Department of Informatics
University of Bergen

Master Thesis

Coded communication on the Internet

By: Ketil Mikalsen Kvifte
Supervisor: Øyvind Ytrehus

June 3, 2019

Abstract

The Internet is giving people many opportunities to interact, but for latency-sensitive activities, there is still room for improvement. It is not always good enough that messages arrive; some times they should arrive quickly. The User Diagram Protocol does not give any guarantees that the messages will arrive, and the Transmission Control Protocol only guarantees that it will arrive, not how fast it will arrive.

In this thesis, we explore how the combination of convolutional codes and Automatic Repeat-reQuests can help reduce the latency in our communications, and how memory maximum distance separable codes give better results than comparable random codes.

Acknowledgements

I want to thank my supervisor, Øyvind Ytrehus, for suggesting the topic, his patience and guidance during my project and for the help in wrapping it up for this master thesis.

Additionally, I would also like to thank Simula UiB for accepting me as a part of the group. I am grateful for all the stimulating conversations I have had with the people there.

Contents

List of Figures

List of Tables

Acronyms

ARQ Automatic Repeat-reQuest.

CDP Column Distance Profile.

CTCP Network Coded TCP.

mMDS Memory Maximum Distance Separable.

RTT Round-trip time.

TCP Transmission Control Protocol.

Chapter 1

Introduction

1.1 Motivation

Sending information over the Internet has become quite important. It is essential that any information sent is received correctly, and that it does not take too much time. There are several situations where latency plays a significant role. First, there are video conferences. If two people are talking, they generally want the communication to be two-way. If the time between someone saying something, until they see the listeners reaction to it is too high, the value of the conversation decreases. Second, there is trading stock in stock exchanges online. Generally, the buyer wants to buy when the price is low, and the seller wants to sell when the price is high. If the latency on the buy request or the sell request is too high, then the chance of trading at a bad rate increase.

In order to transfer information, we use a transport protocol. The User Diagram Protocol does not guarantee that the transmitted information arrives at all. Reliable transport protocols like the Transmission Control Protocol (TCP) keeps track of transmitted data, and retransmit lost packets if needed, but on channels with a high round-trip time, the time required to recover from erasures or errors can be too high.

TCP sees data as a stream of bytes[1]. To transmit data over a channel, the sender divides this sequence of bytes into variable sized chunks and label each chunk by a sequence number identifying where the first data byte in the chunk belongs in the data stream. Together with a TCP header containing this sequence number and other important information, this is sent as an IP-packet over the Internet. The receiver keeps track of received packets and sends accumulative acknowledgements denoting the sequence number of the next byte that they have not received yet.

One of the mechanisms for deciding when to retransmit a packet is a timeout. If the sender has not received an acknowledgement for it within some time, then it is retransmitted. This timeout needs to be larger than the round-trip time because there is no way for the receiver to acknowledge packets faster than this. Another mechanism for scheduling retransmission is when it receives multiple acknowledgements for the same packet. In this case, the receiver has received multiple packets out of order.

To get faster recovery than the round-trip time, the communicating parties need to do something not based on communicating back and forth. With the use of forward error correcting codes, the sender adds redundancy to ensure the receiver can recover from most erasures. Combining this with an Automatic Repeat-reQuest (ARQ) ensures that they get full reliability, also when there are more erasures than what the code is able to correct.

1.2 Objective

This thesis aims to investigate the latency impact of convolutional codes when combined with ARQ, for transmitting information over the Internet when we look at it as an erasure channel with a nonzero round-trip time. We also want to compare optimal codes with similar codes using random coefficients.

1.3 Outline

This thesis has the following structure:

Chapter 2 introduces the abstract algebra and coding theory needed to understand the thesis.

Chapter 3 introduces transport protocols and explains how we designed our software for performing our experiments.

Chapter 4 presents our results, discussing erasure patterns and latency.

Chapter 5 gives a conclusion to the thesis, with ideas for future work.

Chapter 2

Theory

In this chapter, we will give some theoretical background for the content of this thesis. We start with a basic introduction to abstract algebra and follow up with some basic coding theory.

2.1 Abstract algebra

This section contains definitions for groups, rings and fields, and vector spaces. We also discuss polynomial rings and finite vector spaces over finite fields. The definitions in the subsections are based on the definitions in [2].

2.1.1 Groups

Definition 2.1. A group $\langle G, * \rangle$ is a set G closed under a binary operation * satisfying the following conditions:

1. Associativity: For all $a, b, c \in G$, we have that $(a * b) * c = a * (b * c)$
2. Identity element: There exist an identity element $e \in G$, such that for all $x \in G, e * x = x * e = x$
3. Inverse element: For each $x \in G$ there exist an inverse element $x^{-1} \in G$ such that $x * x^{-1} = x^{-1} * x = e$

Definition 2.2. A group $\langle G, * \rangle$ is abelian if the operation $*$ is commutative.

The integers modulo n forms a group over addