade layer, and check if packet loss occurs. If packet loss was detected we downgrade one layer. However if not, we maintain the upgraded layer.

2000s

The 1990s were the origin of RTP, which yielded some early research ideas, using the protocol. As we move into the 2000s, we see several applications make use of the protocol. In this Section we discuss an excerpt of these applications.

IPTV IPTV is a technique of broadcasting TV signals using the Internet, rather than satellite or cable. It uses plain RTP multicast to stream to all the participants. In the 2000s, most of the TV content was streamed in a singular fashion, e.g the content is only streamed once, and at the same time to all its participants. Therefore, it is only logical to use a multicast solution. Although invented in the 1990s, during the early to mid 2000s, a few IPTV services were commercialized. These included services in countries such as Canada in 1999 (Thompson 2011), USA in 2005 (Amino Communications 2005) and Sweden in 2005 (Ericsson AB 2005). IPTV has in general suffered a rather slow growth due to limited bandwidth, which made cable and satellite superior.

Media player applications Several media player applications support RTP in some way, but some of the more popular are Quicktime Player and Windows Media player (previously called Media Player). Although both were developed in the early 1990s, both of these media players supported RTP in the 2000s, and still do to this date.

2010s

Providing adequate bandwidth for unicast Video on demand (VoD) services has been a problem since VoD services was commercialized in the 1990s; and the 2010s are no exception. As media streaming becomes progressively more used, along with growing media content size, bandwidth remains a significant issue. VoD providers such as *Youtube* and *Netflix* have implemented DASH with web caches, to target this issue (Weil 2014). This works well when clients are spread out geographically, far away from the server. In addition, some ideas for using multicast are emerging this decade. Pull Patching and WebRTC are examples of these ideas, which will be presented in this Section, along with RTP in 4G conversations.

Pull patching Pull patching is an idea by Jacobsen et al. (2010) that combines the Patching idea mentioned in Section 3.1.2 with DASH. In this article, we learn that the majority of the logic is placed with the client rather than the host. They argue that the client has the most accurate information about it's own bandwidth condition, which is the reason for this architectural choice. We can think of the client as the consumer and the host as the distributor. A client asks for a certain resource and is delegated to a multicast stream for the main media content, and a patching stream to compensate for potential packet loss, and receiving the beginning of the stream. The multicast stream is distributed in four different qualities (DASH), from which the clients selects one, according to their current bandwidth condition.

Web Real-Time Communication Web Real-Time Communication (WebRTC) is an open API for development of integrated video conference calls in a web browser, described by Singh et al. (2013). Further in this article, it is explained that WebRTC uses RTP as its main protocol for transporting media. As discussed earlier, RTP is very suitable for Internet conferences. WebRTC is still under development, yet applications such as Talky can be used for video conferences. As of 2017, mobile support has been implemented.

For handling congestion, WebRTC uses Google's implementation: the Real-Time Congestion Control (RRTCC). Please note that discussing the algorithm in depth is out of scope of this essay. In a brief summation with a sender and a receiver: The sender uses the observed loss rate (p), measured RTT and the bytes sent with the TFRC equation. This is used to estimate the data rate, unless a receiver report is received, in which case this is used instead. The receiver on the other hand compares the interval between timestamps of incoming frames with the interval between generation timestamps to calculate the overuse or underuse of the bottleneck link.

RTP in 4G conversations One of the more recent uses of RTP is in conversations in mobile phones. It does not use multicast, only RTP for its real time data transportation purposes.

2.3 IP Multicast

In traditional unicast streaming systems, a separate stream is provided for each client. If the clients are watching the same content simultaneously, packets containing the same content are created and sent concurrently by the server to the participants, as opposed to *IP multicast* systems. In IP multicast systems, proposed by Deering (1989), a single stream sends packets, and the packets are duplicated in the network nodes along the way to the receiver.

IP Multicast is essentially a tree structure, as shown in Figure 2.1. Each relation represents a single packet sent, and the "x"s represent duplication. Duplication occurs when the network node has more than one child, e.g, if two or more participants on the same router watch the same stream. As for the direct connection between the server and the clients, the host naively sends packets one by one to a *Multicast address*. A multicast address is nothing more than an IP address, but the participants have to subscribe to it to receive the content.

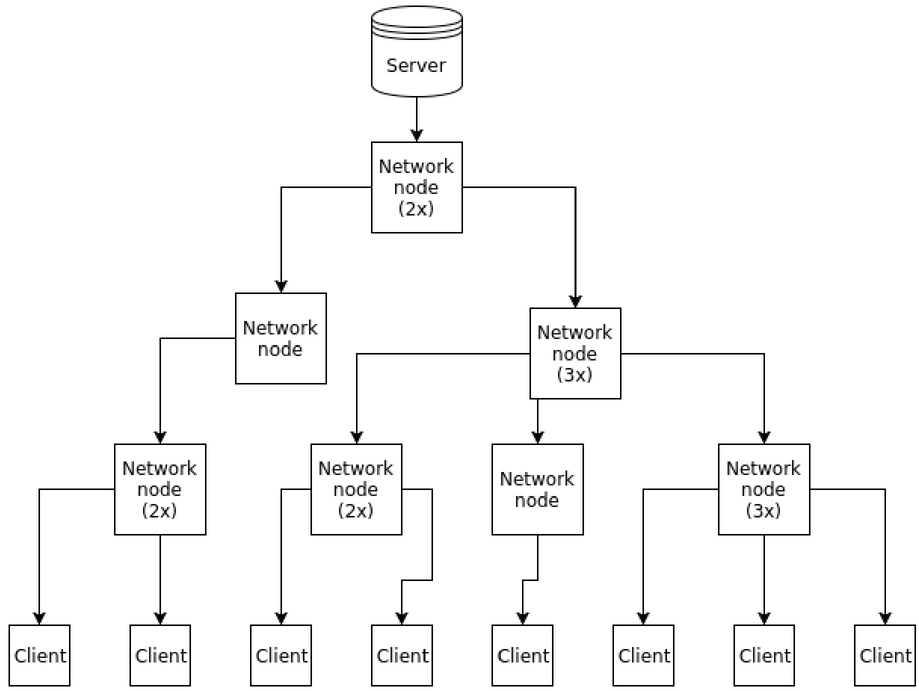

Figure 2.1: IP Multicast tree structure

When participants have subscribed to a multicast address, we define these participants as to be in a *Multicast group*. The participants in a multicast group can either communicate using a one-to-many or many-to-many approach. The many-to-many approach is crucial in a conference call, where every participant needs to send video and/or audio, and adapt to each other's network capacity. Note that IP multicast was originally only defined for many-to-many multicast. Protocol support for one-to-many has only recently been added as a standardized interpretation for some multicast address ranges.

2.3.1 Internet Group Management Protocol (IGMP)

IGMP by Holbrook et al. (2006) is a protocol used to establish multicast groups in IP networks. IGMP version 1 defines two message types: IGMP Query (general query) and IGMP report. It is transported via the IP protocol, where the protocol field is set to 2, which indicates IGMP content. A receiver sends am IGMP membership report to the router to imply that it is listening to multicast, also called IGMP join.

IGMP membership report

If a receiver would like to observe an audio stream it follows the following steps. The receiver begins by sending an IGMP packet with IP source address of the receivers machine, and the destination address as the multicast address. In the IGMP part of the packet, we set the group address to the multicast address we would like to listen to.

IGMP Query

When a receiver is finished listening to a multicast stream it is important to signal to the router that the receiver does not want more multicast traffic. If not, the receiver might be flooded and receive a reduced quality of service. However, the receiver cannot send a new membership report to unsubscribe, because there is no way to signal multicast rejection within a membership report. Instead, we use a IGMP query. A IGMP query is sent from the router in an IP packet with destination address 224.0.0.1, and 0.0.0.0 within IGMP. 224.0.0.1 indicates all multicast hosts address. Within this packet is the aforementioned query. If a receiver is still listening to the multicast traffic, the receiver will start a random timer. When this timer has elapsed, the receivers sends out a new membership report, to signal that it is still listening to the multicast traffic. The router sends queries out every 60 seconds to verify which receivers it should forward multicast to. A receiver exits the multicast stream by not responding. If no receivers respond the stream is cancelled by the host. This is done after three queries have been sent out without any answer.

IGMP Snooping

IGMP Snooping is a technique to limit which receivers multicast is forwarded to. The router initially cannot know what mac address to forward multicast to, because it uses a ip-table with unicast addresses to lookup receivers. IGMP Snooping uses a multicast lookup table in order to keep track of the multicast receivers. If IGMP snooping is now used, the sender has to flood every receiver with the multicast traffic because it does not know who to forward the traffic to, but using IGMP snooping the receivers can be filtered with the multicast lookup table.

2.4 Discussion and conclusions

In this section, we discussed how video streams are carried out today. Additionally, we discussed in depth RTP, multicast and wireless streaming. We learned that wireless multicast is possible to conduct, but it suffers from packet loss issues, due to a lack of link layer retransmissions. We, therefore, investigate FEC in RTP in the next chapter to investigate if it can be applied to wireless multicast streams.

Chapter 3

Generic Forward Error Correction in RTP

In this chapter, we present an explanation of how generic FEC is carried out in RTP. We begin in Section 3.1, by presenting an overview of the procedures used in RTP forward error correction. Next, in Section 3.2, we discuss the development of generic FEC standard in RTP. In Section 3.3 we discuss the most recent FEC draft in depth, because we implement and test this draft in this thesis. Finally, we conclude the chapter in Section 3.4.

3.1 Overview

As previously mentioned, FEC is a proactive mechanism for reducing packet loss in connections. The core idea of FEC is to transmit additional control data generated by the main stream. This control data can in turn be used to repair the stream if any of it is lost. In this section, we present a general overview of how generic FEC works in RTP.

Generic Forward Error Correction in RTP protects RTP *source* packets by generating FEC *repair* packets. These repair packets are encapsulated in regular RTP packets, and sent in separate RTP streams, parallel to the source stream. A receiver can use the repair packets to repair lost or corrupted source packets if any. The FEC error correction scheme is generic, because it is independent of the payload content. It simply protects RTP packets regardless of the packet content. Additionally, receivers that have not implemented FEC are still compatible with the source stream. In RTP, receivers ignore packets which it does not recognize, and thus receivers that have not implemented FEC ignore incoming FEC packets.

Simply put, the FEC protection mechanism applies the XOR operation on

```
+------------------------------+
|          IP Header           |
+------------------------------+
|       Transport Header       |
+------------------------------+
|          RTP Header          |
+------------------------------+ ---+
|          FEC Header          |    |
+------------------------------+    | RTP Payload
|       Repair "Payload"       |    |
+------------------------------+ ---+
```

Figure 3.1: FEC packet structure in RTP (Zanaty et al. 2018, pp. 12)

a set of source packets to generate one repair packet. If any of the source packets in this set is missing when the set arrives at the receiver, the missing information can be recovered by applying the XOR operation on the intact source packets and the repair packet. It is important to note that there can be a maximum of one lost packet when attempting to repair, including the repair packet. If not, the information cannot be retrieved from this packet set. The less FEC scheme the more packets can be repaired and more overhead.

At the receiver side of a RTP and FEC session, it is important to be able associate the RTP source packets with the FEC repair packets, in order to attempt to repair lost RTP packets, if any. This information is provided in the FEC header, typically by using a bitmask or non-interleaved and interleaved offsets from a base sequence number, also present in the FEC header.

3.2 History

In this section, we discuss how generic FEC in RTP has evolved over the years.

3.2.1 RFC 2733

RFC 2733 by Rosenberg and Schulzrinne (1999), is the first standard that defines protocol support for FEC in RTP. The motivation for adding FEC into RTP was that voice over IP technology caused packet loss which lead to low quality voice conversations. The packet loss issues could not be solved by retransmission of lost data, because retransmission increases latency, which is inappropriate for real-time communication.

In this version of FEC in RTP, each repair packet is associated with a set of source packets using a *sequence number base* and *bitmask*. The sequence number base is the RTP sequence number of the earliest source packet protected by this repair packet. The bitmask is a bit string indicating which source packets are protected and which are not. The bitmask is also called the offset mask, because it works as an offset to the sequence number base. In the bitmask, if a bit is set, it indicates that the packet with offset i from the sequence number base is protected. If it is not set, it indicates that the source packet at that offset is not protected. For example: if we have the bitmask *1010* with the base sequence number 1000, this FEC packet is protecting RTP the source packets with RTP sequence number 1000 and 1002. This FEC scheme can be used to protect any arbitrary pattern of packet association, including non-interleaved, interleaved protection, 2d parity, and staircase codes.

This RFC also describes several schemes. By protecting each packet with its neighbors, e.g. a with b and b with c, the stream can tolerate some burst loss of two packets. Another scheme defines that FEC can be sent even without a source stream, by repairing all packets continuously. This requires that all packets are protected together, meaning double overhead. Finally it describes a scheme which is similar to retransmission scheme, crazy stuff, 3x overhead.

The bitmask in the FEC header of RFC 2733 has a length of 24 bits, setting the maximum number of packets protected by one packet to 24 which makes it restricted for interleaved protection. Because every interleaved packet has a gap of L between each source packet the bitmask would have to be able to use 25 bits for associating a 5 by 5 cluster. In a 10 by 10 cluster the bit mask would have to be 100 bits. In addition to the limited bit mask, P, X, and CC fields in the RTP header were not consistent with the RTP header design. This did prevent payload-independent validity check of the RTP packets, because these bits were removed. This issue was later fixed in RFC 5109, which we discuss in Section 3.2.2.

Other problems, in the media packets, e.g. source packets, the CC, E and P must be set to 0. Also marker. "An implementation MAY copy these fields into the recovered packet from another media packet, if available." The E is for extending the FEC header. This is what is done in PRO MPEG.

3.2.2 RFC 5109

RFC 5109 is the second generic FEC standard by (Li 2007). The main motivation behind this RFC, was to fix inconsistencies in RFC 2733 and preserve bandwidth, e.g. reduce FEC overhead. ULP is used to preserve bandwidth. it adds extra protection to the front of the packet, e.g. the frame header.

In RFC 2733, the P, X, and CC fields were removed in the source RTP media packets. This is not consistent with the RTP design, because the bits were removed and only recoverable from another media packet, if it was available. It was also not possible to validate the media packets by using the header, because the bits were removed. This issue is fixed in this version of FEC in RTP, RFC 5109, by using a length recovery field in the FEC packet. This field is calculated by combining the aforementioned fields into one number carried by the FEC packet. This number can be reversed at the receiver side to re generate the original fields.

Uneven layer protection protects packets in different layers. Each layer has a protection length and bitmask and sn base. The bitmask describes which packets are protected by using a 1 if it is and 0 if not. This is used with the offset of the sn base. Additionally, if the L bit is set the packet uses a long bitmask which is 48 bit which can protect a maximum of 48 packets. The protection length indicates how much of a packet is protected in this fec layer. A fec packet can use multiple layers, but the layers can be completely independent. How does it know which part of the packet it is protecting?

There are some limitations to this scheme. First, by using multiple layers, there might be an issue with big packet overhead. Additionally, UDP packets are typically dropped if they are corrupted. This is due to UDP checksum, IP checksum and frame detection of packet loss. Therefore, the use of protecting different parts of a packet is limited.

3.2.3 SMPTE 2022-1

We derive information of SMPTE 2022-1 standard from Edwards (2017). The SMPTE 2022-1 Forward Error Correction scheme is used to provide non-interleaved and interleaved packet protection, e.g. 2d parity FEC. SMPTE 2022-1 is an extension of the aforementioned RFC 2733 FEC scheme.

It uses one FEC header which extends RFC 2733 header. In this header, the bitmask used in RFC 2733 is ignored, and therefore, 24 bits are not used. SMPTE 2022-1 uses an offset and number of packets associated field, for example 4, and 4. That means every fourth packet from the sequence number base, with a total of four packets. The downside is that this is extended on RFC which uses an bitmask. This field is not used in SMPTE, and is therefore wasted 24 bits. This version of FEC also need the offset and number of packets associated which uses one byte each.

There are some issues with SMPTE 2022-1. In the FEC header, as mentioned the bitmask is not used, thus creating overhead. Additionally, the header includes one field of sn base ext bits. This is the high bits of sequence number if needed. However, since RTP sequence number only requires 2 bytes, there is no need for this field, in RTP. This finally results in a header of 16 bytes.

3.3 draft-ietf-payload-flexible-fec-scheme-10

Zanaty et al. (2018) is currently working on a draft for a new generic FEC scheme. This scheme offers a payload format for interleaved and non interleaved FEC packets in addition to bitmask and retransmission support.

One of the main benefits is the new FEC packet with row and column. We can use 256 x 256 by only using 2 bytes of header. In the previous bitmask drafts we could only use bitmask, which needs the length, in bits, of L * D. For example, 5 L 5 D would need 25 bit bitmask, and 10 L 10 D would need 100 bit bitmask.

3.3.1 Schemes

This draft contains five schemes named: Bitmask, non-interleaved, interleaved, 2d parity, and retransmission. Note that 2d parity is a combination of the non-interleaved and interleaved scheme.

Bitmask

In this draft, the bitmask scheme uses a flexible bitmask. It has its own packet, the bitmask packet, see Figure 3.2. Firstly, a bitmask packet is identified by having 0 and 0 in the R and F fields of the FEC packet. The bitmask is flexible because it can either be 14, 45, or 109 bits in length. The minimum is 14 bits. There are two k flags in the packet which can be set to indicate a longer bitmask. For example if the first k bit is set, but not the second the bitmask has a length of 45 bits. As previously mentioned, the source packets protected by this packet is indicated using this bitmask and

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|0|P|X|  CC   |M| PT recovery |        length recovery        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          TS recovery                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           SN base_i           |k|          Mask [0-14]        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|k|                   Mask [15-45] (optional)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Mask [46-109] (optional)                  |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   ... next SN base and Mask for CSRC_i in CSRC list ...       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:               Repair "Payload" follows FEC Header             :
:                                                               :
```

Figure 3.2: FEC Bitmask Packet (Zanaty et al. 2018, pp. 15)

sequence number base. By using offsets from the sequence number base, each bit set to 1 indicates that a packet is protected and a 0 indicates that it is not. For example if the sequence number base is 1000 and the bitmask is 14 bits: 10101010010101, the packets 1000, 1002, 1004, 1006, 1009, 1011, and 1013 are protected.

Non-interleaved

Non-interleaved FEC protection is a mechanism where a row of packets is protected by a single FEC repair packet. The FEC repair packet is a xor product of the source packets in this row. The row length is determined by a value L. This scheme is efficient for protecting against random loss, because a single packet lost within a row can be repaired. This does, however, require that the row is short enough to handle the random losses. If multiple packets within a row is lost, the row cannot be repaired, by this scheme.

Interleaved

Interleaved FEC protection is a mechanism where a column of packets is protected by a single FEC repair packet. The FEC repair packet is a xor product of the source packets in this column. The column length

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|1|P|X|  CC   |M| PT recovery |         length recovery       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          TS recovery                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           SN base_i           |  L (columns)  |    D (rows)   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    ... next SN base and L/D for CSRC_i in CSRC list ...       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:               Repair "Payload" follows FEC Header             :
:                                                               :
```

Figure 3.3: FEC Parity Packet (Zanaty et al. 2018, pp. 16)

is determined by a value D. This scheme is efficient for protecting against burst loss, because multiple packets lost in a row are protected by a column of repair packets.

2d Parity

2d parity protection is combining row non-interleaved and interleaved packets to protect a stream against burst and random loss. We show the FEC header for the 2d parity packet in Figure 3.3. We indicate this packet by using 0 and 1 as R and F fields. In order to associate the source packets to this header we use a combination of the sequence number base and L (columns) and D (rows). We know that each matrix of source packets begin at the sequence number base. We can then associate the non-interleaved packets by sequence number base + i up to L. With the interleaved packets we can use sequence number base + i * D.

Retransmission

The FEC header for the retransmission scheme is shown in Figure 3.4. This header is the most simple of the three. In this packet we use one sequence number which is the sequence number of the source packet this repair packet is protecting. This packet is indicated by using the 0 and 1 for R and F fields. R indicating retransmission. This packet has the same layout as a regular RTP packet, including SSRC at the end. Therefore, this packet can simply be extracted without any complexity if it is needed.

```
 0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|0| P|X|  CC   |M| Payload Type|       Sequence Number        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              SSRC                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:          Retransmission "Payload" follows FEC Header          :
:                                                               :
```

Figure 3.4: FEC Parity Packet (Zanaty et al. 2018, pp. 18)

3.3.2 FEC Repair Packet Construction

We discuss step by step how we create one repair FEC packet from a set of RTP source packets. At this point we assume that we already have associated the appropriate source packets for the given scheme. We create a repair packet in three steps. We begin by calculating a bit string for each source packet. This bit string represents the header information as well as payload information of that source packet. Next, we combine these bit strings and finally generate a fully functional repair packet from these bit strings.

RTP Bit String

The bit string is 10 bytes long. We create the bit string by firstly extracting the first two bytes of the RTP header, which includes RTP version, padding flag, extension header flag, contributing source count, marker flag, and payload type. Next, we calculate a two byte representation of the length of the RTP packet. We do this by adding the RTP payload length, the list of contributing sources, the extension header, and RTP padding length. Next, we extract the timestamp. We then concatenate the first two bytes, the representation count, and the RTP timestamp. Additionally, we append the RTP payload at the tail of the bit string.

FEC Bit String

We calculate the RTP bit strings on a set of packets and apply the XOR operation on all of these bit strings. We begin by xoring the first and second packet, and the product of this xor is then applied to the next packet, and so forth. After this operation, we end up with the FEC Bit String. The

RTP packets are the packets one FEC packet is protecting. We use the FEC bit string to create the FEC packet.

FEC Header

First we calculate padding. We use the FEC Bit String to calculate the FEC header. We begin by skipping the first two bits in the FEC bit string. These bits contain the RTP version, but the first two bits of the FEC packet contains what kind of FEC packet it is. We therefore set this accordingly, e.g. 00 for flexible mask, 10 for non-interleaved and interleaved, and 01 for retransmission. We only support RTP version 2. Next, we copy the third bit into the P recovery field. Next, we copy the fourth bit into the X recovery bit. Next, we copy the following four bits into CC recovery field. Next, we copy the eight bit into the marker recovery. Next, we copy the following seven bits into the PT recovery. Next, we copy 16 bits into the length recovery. Next, we copy the 32 bits into the timestamp recovery. We write the lowest sequence number of the RTP packets into the base sequence number field. If mask is selected we set it with 1 if packet or 0 if not, if non-interleaved and interleaved, we set row and column fields, if retransmission, we don't do anything. Next, we copy the rest of the bit string.

3.3.3 FEC Packet Reconstruction

We can reconstruct one lost source packet from a set of source packets and one repair packet. Note that only one source packet can be lost, and the repair packet must be present. It is, however, possible to repair multiple lost packets in one set when using for example 2d parity matrixes. However, in this section, we discuss how to repair one source packet from a single set of source packets and one repair packet. This process effectively reverses the FEC packet construction, but firstly we discuss how we associate the repair packets with the repair packet. This is important, because the repair packet is sent in a different RTP session and is not directly linked to the set of source packets.

Associating Packets

We associate bitmask packets by collection packets with offset from base sequence number. If 1 in bitmask there is a packet, if 0 there is none. In non-interleaved and interleaved we collect the source and repair packets in a matrix with one repair packet for each row and column. To actually associate the packets, we must account for overflow. The RTP sequence number is an unsigned 16 bit integer. When it wraps around the max value of 65535 we have to account for that. We therefore find a sequence number by using sequencenumberbase + i * X_1 mod 65536. Note that we use 65536 instead of 65535 because the sequence number starts at 0. It is trivial to associate the retransmission packet because its sequence number is directly connected to the RTP packet, e.g. they share the same sequence number.

Recover RTP Source Packet

We recover one missing source RTP packet in a set of multiple source packets and one repair packet. We do this by firstly recovering the 10 byte bit string from the repair packet. Next, we extract the fields from this bit string. Finally we recover the source payload using the latter of the FEC packet.

Recover Source Packet Header

We recover the source packet header by firstly generating a 10 byte bit string for all the source packets. We do this by combining the first eight bytes of the RTP headers with the 2 bytes of length representation from the repair packet. We also need to calculate the bit string from the FEC packet. We do this by extracting the first 10 bytes of the FEC packet. Now, we apply the xor operation on all the bit strings from the source packets with the bit string of the repair packet. We now have a representation of the packet header. Next, we create a new RTP packet and set the version to 2, by setting the first two bits to 10. Note that this packet is empty, including payload. We skip the first two bytes of the bit string because this describes the FEC version, we do not need that. Next, we set padding in the next bit, extension in the next bit, CC in the next four bits, marker in the next bit, payload type from the next seven bits. We set sequence number to the sequence number. We get this by looking at the cluster. We skip 16 bits in the bit string. Next, we copy TS recovery into next four bytes. Next, we calculate variable Y which is the length representation, we collect this from the next 16 bits. We use this later. Finally set SSRC collected from source session. At this point the RTP header is recovered.

Recover Source Packet Payload

We recover the source packet payload by using the variable Y from the header recovery. This is the length, we therefore set the payload size to y. Note that this includes not only payload, but also extension, CSRC list etc. We allocate Y bytes of data for this payload. Next we extract Y bytes from byte 13 and onwards in all RTP packets present. We do the same for FEC payload starting after FEC header. We apply XOR operation on these payloads. Note that all payloads must be padded to Y length with zeros at the end. Finally we copy Y bytes into the RTP payload of the packet we are creating. There is no need to remove padding because we are only copying Y bytes.

Interative repair

In order to repair a cluster of non-interleaved and interleaved packets we use an iterative repair algorithm described in the FEC draft. As previously mentioned, we cannot recover more than one source packet from a set of source packets and one repair packet. It is therefore beneficial to run the repair over rows and columns iteratively, because a packet that is not recoverable per se can be recovered using a different row or column mechanism.

We use two variables: num_recovered_until_this_iteration and num_recovered_so_far. We init both to zero. Next we run the recovery first over rows and then over columns. If we successfully recover a packet on either scheme, we increase num_recovered_so_far by one. At the end, we compare num_recovered_until_this_iteration and num_recovered_so_far. If num_recovered_so_far is greater, we run the algorithm again, because the algorithm was able to recover packets, and thus more packets may be recoverable. We also set num_recovered_until_this_iteration equal to num_recovered_so_far to keep track of how many packets we have repaired. However, if num_recovered_until_this_iteration is greater or equal to num_recovered_so_far we terminate. This is because if no packets were recovered this iterations, no more iterations will recover any further.

3.4 Discussion and Conclusions

In this chapter we discussed how generic forward error correction in RTP works, and how it has evolved through the years. The most recent FEC draft yields benefits over the other FEC schemes, such as reduced overhead. To the best of our knowledge, there has not been conducted any tests using this draft to protect wireless multicast video streaming. We, therefore, in the following chapter, present a design where we test the effects of this draft.

Chapter 4

Design

Wireless multicast streaming suffers from packet loss due to wireless interference and lack of link layer retransmission; resulting in packet loss. In light of this problem, we investigate wether an implementation of the most recent Forward Error Correction (FEC) draft (Zanaty et al. 2018) reduces packet loss and can be performed with low latency and overhead. We determine this by streaming multimedia with FEC protection, and compare the packet loss before and after the reparation. In this chapter, we design these tests. Additionally, we perform preliminary tests and present the test environment used.

In Section 4.1, we discuss the hardware configuration we use for the experiment. In Section 4.2 we explore abnormalities with our test environment. In Section 4.3, we discuss how we perform a procedure to determine the effectivity of FEC. Finally, in Section 4.5, we conclude the chapter.

4.1 Hardware Configuration

To perform the multimedia streaming tests, we need several hardware components. We need a server machine to send the stream, client machines to receive the stream, and a network node to connect the machines. These requirements are fulfilled by *Simula Reaseach Laboratory* who provide hardware for this thesis. In this section, we discuss the benefits and limitations of this hardware.

4.1.1 Server

For multimedia streaming, we use one desktop machine. The machine has 8 GB memory, Intel Core i5-4590 CPU, and a 120 GB SSD. There is no PCI-E GPU and we install Ubuntu 18.04.1 LTS 64bit. Since the CPU is of generation Haswell (3) it supports Quick-Sync for H.264, but not for

H.265. This is relevant for the choice of multimedia encoding which we discuss in Section 4.3.1. The server is transparent to clients and thus the requirements for the server are loose. As long as it can stream multimedia at an acceptable speed it is sufficient enough for the main tests.

4.1.2 Client

For the client we use one Macbook Pro 2018 with 16GB ram. As we discuss later in this chapter, we use wired connectivity between the server and client for our experiments. The Macbook does not have a ethernet port, therefore we supplement this by utilizing a ASUS USB 2.0 to Ethernet Adapter.

4.1.3 Network Node

For network communication, we use one *Linksys WRT54GL Wi-Fi Wireless-G Broadband* router. The router has built in multicast support with several transmission rates ranging from *1 mbps* to *54 mbps*. However, as mentioned, we use only ethernet communication between the server and client. As of the sixth of June 2018, the most recent firmware for the router is *Ver4.30.18*. This version is from 2016, which we use with this router for our tests.

4.1.4 Server Client Communication

We show the configuration for our hardware setup in Figure 4.1. We connect the server machine and Linksys router using one *CAT 5e* ethernet cable. The cable is connected to the integrated motherboard LAN input, in the server machine. The Macbook client is also connected to the router using a *CAT 5e* ethernet cable. As mentioned, this ethernet cabled is connected via the ASUS USB 2.0 to Ethernet Adapter to the Macbook client.

4.2 Preliminary Testing

We conduct preliminary tests to find any abnormalities in our test environment. The first test is streaming H.264 video over unicast and multicast WI-FI with 1 minute samples. We conduct this test to investigate the packet loss distribution and wireless latency. Additionally, we conduct one test to replay a captured packet capture (pcap) file containing multicast video. We intend to replay a packet trace in our main experiment, therefore, it is important to explore any problems this might introduce.

Figure 4.1: Hardware setup
(Fosberg 2018)

4.2.1 Findings

In our first test we found that there was drift when comparing timestamps of the packets received. We realized that this is due to our system being without access to the Internet and therefore no clock synchronization occurred. We therefore conclude that it is not feasible to conduct latency monitoring for the transmissions in our experiments. We argue that wireless latency is not as significant as the potential latency introduced by our FEC implementation. We, therefore, limit our design to only monitor the latency of FEC by only monitoring the incoming source stream and outgoing repaired stream in our experiments.

In our second test, we replayed a pcap file containing H.264 encoded video over RTP multicast. We performed this test on one machine to capture the data. When replaying the stream, we quickly realized that the packets were not showing up. We realized, that this was due to multicast not forwarding the packets to the same address. By instead replaying the capture over LAN the packets were showing up at the client. We therefore conclude that our experiment must be between two machines, one hosting and one receiving.

4.3 Experiment Design

4.3.1 Multimedia

Today, the streams conducted in the football stadium live stream contains video and audio. Video demands the most throughput of the two and we therefore decided to only concern ourselves with video for the main experiment. Additionally we discarded audio to reduce complexity in the network transmission. With us choosing video, we therefore discuss how we prepare a video for transportation and what tools we use for encoding and compression. As discussed in Section 4.2 we are restricted to certain data transmission rates. It is therefore important to calculate the throughput requirement for the video so we can choose the appropriate transmission rate.

We have defined requirements for video which are presented here:

- The video format must be standardized as a payload format in RTP and must also be decodable on the major mobile platforms, e.g. *iOS* and *Android*.
- The video encoding must be optimized for lossy network transmissions.
- The video compression should be optimal for user experience versus bandwidth consumption.
- The video file source should be picked from reliable sources.

Format

The newest and most efficient encodings as of 2018 are MPEG-4 Part 10 H.264/AVC and MPEG-H Part 2 H.265/HEVC. RTP packing for both is standardized in RFC 6184 (Y.-K. Wang et al. 2011) and RFC 7798 (Y.-K. Wang et al. 2016), respectively. The main difference between the two is that while H.265 reduces the file size by a higher factor than H.264, it also requires more processing power, and in some cases a newer generation CPU for hardware acceleration. Additionally H.264 is the current video standard, which means that most applications support it.

There are also a variety of video containers to hold the H.264 or H.265 video frames such as MP4 and MKV. Video containers provide metadata which simplifies synchronizing of video and audio streams and playback options such as rewinding or fast forwarding. Not using a container also increases complexity because indexes have to be calculated on the go. In this case each NAL unit can be packetized directly into RTP payloads which they are optimized for. This works well because it supports different requirements from low to high bitrates.

Although H.265 is to be the standard in the future, it remains to be deployed to major applications. We therefore decided to use H.264 video for testing in the main experiment. Additionally we decided not to encapsulate the stream in a video container to take full advantage of direct packing of NAL units described in RFC 6184.

Encoding

NAL Unit Size If a NAL unit is not regulated, we could get NAL units which are bigger in size than MTU. RTP handles this by fragmenting the NAL units and putting them in different packets. Since the header information for the NAL unit is placed first in the NAL unit, if the first packet is lost we have no metadata for the rest of the frame. The frame will then be dropped. This is called IP fragmentation. We can avoid this by setting a max slice size when encoding the file. This will force the NAL units to be less than that size. The value chosen, also include overhead for the NAL Unit header.

The max transfer unit (MTU) is the maximum amount of bytes which can be carried in a link layer frame. MTU is most commonly set to 1500, therefore we will use 1500 as the MTU in our calculations. In order to find the maximum size for each NAL unit, we must subtract the MTU by packet overhead. As mentioned, MTU is the size inside a link layer frame which means we must take in account the network layer and inwards. The IP header is the first header of each packet. We ignore the options because we are not using it. This leaves us with an IP header of 20 bytes. Next the UDP header is always 8 bytes. Next, there is the RTP header, and because it is single source and no extension header is involved, the header is 12 bytes. This leaves us with an overhead of 40 bytes per packet.

However, we also need to account for FEC overhead. The protected RTP packet is placed in the payload of a FEC packet. Therefore the NAL Unit cannot exceed this size as well. As previously mentioned, we exclusively use the FEC interleaved and noninterleaved scheme. Therefore, the size of a FEC packet is 12, because a interleaved and non interleaved packet is of this size. Additionally, we do not have any contributing sourcers nor extension header. Finally the FEC packet is also incapsulated in an additional RTP header, which adds another 12 bytes. This leaves us with a total overhead of 64 bytes, which makes the max size of each NAL unit to be *1436 bytes*.

Compression

The most common methods of compressing video is to either use constant rate factor *crf* or quantization parameter *qp*. Constant rate factor adjust

```
ffmpeg -i tractor_1080p25.y4m -c:v libx264
     -crf 25 -slice-max-size 1432 output.264
```

Figure 4.2: Encoding of the tractor sample

the bitrate to a average defined by a parameter. When there is more motion, the bitrate is increased and when little motion the bitrate it is decreased. This results in a file with variable quality and average bitrate. On the contrary, quantization parameter applies a constant quantization rate to all the frames. This results in a file with a variable frame rate with constant quality.

We decided to use constant rate factor when encoding our video file. This setting is more commonly used than qp, and we therefore use it to provide a realistic video stream.

Sample File

As mentioned in Section 4.3.1, we decide to use video for the main experiment and exclude audio. Xiph (2018) offers several free and raw video samples with a wide range of resolutions. Based on this and the aforementioned requirements, we decide to use the sample video called *Tractor* because it has our desired resolution of 1080p. Anyhow, the file might not be too representative for football matches because it is not a sequence from a football stadium. We do, however, not have the time nor resources to test wether or not there are significant differences amongst different video files. Therefore, we focus only the *Tractor* sample.

Bandwidth

4.4 FEC Capture Procedure

In this section, we design a procedure to capture data from various packet loss affected video streams, repaired by our FEC implementation. This design is, in other words, a fully functional step by step algorithm of how we capture this data. We aim to make our result as reliable as possible by using a single stream pattern from a H.264 video file. We apply different FEC configurations to this stream pattern and save these to different trace files. Next, we apply packet loss to these files collected from the literature to simulate wireless packet loss. Finally, we play the trace files over a LAN and attempt to repair them at the receiver side of the connection. We can

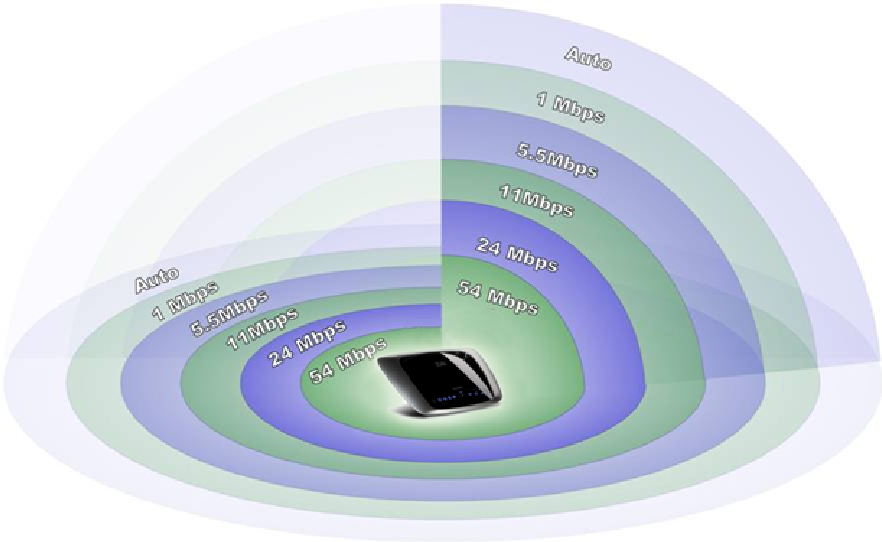

Figure 4.3: Wireless transmission range
(Linksys 2015)

therefore know how well our FEC implementation works for this specific video, with these FEC configurations, at this packet loss.

4.4.1 Loss Pattern

The first thing we do is to establish a loss pattern for our wireless multicast stream. Tang and McKinley (2003) made a model for modeling packet loss in multicast WLANs. In our case we only have one client, and thus a simpler model is sufficient. We therefore use the Gilbert-model, also known as simplified Gilbert Elliot model.

The Gilbert Model is a very simple model which can be used to generate packet loss in trace files. Each packet is put in a state. There are two states, a good and a bad state. When a packet is in the good state, no packets are lost. In the bad state packets are lost. There is also a transitional property between the two states. By doing this we can calculate the state for loss and not very simply and end up with two probability values.

Mochnáč et al. (2010) explored the packet loss rates in wired and wireless networks. They found that burst loss was more common in wireless networks. They present the following Gilbert values: 0.838026 and 0.161974. They also reported observing a maximum of 15 packets lost in a burst and the most common burst to be two and four. We use these loss rates in our experiment because they simulate wireless networks using RTP transported over UDP.

4.4.2 FEC Configuration

We were unable to find any recommendation for appropriate FEC L and D values to utilize in wireless multicast. Previous test have been done by Westerlund (2015), but the chosen L and D values were not explained. We, therefore, find it necessary to simulate different L and D values for a given loss pattern. However, the theoretical maximum combinations of L and D values are $256 * 256$, because both L and D are unsigned one byte fields in the 2d parity FEC header. A test involving all these combinations would not be possible within our time limit. A higher product of L and D results in a bigger repair window, because we have to wait for the packets to arrive at the receiver. We, therefore, in this subsection, explore constraints to narrow down the simulation requirement.

Narrow down

To narrow this down we begin by setting a limit of delay tolerance and overhead cost. We allow a maximum of 500 milliseconds of delay. Note that as we stream our main experiment over cable. We consider the wireless delay irrelevant and therefore only concern ourselves with the delay introduced by FEC. This means that our repair window can be at max 500ms. We argue that 500ms is tolerable because it is faster than the current DASH implementation. We do not know the current segment size of DASH in the live stream, we do however know that it is in the seconds range. We calculate the FEC repair window by dividing the amounts of packets in the FEC matrix by the packets produced by the RTP source and FEC stream each second (pps). Additionally, we multiply it by a factor of 1000 to convert it from seconds to milliseconds.

$$RepairWindow = \frac{fecSize}{pps} * 1000$$

In addition to the cap of 500ms delay, we also introduce a cap of 66.67% overhead, meaning that our FEC configuration must not exceed this amount of overhead. At any higher rate than 66.67% the effect of FEC would seize to exist because it has the same overhead as retransmitting packets. We learn the following overhead calculation from the most recent FEC draft (Zanaty et al. 2018):

$$Overhead = \frac{1}{L} + \frac{1}{D}$$

In addition to cap the repair window and overhead, we can further narrow down our L and D values by analyzing the loss pattern we decided to use in Section 4.4.1. In 2d parity FEC, burst loss is protected by interleaved packets. However, if the burst would be longer than our L value, the repair algorithm would fail, because a burst larger than L would occupy more than one row. We can therefore calculate the longest feasible L value by analyzing the P11 value. Mochnáč et al. (2010), noted that the maximum burst length they observed on wireless transmissions were 15. To cover our basis, we decided to limit our L value to 20. The probability of a 20 packet burst loss is 0.161974^{20}, which is so tiny that a burst of size is infeasible.

4.5 Discussion and Conclusions

In this chapter, we presented the experiment we conduct in this thesis, based on literature and preliminary testing. In summation, the algorithm to reproduce the experiment is as follows:

- Encode the video file.
- Stream the video file and capture the packets to a pcap file. This pcap file is used to determine FEC repair window.
- Stream the video file with all FEC configurations which pass our restrictions in regards to overhead and latency. Capture this data to a pcap file.
- Split the file above into separate pcap files for each of the fec configurations.
- Apply packet loss to all of these separate pcap files, and exclude the first cluster to optimize for our implementation.
- Stream the lossy pcap files over LAN and attemp to repair them at the receiver side.

Next, in the following chapter, we discuss the FEC implementation we use for the experiment above.

Chapter 5

Implementation

In this chapter, we present an implementation of the most recent FEC draft to measure the benefit of the FEC repair scheme, and also the disadvantage of introduced latency. This implementation consists of FEC encoding and decoding operations, as well as a fully operational FEC protected streamer and receiver. We intend to utilize this implementation in the main experiment, described in Chapter 4, by generating FEC repair packets from a source media stream, as well as combining the source and repair stream to attempt to repair the source stream. We only intend to use the 2D parity scheme for our main experiment and thus we present an implementation of this scheme only. It is nonetheless trivial to extend our implementation to support the bit mask and retransmission schemes, in potential future work.

In Section 5.1, we provide an overview of the Live555 Streaming Media framework. In Section 5.2, we discuss our applications for streaming and receiving FEC protected RTP streams. These applications make use of our FEC extension. In Section 5.3, we discuss the FEC components we have made for our FEC extension. In Section 5.4, we discuss the core FEC operations for our FEC extension. Section 5.5 presents the limitations with our FEC extension. Finally, in Section 5.6, we summarize and conclude this chapter.

5.1 Live555 Streaming Media

Live555 Streaming Media is a well-maintained collection of C++ libraries for streaming multimedia over RTP unicast and multicast in IP networks. In addition to streaming multimedia, it also offers playback options and metadata negotiation using protocols such as RTSP and SIP, and QoS monitoring via RTCP. Furthermore, Live555 is also used in the wild by well-known applications such as VideoLan and MPlayer. In addition to providing a complete RTP stack, Live555 also offers several schemes for packing video, audio and text in RTP. These schemes are defined in Internet Standard documents such as RFCs, for example RFC 6184 (Y.-K. Wang et al. 2011).

In this thesis, we implement the most recent FEC draft using *Live555 Streaming Media* as our base RTP-implementation, primarily because it offers most of what we need to conduct the experiments. This includes a complete RTP/RTCP stack and H.264 video streaming and receiving applications. Finlayson (2018), the CEO and founder of Live Networks Inc., recommends to extend Live555 via C++ subclassing rather than changing the existing codebase. One major benefit of this method is that we face no obligation under the Live555's current license, LGPL. As a result, we follow this recommendation, which naturally results in our implementation being created in C++.

Live555 supports all major operating systems, Windows, MacOS and Linux. We do, however, only have access to Mac and Linux machines. These operating systems are very similar because they are both based on the UNIX architecture. As a result, we develop the FEC implementation to support these platforms, and do not focus on Windows. On MacOS, we support the *clang* compiler, and on Linux, we support the *gcc* compiler.

5.1.1 Flow of Operation

Live555 is an event-based C++ library running an event loop in a single-threaded task scheduler, eliminating the risk of race conditions and complexity of multi-threadedness. In this loop, the task scheduler looks for available tasks, either in the delayed queue, or the list of network read handlers. When a task is found, it is executed. In addition to delayed tasks and network read handlers, tasks can also be set by using watch variables or event triggers. These event handlers are applied whenever manual event triggering is needed.

Live555 utilizes three core components: *sources*, *filters*, and *sinks*. A filter is actually a source receiving data from another source, i.e., filtering or manipulating the data in some way. Media streams in Live555 are constructed by combining components into a pipeline. Each pipeline consists of primarily one sink, one source, and an optional set of filters. We provide a visual example of streaming H.264 video in live, in Figure 5.1.

The sink is the initiator of each pipeline. A sink requests a frame of data and sets off a chain of events, eventually retrieving the frame from the source. On the sender side, when a source is requested by a sink to fetch

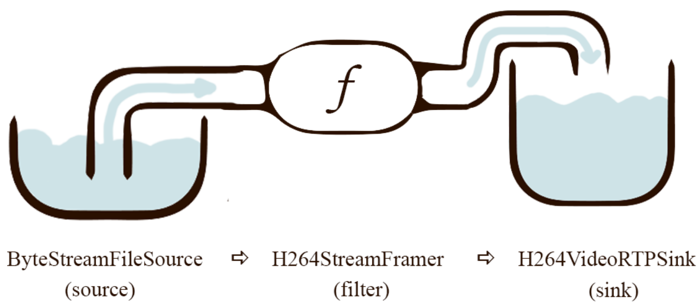

Figure 5.1: Live example (Fosberg 2018)

data, the source schedules a background task to fetch this data from a media resource. When the data is ready, the source fetches data and packs it into a frame, appropriate for the media resource type (i.e., H.264). When the frame is fetched, it is instantly delivered to the sink. The sink then encapsulates this frame into an RTP packet, and sends the packet over a network via a socket. On the receiver side, the same logic applies. However, the source fetches network packets when they are available using a socket. The source extracts the network packet payload and deliver this data as a frame to the sink. The sink then uses this data for different applications, such as playback or file recording.

Every source in Live555 is a child of the *FramedSource* class. This class has an important virtual method called *doGetNextFrame*. This method is implemented in the parent FramedSource class method *getNextFrame*. In a child class of FramedSource, this method is overridden to deliver a frame of appropriate data. Then either a filter or a sink calls the *getNextFrame* of its connected source. Upon execution, we register a *afterGetting* function in the source class. When the source has finished fetching the frame, it calls the *afterGetting* function which delivers the frame to the filter or sink.

5.2 Streaming Applications

In this section, we discuss our applications for streaming and receiving FEC protected raw H.264 video over RTP. We utilize the *testH264VideoStreamer* and *testMP3Receiver* applications provided within Live555s *testProgs* directory. We use these programs as a starting point to develop our streaming and receiving applications. With the concepts defined in these applications in mind, we add the components from our FEC extension accordingly. We discuss our FEC extension in detail in Section 5.3.

There are several similarities in the sender and receiver applications. Firstly, they both use one *main* and one *afterPlaying* procedure. The main function is the startup point of the program. In this function we do all initiation for the streams. In the afterPlaying function we clear up the streaming components and make ready for a new stream to be conducted.

- In the *main* procedure, we firstly create a new instance of the *BasicTaskScheduler* class. Additionally, we create a new *BasicUsageEnvironment* while passing in the *BasicTaskScheduler* instance. We can now assign the same usage environment to every component in our application. Second, we setup appropriate instances for the *Groupsock* class, assigning port number 18888 to the source stream. For every protection stream we increment the host port number by two. This is in case we want to use RTCP for any of the streams, because we assign RTCP port number one more than the original stream. Third, we initiate the sources and the sinks. We connect each sink to its appropriate source by passing the sink instance into the source's constructor.

- In the *afterPlaying* procedure we stop playing all the sinks, and close all sources. We pass this function into each sink, so that sink can call this function when it is done streaming.

5.2.1 Sender Application

Our sender application is called *testH264Video2DFECStreamer*. We use one instance of *ByteStreamFileSource*, *H264VideoStreamFramer*, and *H264VideoRTPSink* to read a H.264 file from disk, packetize the received data into frames and encapsulate the frames into RTP packets. All of these classes are provided by Live555. Additionally, we use two sources from our FEC implementation: *FECNonInterleavedSource* and *FECInterleavedSource*. We push the RTP packets produced by the *H.264 Sink* into these sources in order to generate FEC repair packets. These FEC sources are in turn connected to two *SimpleRTPSink* classes. These sinks packetize the repair packets into RTP packets and transmit them over the network. We show the flow of operation for the FEC sources in Figure 5.2.

We copy packets from the *H.264 Sink* into the *FEC Sources* by using an additional class from our FEC extension called *FECGroupsock*. *FECGroupsock* is a child class of the aforementioned *Groupsock* class, and thus we can pass it into to any sink using a *Groupsock* instance. *Groupsock* was intended to be Live555's implementation of multicast sockets but it supports unicast as well. The *Groupsock* handles both network reads and writes using the virtual methods *handleRead* and *output*. Both methods return a boolean result on wether the read or write was successful. In our *FECGroupsock* class, we override the *output* function and copy the incoming packets directly to our *FECNonInterleavedSource* and *FECInterleavedSource*. We save an instance of these fec sources as fields in the *FECGroupsock* class. We apply our *FECGroupsock* to our H.264 sink and regular groupsocks to our FEC sinks, as we only want to protect the H.264 stream. We use use 255 ttl to prevent the packet from being tossed.

In addition to the *H.264 sink*, we setup sinks for both FEC sources. We use the SimpleRTPSink provided by the Live555 framework. We have to use separate sinks because the payload format in Live555 is set by the sink. Therefore, since *FECNonInterleavedSource* and *FECInterleavedSource* need different payload types, we use two sinks. FEC packets are sent in RTP packets, which adds an additional RTP header. Live555 has a max size of 1448 for the RTP payload. We exceed this limit when we create FEC packets, because we append an additional 12 bytes of FEC header. We resolve this by changing the payload sizes of the H.264 sink. We change the MTU by calling the function *setPacketSizes* with the values *1444* and *1444*, which sets the preferred and max size, respectively. Note that Live555 supports *jumbo frames*. However, since most routers use 1500 MTU, we also apply it to our implementation.

Receiver Application

Our receiver application is called *testH264Video2DFECReceiver*. The core idea of this application, is to receive, multiplex, repair (if needed) and forward incoming source and repair packets. We use three instances of the Live555 provided class *BasicUDPSource*; one to receive source packets, and two to receive noninterleaved and interleaved repair packets. We cannot use any *RTP Sources* at this point, because we need complete RTP packets in order to apply the FEC scheme. Using a *RTP Source* would remove the RTP header. Next, we multiplex the source and repair packets into a custom *FEC2DParityMultiplexor* class. We achieve this by using another class

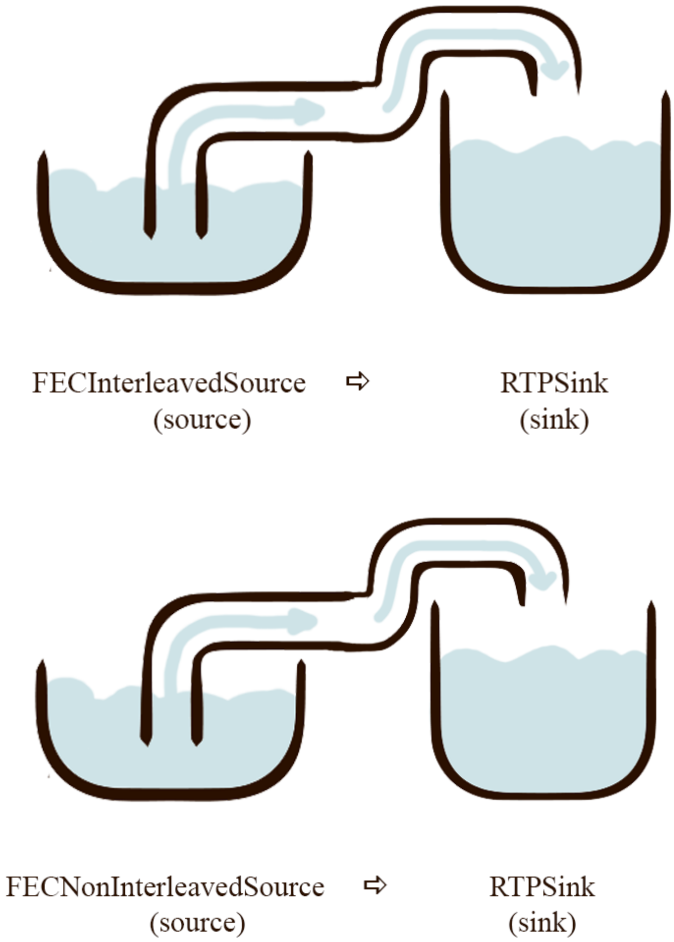

Figure 5.2: FEC interleaved and non-interleaved sender (Fosberg 2018)

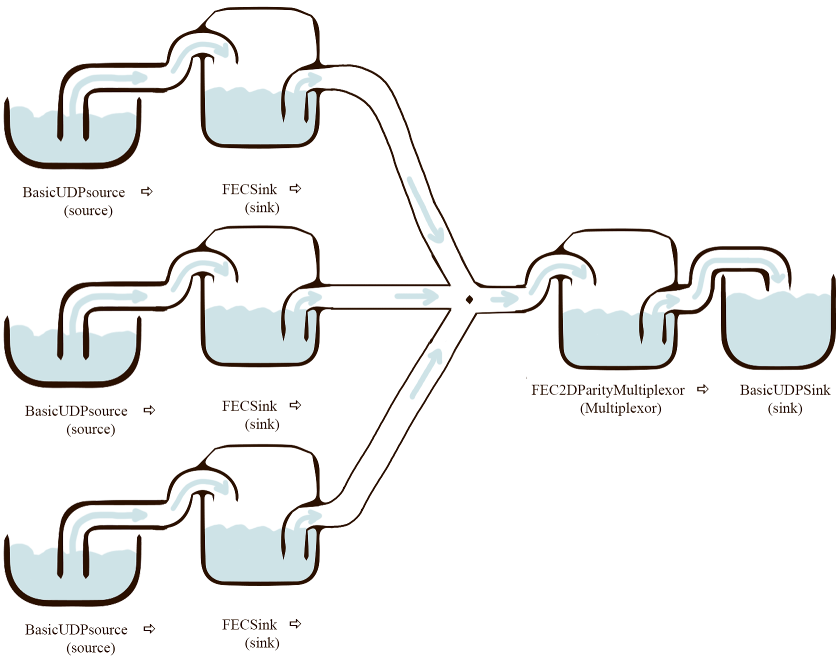

Figure 5.3: FEC receiver
(Fosberg 2018)

from our FEC extension called *FECSink*. We utilize this sink by connecting one instance to the noninterleaved source and one to interleaved source, and then pushing the repair packets into our *FEC2DParityMultiplexor*. We use an additional *FECSink* connected to our source stream to also push the source packets into our *FEC2DParityMultiplexor*. Finally, we connect one instance of the *BasicUDPSink* to the *FEC2DParityMultiplexor* and forward the packets to localhost. This can in turn either be decoded in video applications or recorded for further analysis.

5.3 FEC Components

In this section, we discuss the components of our FEC extension of Live555 Streaming Media. In Live555, a FramedSource is requested by a sink or source to fetch data from some resource and deliver frames to a corresponding filter or sink. In our scenario, we envision every RTP packet we wish to protect as a *frame*. We, therefore, use our custom FEC sources to push RTP packets, packed into FEC packets, as frames to an RTP sink.

The receiver side of the connection uses several Live555 provided classes and some custom classes. We use pre build *BasicUDPSource* classes to receive source and repair packets. Because the entire RTP packet is needed to rebuild lost packets, we cannot remove any RTP header at this point; and thus no RTPSource object. Instead, we use custom FEC sinks connected to the UDP sources and push the packets into a custom FEC multiplexor. In the multiplexor we aligning packets in clusters and repair lost packets, if any. The computed source packets are at last forwarded to a local address.

5.3.1 Sender Side

FEC Groupsock

As mentioned previously, we need complete RTP *source packets* to generate appropriate FEC *repair packets*. In our scenario, we use a *H264VideoStreamFramer* to encapsulate H.264 frames in RTP packets and a *H264VideoRTPSink* to send source packets over the network using an instance of the *Groupsock* class. All of these classes are included in the Live555 framework. The last chain in creation of RTP packets is a sink. Therefore, we cannot use this sinks source to request frames with an additional sink, because the packets are not complete at this point in time. Instead, we extend the *Groupsock* class and make copies of the packets that are passed through. We push these packets to custom FEC sources which we discuss in below. Because every source uses an instance of the *Groupsock* class to transfer data over a network, this extension works regardless of the media type.

FECSource

We use two FEC source classes to create non interleaved and interleaved FEC repair packets. These classes are called *FECNonInterleavedSource* and *FECInterleavedSource*, respectively. In the most recent FEC draft it is stated that the non-interleaved and interleaved source packets should have different payload formats to distinguish them from another. In Live555, the payload format is set in the sink, and there is only sink per pipeline endpoint. Therefore, we cannot use a single source to generate both non interleaved and interleaved FEC repair packets.

The main responsibility of these classes is to receive source packets and place them in a packet array, with the intention of generating repair packets. When a sufficient amount of source packets have been placed in the array, we generate repair packets for that sequence of source packets. When the repair packets are complete, they are placed in a queue, and the temporary source packets are deleted. When these operations have concluded, we perform an event-trigger. When this event-trigger, triggers, we deliver a repair packet to the connected sink.

There are a few differences between *FECNonInterleavedSource* and *FECInterleavedSource*. In *FECNonInterleavedSource* the array of source packets is the size of D. Note that the constants D and L are the row and column length of the source packets, respectively. Since the packets are generated for each row, there is no need to hold more than D source packets. In *FECInterleavedSource* we maintain an array of D * L size. This is because the packets are protected by column and we must therefore hold all packets until the array is filled. Additionally, when we generate repair packets, we only do this once for *FECNonInterleavedSource*, because only one row is done at once. In *FECInterleavedSource* we perform the operation L times because we need to protect each column.

For both *FECNonInterleavedSource* and *FECInterleavedSource* we use the same *FECEncoder* function: *repairRow*. This function returns a new FEC repair packet generated by a row of source packets sent in as a parameter. Therefore, in *FECInterleavedSource*, as we have columns not rows, we extract a column of packets and convert them to a row. We send this row into the *repairRow* procedure. Next, we push this FEC packet into the FEC Packet queue and clear out the temporary RTP array. At last, we trigger an event via the event-trigger. This event signals to the task scheduler that there is data ready in the FEC packet queue.

5.3.2 Receiver Side

FEC Sink

We receive the source and repair packets in multiple sources, and therefore, we use several FECSinks to request these packets from the corresponding source and forward them to our *FEC2DParityMultiplexor*. We were not able to connect our *FEC2DParityMultiplexor* directly to the receiving sources, because multiple calls to the getNextFrame method in FramedSource are not allowed in Live555. Multiple calls to getNextFrame are not allowed because it causes race conditions. However, connecting the *FEC2DParityMultiplexor* and the receiving sources might be possible in future work.

FEC2DParityMultiplexor

FEC FEC2DParityMultiplexor is a class which attempts to repair RTP source packets, if any are lost. This implementation is specifically for repairing matrixes of RTP source packets, protected by non-interleaved and interleaved FEC repair packets. We apply some of the base ideas as we did in Section 5.3.1. We use a queue of RTP packets to hold our outgoing repaired packets. When there are available repaired packets, we use an event-trigger to signal to the task scheduler that there are available packets. When the task scheduler executes this task, the packets are delivered to the appropriate sink.

As mentioned in Section 5.3.2, packets are pushed into this class using a callback procedure. Within this procedure, we forward both source and repair packets to into appropriate *FECClusters*. We discuss *FECCluster* in detail in Section 5.3.2. In short terms, *FECCluster* is a matrix containing the source and repair packets for interleaved and non interleaved protection. In *FEC FEC2DParityMultiplexor* we maintain a vector of these clusters. When packets arrive, we search through this vector for a corresponding cluster for the current packet. If no cluster is found, we create a new one and insert the packet. Additionally, in order to know which cluster we should insert an packet into, we hold a variable with the current sequence number base. We update this base when a new cluster is created.

When *FEC2DParityMultiplexor* is instantiated we start a delayed task which executes every *20ms*. Note that this number is completely arbitrary, as we do not know if we should schedule it more or less frequent. It is, nonetheless a low cost operation. In this task we loop through the vector of FECClusters to determine if either all source packets have arrived intact, or the repair window of a cluster has expired. If a repair window has expired and not all source packets have arrived, we attempt to repair the cluster. We use the *repairCluster* procedure, defined in Section 5.4.2. When we have completed repairing the packets, or all the packets did arrive intact, we push the packets into the aforementioned queue of ready RTP packets. When we push ready packets into the queue, we also trigger the event-trigger to signal to the task scheduler that the packets are ready for delivery.

The RTP sequence number is chosen at random and is an unsigned 16 bit integer. When the sequence number reaches the maximum number for 16 bits (65535) the sequence number overflows and reverts back to 0. This causes problems if the sequence number overflows in the middle of our cluster. Therefore, we compute the potential next sequence number by adding the cluster size to the current sequence number. If the potential new sequence number is smaller than the current sequence number, we have a special case. Figure 5.4 shows how we handle this scenario. Note that

```
if (newSeq < currentSequenceNumber) { /*Special case*/
    if (seq >= currentSequenceNumber || seq < newSeq) { /*Find cluster*/
        FECCluster* fecCluster = findCluster(seq);
        if (fecCluster != NULL) fecCluster->insertPacket(rtpPacket);
    }
    else { /*Make cluster*/
        u_int16_t diff = seq - currentSequenceNumber;
              if (diff > 30000) { /*Arbitrary failsafe.*/ }
        else {
                updateCurrentSequenceNumber(seq, sourcePacketCount);
                FECCluster* fecCluster =
                            FECCluster::createNew(currentSequenceNumber,
                    fRow, fColumn);
        fecCluster->insertPacket(rtpPacket);
        superBuffer.push_back(fecCluster);
      }
    }
}
```

Figure 5.4: Special case for RTP sequence number overflow

we also use an arbitrary failsafe. We do this in case a packet arrives late within the sequence number wraparound at 65535.

FECCluster The FECCluster class holds an array of RTP packets (source and repair), the size of this array and a timestamp of when the cluster was created, e.g. when the first packet was inserted into the cluster. As mentioned previously, when we repair the clusters in *FEC2DParityMultiplexor* we check wether the repair window has expired or not. We calculate this by comparing the difference between the timestamp of the cluster and the repair window. We also include a procedure to insert a packet at the appropriate index within the cluster. In this procedure, we differentiate between source and different repair packets by checking the payload type. We also account for the previously mentioned overflow by using 16 bit unsigned variables and whole number division. Figure 5.5 shows this scenario in detail.

RTPPacket and FECPacket

We use two container classes for holding RTP and FEC packets. These classes only hold a size and a buffer. The buffer contains the entire packet as bytes and the size is the volume of the buffer. While it is technically unnecessary to separate this logic into two classes, we still do it to make the code more readable.

```
void FECCluster::insertPacket(RTPPacket* rtpPacket) {
    int index = getIndex(rtpPacket);
    fRTPPackets[index] = rtpPacket;
}

int FECCluster::getIndex(RTPPacket* rtpPacket) {
        int payload = EXTRACT_BIT_RANGE(0, 7, rtpPacket->content()[1]);
        if (payload == 115) { /*Non-Interleaved*/
                u_int16_t base = (((u_int16_t)rtpPacket->content()[20]) << 8) | rtpPacket->content()[21];
        u_int16_t prelimIndex = base - fBase;
                return prelimIndex / fColumn * (fColumn + 1) + fColumn;
        }
        else if (payload == 116) { /*Interleaved*/
                u_int16_t columnBase = (((u_int16_t)rtpPacket->content()[20]) << 8) | rtpPacket->content()[21];
        u_int16_t prelimIndex = columnBase - fBase;
                return prelimIndex + fRow * (fColumn + 1);
        }
        else { /*Source*/
                u_int16_t seqNum = (((u_int16_t)rtpPacket->content()[2]) << 8) | rtpPacket->content()[3];
                u_int16_t prelimIndex = seqNum - fBase;
                return prelimIndex + prelimIndex / fColumn;
        }
}
```

Figure 5.5: Insert packet in FEC cluster

5.4 FEC Encoding & Decoding

In this section, we present our implementation of the core FEC Encoding & Decoding operations, in the most recent FEC draft. We separate these two classes so that the sender only needs to import the FECEncoder and the receiver only needs to import the FECDecoder. These classes are really not classes, as they have no constructor, only static methods. We only use static methods, because there is no need for state. These collections of operations only executes one operation at a time, without internal side-effects.

5.4.1 FECEncoder

Our FECEncoder class consists of one public method: *protectRow*. As mentioned previously, this function generates a FEC packet by combining a row of RTP packets. We use four additional helper methods to aid this operation. These are called *calculateLongestPayload*, *generateBitString*, *getPaddedRTPPayload*, and *createFECPacket*. We present these in the following subsection.

protectRow

The protect row function takes an array of RTP packets (with size) and returns a newly created FEC packet. The idea is that a FEC packet will be computed regardless of the amount of RTP packets. We can then use this logic for both the FEC non-interleaved and interleaved protection schemes, and also others schemes. We begin by finding the longest payload of in the source packet array, by utilizing our *calculateLongestPayload* method. We need this size to apply optional padding to packets with shorter sizes. This is because the FEC repair scheme requires each packet to be of the same length when we generate the final FEC packet. Next we compute the bit string for every packet in the source packet array, using the *generateBitString* procedure. This bit string is a representation of the source packet which is used to generate the FEC header.

When we have the bit string representation of all the source packets, we apply the xor operation on every source packet. This includes both the bit strings and source packet payloads. This is the base of the FEC repair scheme, as this process can be reversed to generate any of the source packets, if missing. Before we combine the payloads we pad them to the longest packet size of the source packets, using the *getPaddedRTPPayload* method. This is necessary to match the length of each payload when we apply the xor operation. After this computation, we end up with one *FEC bit string* and one *FEC payload*. Next, we extract the sequence number of the first packet in the array because we include this in the FEC packet to indicate the base sequence number this FEC packet is protecting. Finally, we create a FEC packet by combining the FEC bit string, packet size, FEC payload, sequence number base, l, and d in our *createFECPacket* procedure.

calculateLongestPayload

We calculate the longest payload of each source packet by comparing their payload lengths. In the most recent FEC draft it is stated that we should pad the payload to the longest source packet size to protect all bits. We also deduct the RTP header size because we are calculating the longest payload.

generateBitString

The bit string is a ten byte string which contains what we need to repair. We generate the source packet bit string by combining several elements from the source packet. In our implementation, we begin by copying the first eight bytes of the RTP header to the FEC bit string. In the FEC draft, the first two bits are skipped, but we copy these bytes directly for simplicity, and we change these bits later. Next, we calculate a sixteen bit number which is the sum of the CRSC list, extension header, RTP- payload and padding. We copy this sum to the last two bits of the bit string.

getPaddedRTPPayload

The padded payload function takes a RTP packet and longest packet size. We compute the padded payload by appending 0x00 bytes at the end of the payload until the length matches the longest payload.

createFECPacket

We begin making the FEC packet by allocating a buffer of the parameter packet size to hold the packet content. Be begin by copying the first two bytes of the FEC bit string into the first two bytes of the buffer. We also add *10* as the first two bits to indicate that this is a 2d parity packet. Next, we copy the aforementioned sum of CRSC list, extension header, RTP- payload and padding into the next two bytes of the buffer. Next, we copy the timestamp, sequence number base, l, and d into the eight bytes of the buffer. Finally, we copy the payload directly onto the end of the FEC header. We now how a buffer containing the FEC packet, and we instantiate a FECPacket object with this buffer and size, before returning.

5.4.2 FECDecoder

Our FECDecoder implementation handles the core operations of FEC decoding, which is to repair a FECCluster. This function, as mentioned previously, is called *repairCluster* and is the only function used directly outside this class. *repairCluster* uses an additional nine helper functions which are called: *repairNonInterleaved*, *repairInterleaved...*

repairCluster

Our *repairCluster* procedure is our implementation of the *iterative repair* algorithm from the most recent FEC draft. The purpose of this function is to loop through the cluster of packets and attempt to repair as many lost packets as possible. In this function, we use two variables: *numRecoveredUntilThisIteration* and *numRecoveredSoFar*, both set to 0. We loop through the rows and columns and attempt to repair them, using our *repairNonInterleaved* and *repairInterleaved* functions. In these functions, we increase the *numRecoveredSoFar* if we repair a row or column successfully. At the end of the loop, we compare *numRecoveredUntilThisIteration* and *numRecoveredSoFar*. If *numRecoveredSoFar* is greater than *numRecoveredUntilThisIteration*, we set *numRecoveredUntilThisIteration* equal to *numRecoveredSoFar*. If not, we break out of the loop. Essentially, we loop through the rows and columns and attempt to repair them. If we encounter an iteration where we cannot repair any packet, we stop the repair process.

repairNonInterleaved and repairInterleaved

The *repairNonInterleaved* and *repairInterleaved* functions are very similar. The main idea of each function is to loop through either the rows or columns in a cluster and attempt to repair them using the *repairRow* function. In both functions we loop through the rows or columns accordingly. If one of these is either missing more than one RTP packet or just the FEC packet, we do not attempt to repair it. This is because more than one RTP packet cannot be recovered using the reversed xor scheme. Additionally, we do not care if the FEC packet is missing, because our intention is to repair RTP packets. In repairInterleaved we have to convert each column to a row, because we utilize the *repairRow* function. If we have a suitable row or column for repair, we perform the repair row function and on completion, we receive a repaired RTPPacket. Finally we insert this RTPPacket into the cluster and increase the numRecoveredSoFar by 1.

repairRow

The *repairRow* function takes a row of source and repair packets, and returns one repaired RTPPacket, by utilizing several helper functions. We begin by finding the sequence number of the missing packet in the row, using the *findSequenceNumber* function. Next, we use our *calculateRowBitString* method to acquire a bit string for the row, and use this bit string to create a repaired RTP header, using the *createHeader* function. Next we extract the variable y from byte eight through nine in the bit string. The variable y is the sum of CRSC list, extension header, RTP- payload and padding. We use this variable to calculate the payload, using the *calculatePayload* function.

Next, we calculate the new RTP packet's size by combining the static RTP header size (12 bytes) and y which accounts for the payload of the packet. We allocate a buffer of this size and copy the header and payload into this array. Finally we create and return a new RTPPacket using this buffer and size.

findSequenceNumber

The *findSequenceNumber* function finds the sequence number of the missing packet in a row of RTP packets and one FEC packet. We extract the base sequence number from the FEC packet, since we do not know which RTP packet is missing, but we know that the FEC packet is present. We loop through the source packets, and when we find the missing packet we return the iterator count plus the sequence number base. If the packet is protected by an interleaved column, we multiply the iterator by l.

calculateRowBitString

The purpose of the *calculateRowBitString* is to apply the xor operation to the bit strings of all the available RTP packets and FEC packet. We begin by generating the FEC bit string by utilizing the *generateFECBitString* method. We do this prior to the RTP packets, because this packet must be present. Next, we iterate over the RTP packets in the row. If a packet is missing, we do nothing. If the packet is present, we calculate the bit string of the packet, using the *generateBitString* method. We then apply the xor operation on this bit string and the previous bit string. Note that we override the previous bit string, always calculating the xor product of the old and new bit string.

generateFECBitString In *generateFECBitString*, we calculate the FEC bit string from the provided FEC packet. We firstly allocate a 10 byte array, because the FEC bit string is 10 bytes. We copy the first two bytes and the timestamp from the FEC packet to the FEC bit string at the same offsets. We also copy the length recovery from the FEC packet into the eight and ninth byte of the FEC bit string. Note that we add 12 bytes for each copying on the FEC packet, because the FEC packet is encapsulated within a RTP packet.

generateBitString The *generateBitString* method generates a bit string for a given RTP packet. This method is identical to the method with the same in FECEncoder, Section 5.4.1. We copy the first eight bytes of the RTP header into the bit string and compute the variable y, which we copy into the last two bytes of the bit string. Note that while these methods are identical, we keep them in separate files to improve code readability.

createHeader

In the *createHeader* function, we generate the RTP repair packet header, using the acquired bit string, sequence number and ssrc. We begin by allocating a 12 byte buffer, which holds the new RTP header. We copy the first two bytes from the bit string into the RTP header. Note that we also change the first two bits of the first byte to *10*, which indicates RTP version 2. Next, we copy the sequence number into byte three and four of the RTP header. We also copy the timestamp and ssrc into the RTP header, respectively. At last we return the new RTP header.

calculatePayload

In *calculatePayload*, we combine the payloads of the available RTP packets and one FEC packet to create the missing RTP packet payload. We also use the previously computed variable Y, to determine the size of the new payload. We begin by allocating the new payload as a buffer of size Y. As mentioned previously: the payloads of source packets on the sender side are padded 0x00 on the end of each packet, to match the length of the longest packet. This is done prior to generate the FEC bit string. Therefore, we have to pad the FEC and RTP packets to the length of Y to replicate this process. We copy the padded FEC payload into the payload buffer, because the FEC packet must be present. Next we apply the xor operation on this buffer with the available RTP packet payloads.

5.5 Limitations

Due to the limited time and resources of this thesis, we avoid implementing all of the specifications of the FEC draft, which leads to some limitations in our implementation. In order to protect the wireless multicast stream from both random as well as burst packet loss, it is recommended in the most recent FEC draft to use 2d parity FEC protection. For this reason, we only implement the 2d parity FEC scheme and disregard implementing the bit mask and retransmission scheme. However, as previously mentioned, it is trivial to extend our implementation to support these schemes, in future work.

We also do not implement FEC support for RTSP or SDP. Instead we use fixed payload types for the FEC stream with 115 for non interleaved repair packets and 116 for interleaved repair packets. We also set the column and row count statically, on both receiver and sender side, in addition to a static repair window at the receiver side. However, because we the RTP sequence number is generated randomly for each RTP session, we cannot know the offset of each FEC cluster base prior to the stream. The only way we know how to synchronize this is to find one non-interleaved and one interleaved FEC packet with the same base sequence number. In that case we know that sequence number and can calculate the offsets for later clusters. We do this in *FEC2DParityMultiplexor* by using a *emergencyBuffer*. Before we know the FECCluster offset, we place incoming packets in this buffer. For each packet we place in the buffer we look for the non-interleaved and interleaved packets with the same sequence number base. When we find it, we flush the packets and continue the stream in the regular fashion. Additionally, because we do not that metadata negotiation, we do not know the ssrc of the stream. We cannot know this prior to the stream, because this is also generated randomly. Therefore, we save the ssrc from the first packet we receive into a local variable which we use when repairing RTP packets.

Finally, we do not support RTP extension header in our FEC implementation. This is not a priority for us, because we stream H.264 video in our main experiment, which does not require RTP extension. This is, however, simple to add in future work.

5.6 Discussion and Conclusions

In this chapter we implemented a fully functional FEC streamer and receiver as an extension in Live555 Media Streaming. This implementation follows the instructions of the most recent FEC draft (Zanaty et al. 2018), with exceptions which we state in Section 5.5. The FEC streamer can be connected to most media sources in Live555, and produces repair packets. The receiver combines source and repair packets to produce a stream of repaired source packets. We provide an example of how to execute the streaming and receiving in Figure 5.6. In this example we use L = 4, D = 5 and repair window = 300ms. The multicast address used is 232.126.107.222,

```
#Sender
$ ./testH264VideoStreamer 4 5 test.264

#Receiver
$ ./client 232.126.107.222 4 5 300
```

Figure 5.6: Example of how to run the test programs in our FEC implementation.

and the input file is test.264. In the following chapter, we evaluate the experiment of this thesis. The experiment utilize this implementation.

Chapter 6

Evaluation

In Chapter 4, we designed and executed an experiment where we streamed multiple instances of a source trace file containing an RTP stream of H.264 encoded 1080p video. These instances contained several FEC configurations, but were limited by a L value of 20, a maximum FEC repair window of 500ms, and a maximum overhead of 66.67%. We ran these trace files through our FEC implementation and captured the packet loss reduction. In this chapter, we evaluate these results and explore how the most recent FEC draft can be applied to low latency wireless multicast streams.

6.1 Experiment Summary

We ran the experiment three times, and therefore we imagine the graphs would be more smooth if we ran it more times.

6.1.1 Constraints

As we discussed in Chapter 4, we have set some hard constraints concerning both FEC latency and overhead. The latency constraint is set at 500 milliseconds. We chose this constraint because we consider it reasonable and it, nonetheless, faster than the current DASH approach in the football stadium. The overhead is set at a maximum of 66.67%. If we go above this limit, it is effectively the same as retransmission.

6.2 Video Encoding

For this experiment we used the tractor sample from (Xiph 2018). We encoded this file with a NAL unit size of 1432 to prevent RTP fragmentation. Additionally, we used variable bitrate, with a crf value of 25. We decided to use crf value of 25 to simulate an approximately 6 mbps stream. By

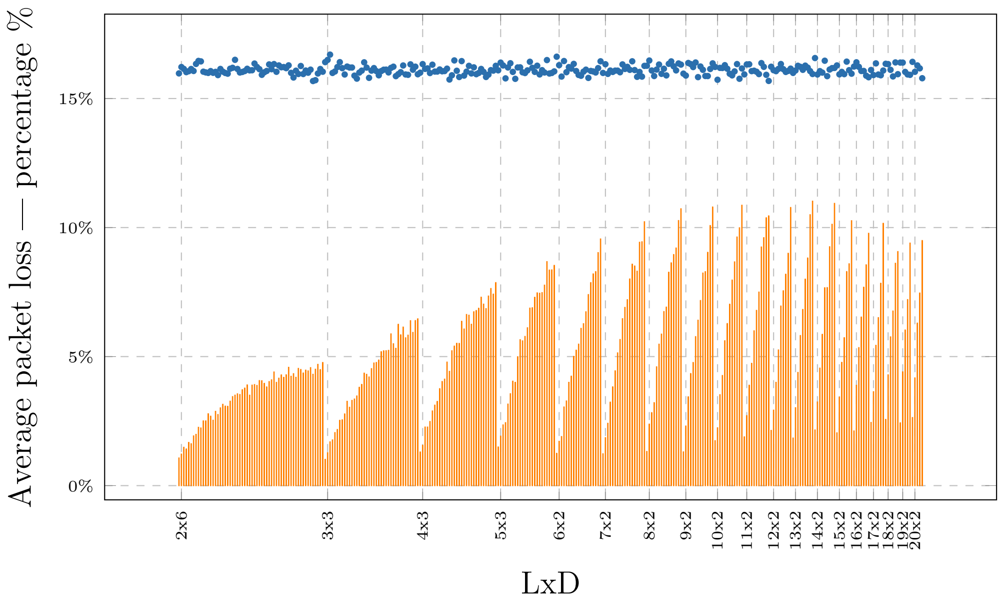

Figure 6.1: Average packet loss before and after FEC

keeping it at 6 mbps we maintain our restriction of total bandwidth consumption of 10 mbps. The reason for using variable bitrate was to simulate real case scenarios.

6.3 Packet Loss

Figure 6.1 presents the average packet loss before and after the FEC repair scheme. They are ordered by increasing L and D value. The packet loss before our FEC implementation is $\approx 16\%$, which is expected, because we used the simple Gilbert Model to apply packet loss at a rate of 0.838026. This rate was collected from Mochnáč et al. (2010). The average packets loss after our FEC implementation ranges from $\approx 1.02\%$ to $\approx 11.02\%$.

Figure 6.1 shows a clear sawtooth pattern. This pattern is expected because bigger L and D clusters yield lower protection, but does provide lower overhead. Note that the sawtooth pattern is more abrupt when reaching higher L values. This is simply because there were fewer values which met our requirements when L was increased.

The packet loss is most efficient with smaller clusters and less efficient with larger clusters. The graph does indicate, however, that it is not a linear curve. For example, on the L = 2 values, we see a rapid increase in packet loss in the beginning and then it gets more even. This indicates that when reaching a certain cluster size, the packet loss will not increase as rapidly as for the lower D values.

Figure 6.1 indicates that when L increases, but D is equal to 2, the clusters are less efficient. This is also expected, as the overall cluster size is bigger.

6.4 Latency

In live streaming, low latency is an important priority, which is why we examine streams with the lowest latency in **??**. This table shows an ordered list of the 30 streams with lowest latency. Not surprisingly, smaller clusters yield lower latency. There are a few at the beginning at approximately 160ms. 3 by 3 has the highest overhead, but yields higher packets in regards to for example 10 by 2.

6.4.1 Correlation between latency and overhead

Figure 6.2 shows the correlation between latency, e.g. the repair window and the overhead. The natural conclusion would be that as the overhead increases the latency would drop, due to smaller clusters. We do, however se that the pattern is quite arbitrarily distributed. However, towards the higher overhead values we see that the repair window is still quite high. This is because while having big overhead does not necessarily mean small clusters. When either L or D is small such as 2 small cluster sizes it is possible to have large cluster with high overhead.

6.5 Discussion and conclusions

What is the best configuration for live streaming? We therefore need to consider packet loss and latency. If packet loss is high, the quality of experience for users will drop. If the overhead is too large we have to sacrifice quality when encoding the file, also resulting in lower quality of experience for users. We, therefore, look at the FEC configurations which yielded the lowest packet error rate after FEC, and consider the cost of overhead. FEC configuration 3x3 seems promising for live streaming because it has low latency and low packet loss.

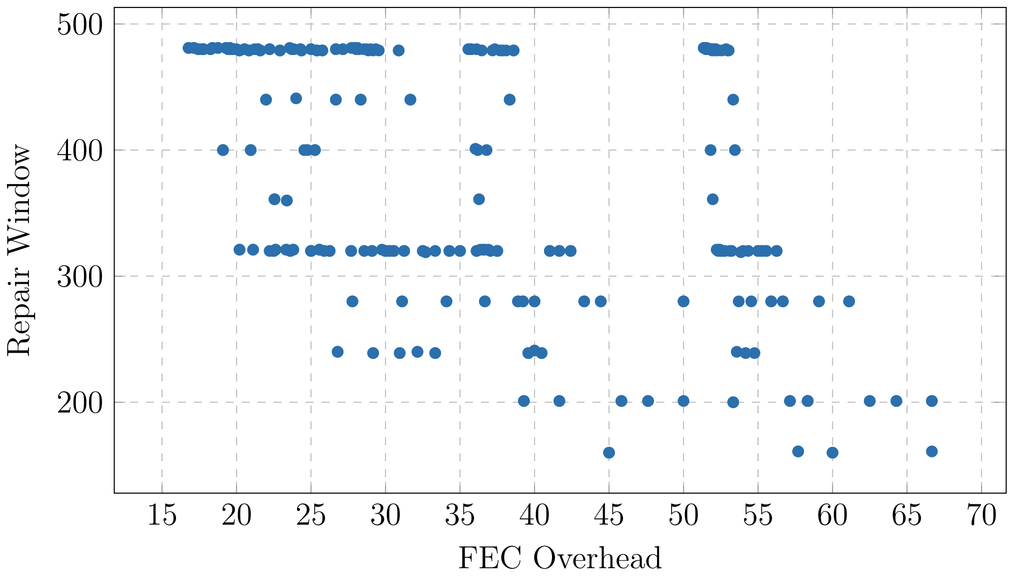

Figure 6.2: Correlation between overhead and latency

Chapter 7

Conclusion

In this chapter, we begin in Section 7.1 by providing a brief summary of our contributions, which includes packet loss, overhead and latency considerations. In Section 7.2, we provide a critical assessment of our work. Finally, in Section 7.3, we discuss several possible directions for future work.

7.1 Summary of Contributions

In this work, we investigate wether the most recent FEC draft is suitable for real-time wireless multicast live streaming, with emphasis on three main points: packet reduction effectivity, and latency and overhead impact. We design and perform an experiment in which we simulate wireless packet loss in multicast streams with a Gilbert model pattern of $\approx 16\%$ random packet loss. We check all L and D values within several constraints: maximum 500ms repair window (latency impact), 66.67% overhead, and a maximum L value of 20. For all these L and D values we stream the tractor sample three times, to avoid possible outliers in the data.

7.1.1 Evaluation of the most recent FEC draft

To the best of our knowledge, no other studies have evaluated the most recent FEC draft in regards to wireless multicast RTP streaming. We evaluate this FEC draft's effectivity on its ability to reduce packet loss, maintain low latency and overhead costs.

Packet Loss Reduction

Our results show that some FEC configurations (L and D values) are very effective at reducing packet loss. The best configurations are able to reduce the packet loss from $\approx 16\%$ down to $\approx 1.02\%$ (3x3 FEC configuration). The worst case reduces it from $\approx 16\%$ to $\approx 11.02\%$ (13x11 FEC configuration).

The pattern is very clear: the bigger the cluster the worse packet loss reduction. As we can see from these results 3x3 provides also the best packet loss reduction.

Latency Impact

In real-time streams latency is critical for a good user experience. It is, therefore, important to analyze its impact in depth. Our results show that the lowest latency overhead is 160ms, and the highest is 481ms (due to our constraint at 500ms). With the lowest being 160ms this FEC draft may or may not be suitable for real-time scenarios. For most video streams though, 160ms is not noticeable though.

Overhead Cost

Because we have to send additional packets with FEC some overhead is inevitable. It is especially critical to minimize overhead in multicast streams because the bandwidth is restricted. Our results show that small clusters have the highest overhead, and the bigger clusters have lower overhead. There are some exceptions, where small values of either L or D combined with a large value causes the cluster to be large, but also the overhead to be large. This is as expected.

Takeaway points

As we can see, a smaller cluster reduces packet and the latency the most, however, it also has the most overhead, as expected. So, the conclusion is that, as expected, for each scenario we have to make an educated choice to wether sacrifice packet loss, overhead or latency.

7.2 Critical Assessment

If more time was in our hands, we would certainly have ran the experiment in more iterations to reduce the effect of even more outliers. Additionally, we used the simple gilbert model, but it might have been better to use the more advance wireless multicast packet loss model.

7.3 Future Work

In this thesis we have established a basis for streaming live H.264 video over wireless multicast with FEC protection. There are, of course, plenty of open ends for which future work can be based.

For example, we have not ran our experiments in a real case football stadium scenario with real interference and nodes. So, it is critical for future work to assess the effectiveness of this FEC draft out in the field.

In this thesis, we evaluate a static FEC scheme. However, in certain scenarios a more adaptive approach might be more appropriate. For example, in periods of burst loss, it might be better to switch to a higher L value and lower D value. In periods of low random loss, it might be better to switch to a lower L and D value to reduce latency.

Bibliography

Amino Communications. 2005. "Amino selected for first HD IPTV deployment in USA." November. Accessed April 28, 2017. `https://investor.aminocom.com/story-EMQ1314`.

Carmel, S., T. Daboosh, E. Reifman, N. Shani, Z. Eliraz, D. Ginsberg, and E. Ayal. 2002. *Network media streaming.* US Patent 6,389,473, May. `https://www.google.com/patents/US6389473`.

Edwards, Thomas. 2017. *SMPTE ST 2022: Moving Serial Interfaces (ASI & SDI) to IP.* `https://www.smpte.org/sites/default/files/2017-08-17-ST-2022-Edwards-V4-Handout.pdf`.

eMarketer. 2017. *Average Time Spent per Day with Video by US Adults, by Device, 2015-2019 (hrs:mins),* September. Accessed July 16, 2018. `http://www.emarketer.com/Chart/Average-Time-Spent-per-Day-with-Video-by-US-Adults-by-Device-2015-2019-hrsmins/211432`.

Ericsson AB. 2005. "B2 sharpens its competitive edge with ADSL2+ from Ericsson." Accessed April 28, 2017. `http://archive.ericsson.net/service/internet/picov/get?DocNo=1/21331-FGB101063&Lang=EN`.

Finlayson, Ross. 2018. "What is the best way to modify or extend the functionality of the code?" Accessed September 8, 2018. `http://www.live555.com/liveMedia/faq.html#modifying-and-extending`.

Forgie, James W. 1979. *ST — A Proposed Internet Stream Protocol,* September. `http://www.cis.ohio-state.edu/htbin/ien/ien119.html`.

Fosberg, Ina Alette. 2018. *Hardware setup.*

Holbrook, H., B. Cain, and B. Haberman. 2006. *Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast.* RFC 4604. RFC Editor, August. `https://tools.ietf.org/rfc/rfc4604.txt`.

Hua, Kien A., Ying Cai, and Simon Sheu. 1998. "Patching." In *Proceedings of the sixth ACM international conference on Multimedia - MULTIMEDIA '98,* 191–200. New York, New York, USA: ACM Press. ISBN: 0201309904. doi:10.1145/290747.290771. http://portal.acm.org/citation.cfm?doid=290747.290771.

Jacobsen, Espen, Carsten Griwodz, and Pål Halvorsen. 2010. "Pullpatching." In *Proceedings of the international conference on Multimedia - MM '10,* 799. New York, New York, USA: ACM Press. ISBN: 9781605589336. doi:10.1145/1873951.1874081. http://dl.acm.org/citation.cfm?doid=1873951.1874081.

Li, A. 2007. *RTP Payload Format for Generic Forward Error Correction.* RFC5109, December. http://tools.ietf.org/rfc/rfc5109.txt.

Linksys. 2015. "How to resolve poor or no signal from a wireless router." Accessed May 24, 2018. https://www.linksys.com/ca/support-article?articleNum=136786.

McCanne, Steven, Van Jacobson, and Martin Vetterli. 1996. "Receiver-driven layered multicast." In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '96,* 117–130. New York, New York, USA: ACM Press. ISBN: 0897917901. doi:10.1145/248156.248168. http://portal.acm.org/citation.cfm?doid=248156.248168.

Mochnáč, J., S. Marchevský, and P. Kocan. 2010. "Simulation of packet losses in video transfers using real-time transport protocol." In *20th International Conference Radioelektronika 2010,* 1–4. April. doi:10.1109/RADIOELEK.2010.5478577.

Multicast Technologies. 2008. "The Relative Size of the Multicast Enabled Internet." Multicast Technologies. Accessed February 24, 2008. http://www.multicasttech.com/status/.

National Institute of Standards and Technology. 1995. *FIPS PUB 46-3: DATA ENCRYPTION STANDARD (DES).* Withdrawn on May 19, 2005. October. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

———. 2001. *FIPS PUB 197: Advanced Encryption Standard (AES).* November. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

Nayarasi. 2012. "Wireless Multicast is not working – Why ?" Accessed March 30, 2018. https://mrncciew.com/2012/12/20/why-wireless-multicast-is-not-working/.

Perkins, C., M. McBride, D. Stanley, W. Kumari, and JC. Zuniga. 2018. *Multicast Considerations over IEEE 802 Wireless Media draft-ietf-mboned-ieee802-mcast-problems-02.* https://tools.ietf.org/html/draft-ietf-mboned-ieee802-mcast-problems-02.

Rainer, Benjamin, Stefan Petscharnig, and Christian Timmerer. 2018. "Merge and Forward: A Self-Organized Inter-Destination Media Synchronization Scheme for Adaptive Media Streaming over HTTP." In *MediaSync: Handbook on Multimedia Synchronization,* edited by Mario Montagud, Pablo Cesar, Fernando Boronat, and Jack Jansen, 593–627. Cham: Springer International Publishing. ISBN: 978-3-319-65840-7. doi:10.1007/978-3-319-65840-7_21. https://doi.org/10.1007/978-3-319-65840-7_21.

Deering, S.E. 1989. *Host extensions for IP multicasting.* RFC 1112 (Internet Standard). RFC. Updated by RFC 2236. RFC Editor, RFC Editor, Fremont, CA, USA, August. doi:10.17487/RFC1112. https://www.rfc-editor.org/rfc/rfc1112.txt.

Topolcic, C. 1990. *Experimental Internet Stream Protocol: Version 2 (ST-II).* RFC 1190 (Experimental). RFC. Obsoleted by RFC 1819. RFC Editor, RFC Editor, Fremont, CA, USA, October. doi:10.17487/RFC1190. https://www.rfc-editor.org/rfc/rfc1190.txt.

Schulzrinne, H., S. Casner, R. Frederick, and V. Jacobson. 1996. *RTP: A Transport Protocol for Real-Time Applications.* RFC 1889 (Proposed Standard). RFC. Obsoleted by RFC 3550. RFC Editor, RFC Editor, Fremont, CA, USA, January. doi:10.17487/RFC1889. https://www.rfc-editor.org/rfc/rfc1889.txt.

Hoffman, D., G. Fernando, and V. Goyal. 1996. *RTP Payload Format for MPEG1/MPEG2 Video.* RFC 2038 (Proposed Standard). RFC. Obsoleted by RFC 2250. RFC Editor, RFC Editor, Fremont, CA, USA, October. doi:10.17487/RFC2038. https://www.rfc-editor.org/rfc/rfc2038.txt.

Rosenberg, J., and H. Schulzrinne. 1999. *An RTP Payload Format for Generic Forward Error Correction.* RFC 2733 (Proposed Standard). RFC. Obsoleted by RFC 5109. RFC Editor, RFC Editor, Fremont, CA, USA, December. doi:10.17487/RFC2733. https://www.rfc-editor.org/rfc/rfc2733.txt.

Kobayashi, K., A. Ogawa, S. Casner, and C. Bormann. 2002. *RTP Payload Format for 12-bit DAT Audio and 20- and 24-bit Linear Sampled Audio.* RFC 3190 (Proposed Standard). RFC. RFC Editor, RFC Editor, Fremont, CA, USA, January. doi:10.17487/RFC3190. https://www.rfc-editor.org/rfc/rfc3190.txt.

Schulzrinne, H., S. Casner, R. Frederick, and V. Jacobson. 2003. *RTP: A Transport Protocol for Real-Time Applications.* RFC 3550 (Internet Standard). RFC. Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164, 8083, 8108. RFC Editor, RFC Editor, Fremont, CA, USA, July. doi:10.17487/RFC3550. https://www.rfc-editor.org/rfc/rfc3550.txt.

J. van der Meer, D. Mackie, V. Swaminathan, D. Singer, and P. Gentric. 2003. *RTP Payload Format for Transport of MPEG-4 Elementary Streams.* RFC 3640 (Proposed Standard). RFC. Updated by RFC 5691. RFC Editor, RFC Editor, Fremont, CA, USA, November. doi:10.17487/RFC3640. https://www.rfc-editor.org/rfc/rfc3640.txt.

Hellstrom, G., and P. Jones. 2005. *RTP Payload for Text Conversation.* RFC 4103 (Proposed Standard). RFC. RFC Editor, RFC Editor, Fremont, CA, USA, June. doi:10.17487/RFC4103. https://www.rfc-editor.org/rfc/rfc4103.txt.

Gharai, L., and C. Perkins. 2005. *RTP Payload Format for Uncompressed Video.* RFC 4175 (Proposed Standard). RFC. Updated by RFC 4421. RFC Editor, RFC Editor, Fremont, CA, USA, September. doi:10.17487/RFC4175. https://www.rfc-editor.org/rfc/rfc4175.txt.

Lazzaro, J., and J. Wawrzynek. 2006. *An Implementation Guide for RTP MIDI.* RFC 4696 (Informational). RFC. RFC Editor, RFC Editor, Fremont, CA, USA, November. doi:10.17487/RFC4696. https://www.rfc-editor.org/rfc/rfc4696.txt.

Wang, Y.-K., R. Even, T. Kristensen, and R. Jesup. 2011. *RTP Payload Format for H.264 Video.* RFC 6184 (Proposed Standard). RFC. RFC Editor, RFC Editor, Fremont, CA, USA, May. doi:10.17487/RFC6184. https://www.rfc-editor.org/rfc/rfc6184.txt.

Spittka, J., K. Vos, and JM. Valin. 2015. *RTP Payload Format for the Opus Speech and Audio Codec.* RFC 7587 (Proposed Standard). RFC. RFC Editor, RFC Editor, Fremont, CA, USA, June. doi:10.17487/RFC7587. https://www.rfc-editor.org/rfc/rfc7587.txt.

Lennox, J., K. Gross, S. Nandakumar, G. Salgueiro, and B. Burman. 2015. *A Taxonomy of Semantics and Mechanisms for Real-Time Transport Protocol (RTP) Sources.* RFC 7656 (Informational). RFC. RFC Editor, RFC Editor, Fremont, CA, USA, November. doi:10.17487/RFC7656. https://www.rfc-editor.org/rfc/rfc7656.txt.

Siddiqi, Faisal Zakaria. 2016. "Caching for a Global Netflix." March. Accessed April 28, 2017. http://techblog.netflix.com/2016/03/caching-for-global-netflix.html.

Singh, Varun, Albert Abello Lozano, and Jorg Ott. 2013. "Performance Analysis of Receive-Side Real-Time Congestion Control for WebRTC." In *2013 20th International Packet Video Workshop,* 1–8. IEEE, December. ISBN: 978-1-4799-2172-0. doi:10.1109/PV.2013.6691454. http://ieeexplore.ieee.org/document/6691454/.

Tang, Chiping, and Philip K. McKinley. 2003. "Modeling Multicast Packet Losses in Wireless LANs." In *Proceedings of the 6th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems,* 130–133. MSWIM '03. San Diego, CA, USA: ACM. ISBN: 1-58113-766-4. doi:10.1145/940991.941015. http://doi.acm.org/10.1145/940991.941015.

Thompson, Hugh. 2011. "IPTV in Canada: An update." December. Accessed April 28, 2017. http://www.digitalhome.ca/2011/12/iptv-in-canada-an-update/.

Wang, Bing, Jim Kurose, Prashant Shenoy, and Don Towsley. 2008. "Multimedia Streaming via TCP: An Analytic Performance Study." *ACM Trans. Multimedia Comput. Commun. Appl.* (New York, NY, USA) 4, no. 2 (May): 16:1–16:22. ISSN: 1551-6857. doi:10.1145/1352012.1352020. http://doi.acm.org/10.1145/1352012.1352020.

Wang, Y.-K., Y. Sanchez, T. Schierl, S. Wenger, and M. M. Hannuksela. 2016. *RTP Payload Format for High Efficiency Video Coding (HEVC).* RFC7798, March. http://tools.ietf.org/rfc/rfc7798.txt.

Weil, Nicolas. 2014. "The State of MPEG-DASH Deployment." April. Accessed April 28, 2017. http://www.streamingmediaglobal.com/Articles/Editorial/Featured-Articles/The-State-of-MPEG-DASH-Deployment-96144.aspx.

Westerlund, Johan. 2015. "Forward Error Correction in Real-time Video Streaming Applications." Master's thesis, Umeaa University.

Xiph. 2018. "Xiph.org Video Test Media [derf's collection]." Accessed June 4, 2018. https://media.xiph.org/video/derf/.

Zanaty, M., V. Singh, A. Begen, and G. Mandyam. 2018. *RTP Payload Format for Flexible Forward Error Correction (FEC) draft-ietf-payload-flexible-fec-scheme-10.* https://tools.ietf.org/html/draft-ietf-payload-flexible-fec-scheme-10.

Appendices

Code Manual

The implementation of the most recent FEC draft used in this thesis can be found here: `https://bitbucket.org/mpg_code/generic_fec/src/master/` The scripts used in the main experiment can be found here: `https://bitbucket.org/mrfonnes/mainexperiment/src/master/`

Experiment Results

The experiment results are presented in the following tables.

L	D	Original Loss	FEC loss	Repair Window	Overhead
2	6	15.97%	1.07%	201ms	66.67%
2	7	16.22%	1.22%	201ms	64.29%
2	8	16.15%	1.48%	201ms	62.50%
2	9	16.02%	1.42%	280ms	61.11%
2	10	16.06%	1.67%	160ms	60.00%
2	11	16.14%	1.62%	280ms	59.09%
2	12	16.07%	1.93%	201ms	58.33%
2	13	16.34%	1.99%	161ms	57.69%
2	14	16.46%	2.26%	201ms	57.14%
2	15	16.44%	2.23%	280ms	56.67%
2	16	16.04%	2.51%	320ms	56.25%
2	17	16.01%	2.51%	280ms	55.88%
2	18	15.99%	2.78%	320ms	55.56%
2	19	16.06%	2.69%	320ms	55.26%
2	20	16.00%	2.53%	320ms	55.00%
2	21	16.03%	2.88%	239ms	54.76%
2	22	15.91%	2.75%	280ms	54.55%
2	23	16.15%	3.01%	320ms	54.35%
2	24	16.04%	3.15%	239ms	54.17%
2	25	15.99%	3.08%	320ms	54.00%
2	26	15.96%	3.07%	319ms	53.85%
2	27	16.16%	3.27%	280ms	53.70%
2	28	16.19%	3.45%	240ms	53.57%
2	29	16.50%	3.50%	400ms	53.45%
2	30	16.15%	3.56%	440ms	53.33%
2	31	16.01%	3.53%	320ms	53.23%
2	32	16.03%	3.71%	320ms	53.13%
2	33	16.07%	3.78%	479ms	53.03%
2	34	16.15%	3.90%	479ms	52.94%
2	35	16.09%	3.50%	480ms	52.86%

Table 1: First 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
2	36	16.10%	3.89%	320ms	52.78%
2	37	16.36%	3.91%	320ms	52.70%
2	38	16.22%	3.88%	320ms	52.63%
2	39	16.14%	4.07%	479ms	52.56%
2	40	15.92%	4.06%	320ms	52.50%
2	41	16.02%	3.96%	321ms	52.44%
2	42	16.07%	3.82%	321ms	52.38%
2	43	16.32%	4.03%	320ms	52.33%
2	44	16.14%	4.10%	479ms	52.27%
2	45	16.34%	4.40%	321ms	52.22%
2	46	16.04%	4.00%	480ms	52.17%
2	47	16.28%	4.16%	480ms	52.13%
2	48	16.21%	4.29%	479ms	52.08%
2	49	16.22%	4.19%	480ms	52.04%
2	50	16.18%	4.29%	480ms	52.00%
2	51	16.29%	4.58%	361ms	51.96%
2	52	15.99%	4.24%	479ms	51.92%
2	53	15.81%	4.34%	480ms	51.89%
2	54	16.08%	4.20%	480ms	51.85%
2	55	15.95%	4.54%	400ms	51.82%
2	56	16.26%	4.50%	480ms	51.79%
2	62	15.95%	4.36%	480ms	51.61%
2	63	16.09%	4.47%	480ms	51.59%
2	64	16.00%	4.44%	481ms	51.56%
2	65	16.12%	4.57%	480ms	51.54%
2	66	15.68%	4.31%	480ms	51.52%
2	68	15.71%	4.51%	481ms	51.47%
2	70	15.97%	4.70%	481ms	51.43%
2	73	16.14%	4.49%	481ms	51.37%
2	74	16.04%	4.77%	481ms	51.35%

Table 2: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
3	3	16.41%	1.02%	161ms	66.67%
3	4	16.50%	1.28%	201ms	58.33%
3	5	16.70%	1.69%	200ms	53.33%
3	6	15.99%	1.77%	280ms	50.00%
3	7	16.05%	2.05%	201ms	47.62%
3	8	16.22%	2.17%	201ms	45.83%
3	9	16.41%	2.53%	280ms	44.44%
3	10	16.18%	2.55%	280ms	43.33%
3	11	15.93%	2.78%	320ms	42.42%
3	12	16.24%	3.27%	320ms	41.67%
3	13	16.20%	3.04%	320ms	41.03%
3	14	16.20%	3.30%	239ms	40.48%
3	15	15.89%	3.35%	280ms	40.00%
3	16	15.77%	3.47%	239ms	39.58%
3	17	16.01%	3.81%	280ms	39.22%
3	18	16.08%	3.92%	280ms	38.89%
3	19	16.40%	4.36%	479ms	38.60%
3	20	16.20%	4.31%	440ms	38.33%
3	21	15.80%	4.21%	479ms	38.10%
3	22	16.08%	4.53%	479ms	37.88%
3	23	16.18%	4.75%	479ms	37.68%
3	24	16.06%	4.77%	320ms	37.50%
3	25	15.90%	4.87%	480ms	37.33%
3	26	16.06%	5.19%	479ms	37.18%
3	27	16.12%	5.22%	320ms	37.04%
3	28	16.12%	5.22%	321ms	36.90%
3	29	16.01%	5.24%	400ms	36.78%
3	30	16.19%	5.88%	321ms	36.67%
3	31	16.23%	5.50%	321ms	36.56%
3	32	15.87%	5.32%	479ms	36.46%

Table 3: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
3	33	15.94%	6.25%	321ms	36.36%
3	34	16.02%	5.84%	361ms	36.27%
3	35	16.28%	6.14%	400ms	36.19%
3	36	15.93%	5.73%	480ms	36.11%
3	37	15.92%	5.83%	401ms	36.04%
3	41	16.32%	6.39%	480ms	35.77%
3	42	16.12%	5.94%	480ms	35.71%
3	44	16.24%	6.39%	480ms	35.61%
3	45	15.94%	6.46%	480ms	35.56%
4	3	16.00%	1.30%	201ms	58.33%
4	4	16.33%	1.56%	201ms	50.00%
4	5	16.15%	2.27%	160ms	45.00%
4	6	16.17%	2.27%	201ms	41.67%
4	7	15.99%	2.48%	201ms	39.29%
4	8	16.31%	2.89%	320ms	37.50%
4	9	16.05%	3.12%	320ms	36.11%
4	10	16.05%	3.26%	320ms	35.00%
4	11	16.13%	3.76%	280ms	34.09%
4	12	16.06%	4.03%	239ms	33.33%
4	13	16.06%	4.12%	319ms	32.69%
4	14	16.16%	4.79%	240ms	32.14%
4	15	15.74%	4.43%	440ms	31.67%
4	16	15.90%	5.22%	320ms	31.25%
4	17	16.48%	5.40%	479ms	30.88%
4	18	16.08%	5.51%	320ms	30.56%
4	19	15.83%	5.51%	320ms	30.26%
4	20	16.45%	6.37%	320ms	30.00%
4	21	15.89%	6.07%	321ms	29.76%
4	22	16.04%	6.63%	479ms	29.55%
4	23	16.27%	6.60%	480ms	29.35%

Table 4: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
4	24	15.97%	6.25%	479ms	29.17%
4	25	16.21%	6.73%	480ms	29.00%
4	26	15.94%	6.78%	479ms	28.85%
4	27	15.99%	6.87%	480ms	28.70%
4	28	16.15%	7.30%	480ms	28.57%
4	31	16.03%	7.03%	480ms	28.23%
4	32	15.84%	6.86%	481ms	28.13%
4	33	15.93%	7.34%	480ms	28.03%
4	34	16.10%	7.64%	481ms	27.94%
4	35	16.08%	7.42%	481ms	27.86%
4	37	16.27%	7.87%	481ms	27.70%
5	3	16.09%	1.50%	200ms	53.33%
5	4	16.39%	1.92%	160ms	45.00%
5	5	16.28%	2.35%	241ms	40.00%
5	6	15.78%	2.44%	280ms	36.67%
5	7	16.20%	3.16%	320ms	34.29%
5	8	16.37%	3.56%	320ms	32.50%
5	9	16.03%	4.05%	280ms	31.11%
5	10	15.76%	4.00%	320ms	30.00%
5	11	16.20%	4.99%	320ms	29.09%
5	12	16.21%	5.65%	440ms	28.33%
5	13	16.02%	5.61%	320ms	27.69%
5	14	15.99%	5.78%	480ms	27.14%
5	15	16.09%	6.11%	480ms	26.67%
5	16	16.42%	6.88%	320ms	26.25%
5	17	16.19%	6.89%	320ms	25.88%
5	18	16.03%	7.30%	321ms	25.56%
5	19	16.27%	7.47%	400ms	25.26%
5	20	15.93%	7.45%	480ms	25.00%
5	21	16.10%	7.48%	400ms	24.76%

Table 5: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
5	22	16.01%	7.77%	400ms	24.55%
5	25	16.49%	8.68%	441ms	24.00%
5	26	15.97%	8.35%	480ms	23.85%
5	27	15.99%	8.36%	480ms	23.70%
5	28	16.05%	8.53%	481ms	23.57%
6	2	16.62%	1.25%	201ms	66.67%
6	3	16.29%	1.72%	280ms	50.00%
6	4	15.85%	1.90%	201ms	41.67%
6	5	16.45%	3.05%	280ms	36.67%
6	6	16.02%	3.28%	320ms	33.33%
6	7	16.22%	4.00%	239ms	30.95%
6	8	16.16%	4.24%	239ms	29.17%
6	9	16.34%	5.01%	280ms	27.78%
6	10	16.07%	5.25%	440ms	26.67%
6	11	15.91%	5.48%	479ms	25.76%
6	12	15.88%	6.09%	320ms	25.00%
6	13	16.04%	6.27%	479ms	24.36%
6	14	16.12%	6.73%	321ms	23.81%
6	15	15.80%	7.40%	321ms	23.33%
6	16	16.07%	7.87%	479ms	22.92%
6	17	16.05%	8.20%	361ms	22.55%
6	18	16.01%	8.29%	480ms	22.22%
6	21	16.18%	9.03%	480ms	21.43%
6	22	16.44%	9.56%	480ms	21.21%
7	2	15.99%	1.23%	201ms	64.29%
7	3	16.33%	1.86%	201ms	47.62%
7	4	15.96%	2.41%	201ms	39.29%
7	5	16.08%	3.23%	320ms	34.29%
7	6	16.03%	3.83%	239ms	30.95%
7	7	16.08%	4.45%	320ms	28.57%

Table 6: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
7	8	16.08%	5.14%	240ms	26.79%
7	9	16.02%	5.66%	479ms	25.40%
7	10	16.32%	6.47%	480ms	24.29%
7	11	16.15%	6.91%	360ms	23.38%
7	12	16.23%	7.20%	321ms	22.62%
7	13	16.11%	8.01%	440ms	21.98%
7	14	16.44%	8.58%	480ms	21.43%
7	15	16.10%	8.51%	400ms	20.95%
7	16	15.84%	8.31%	480ms	20.54%
7	18	16.03%	9.43%	480ms	19.84%
7	19	15.85%	9.45%	481ms	19.55%
7	20	16.27%	10.22%	481ms	19.29%
8	2	16.27%	1.32%	201ms	62.50%
8	3	16.47%	2.38%	201ms	45.83%
8	4	16.09%	2.83%	320ms	37.50%
8	5	15.87%	3.21%	320ms	32.50%
8	6	16.12%	4.60%	239ms	29.17%
8	7	16.32%	5.48%	240ms	26.79%
8	8	15.96%	5.87%	320ms	25.00%
8	9	16.15%	6.74%	320ms	23.61%
8	10	15.90%	6.92%	320ms	22.50%
8	11	16.35%	8.27%	479ms	21.59%
8	12	16.44%	8.63%	479ms	20.83%
8	13	16.25%	8.95%	479ms	20.19%
8	14	16.09%	9.21%	480ms	19.64%
8	16	16.36%	10.27%	481ms	18.75%
8	17	16.31%	10.73%	481ms	18.38%
9	2	15.97%	1.31%	280ms	61.11%
9	3	15.89%	2.31%	280ms	44.44%
9	4	16.38%	3.44%	320ms	36.11%

Table 7: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
9	5	16.33%	4.34%	280ms	31.11%
9	6	16.22%	4.77%	280ms	27.78%
9	7	16.40%	5.77%	479ms	25.40%
9	8	15.83%	6.41%	320ms	23.61%
9	9	16.24%	7.17%	320ms	22.22%
9	10	16.11%	8.24%	321ms	21.11%
9	11	15.87%	8.29%	321ms	20.20%
9	12	15.87%	9.04%	480ms	19.44%
9	14	16.14%	10.08%	480ms	18.25%
9	15	16.36%	10.80%	480ms	17.78%
10	2	16.22%	1.74%	160ms	60.00%
10	3	15.73%	2.24%	280ms	43.33%
10	4	16.19%	3.52%	320ms	35.00%
10	5	16.18%	4.27%	320ms	30.00%
10	6	16.29%	5.61%	440ms	26.67%
10	7	16.15%	6.28%	480ms	24.29%
10	8	15.96%	7.03%	320ms	22.50%
10	9	15.90%	7.96%	321ms	21.11%
10	10	16.05%	8.67%	480ms	20.00%
10	11	16.34%	9.63%	400ms	19.09%
10	13	15.77%	9.99%	480ms	17.69%
10	14	16.16%	10.86%	481ms	17.14%
11	2	15.93%	1.89%	280ms	59.09%
11	3	16.32%	2.71%	320ms	42.42%
11	4	16.33%	3.89%	280ms	34.09%
11	5	16.06%	4.74%	320ms	29.09%
11	6	16.08%	6.00%	479ms	25.76%
11	7	16.04%	6.79%	360ms	23.38%
11	8	15.95%	7.50%	479ms	21.59%
11	9	16.37%	9.25%	321ms	20.20%

Table 8: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
11	10	16.23%	9.61%	400ms	19.09%
11	12	15.92%	10.38%	480ms	17.42%
11	13	15.68%	10.45%	481ms	16.78%
12	2	16.23%	2.14%	201ms	58.33%
12	3	16.10%	2.92%	320ms	41.67%
12	4	16.06%	4.00%	239ms	33.33%
12	5	16.15%	5.26%	440ms	28.33%
12	6	16.34%	6.96%	320ms	25.00%
12	7	16.14%	7.55%	321ms	22.62%
12	8	16.03%	8.19%	479ms	20.83%
12	9	16.07%	9.00%	480ms	19.44%
12	11	16.16%	10.78%	480ms	17.42%
13	2	15.99%	1.84%	161ms	57.69%
13	3	16.09%	3.02%	320ms	41.03%
13	4	16.27%	4.38%	319ms	32.69%
13	5	16.23%	5.81%	320ms	27.69%
13	6	16.11%	6.82%	479ms	24.36%
13	7	16.27%	8.00%	440ms	21.98%
13	8	16.20%	8.82%	479ms	20.19%
13	10	16.06%	10.50%	480ms	17.69%
13	11	15.95%	11.02%	481ms	16.78%
14	2	16.57%	2.16%	201ms	57.14%
14	3	15.93%	3.24%	239ms	40.48%
14	4	16.06%	4.55%	240ms	32.14%
14	5	16.00%	5.86%	480ms	27.14%
14	6	16.47%	7.67%	321ms	23.81%
14	7	15.87%	7.67%	480ms	21.43%
14	8	16.12%	9.25%	480ms	19.64%
14	9	16.18%	10.13%	480ms	18.25%
14	10	16.19%	10.94%	481ms	17.14%

Table 9: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
15	2	16.28%	2.04%	280ms	56.67%
15	3	15.94%	3.43%	280ms	40.00%
15	4	16.03%	4.78%	440ms	31.67%
15	5	15.95%	5.73%	480ms	26.67%
15	6	16.43%	8.29%	321ms	23.33%
15	7	16.10%	8.59%	400ms	20.95%
15	9	15.96%	10.26%	480ms	17.78%
16	2	16.24%	2.12%	320ms	56.25%
16	3	16.40%	3.89%	239ms	39.58%
16	4	16.23%	5.35%	320ms	31.25%
16	5	16.07%	6.53%	320ms	26.25%
16	6	16.07%	7.69%	479ms	22.92%
16	7	15.89%	8.55%	480ms	20.54%
16	8	15.83%	9.78%	481ms	18.75%
17	2	16.11%	2.45%	280ms	55.88%
17	3	15.91%	3.64%	280ms	39.22%
17	4	16.36%	5.43%	479ms	30.88%
17	5	15.93%	6.51%	320ms	25.88%
17	6	15.90%	7.84%	361ms	22.55%
17	8	16.09%	10.16%	481ms	18.38%
18	2	16.35%	2.56%	320ms	55.56%
18	3	16.33%	4.29%	280ms	38.89%
18	4	16.11%	5.76%	320ms	30.56%
18	5	15.86%	6.76%	321ms	25.56%
18	6	16.40%	8.61%	480ms	22.22%
18	7	15.94%	9.07%	480ms	19.84%
19	2	16.39%	2.43%	320ms	55.26%
19	3	16.39%	4.41%	479ms	38.60%
19	4	16.03%	6.03%	320ms	30.26%
19	5	15.93%	7.21%	400ms	25.26%

Table 10: Next 30 experiment results.

L	D	Original Loss	FEC loss	Repair Window	Overhead
19	7	15.92%	9.40%	481ms	19.55%
20	2	16.42%	2.64%	320ms	55.00%
20	3	16.03%	4.17%	440ms	38.33%
20	4	16.28%	6.30%	320ms	30.00%
20	5	16.17%	7.46%	480ms	25.00%
20	7	15.79%	9.50%	481ms	19.29%

Table 11: Last 6 experiment results.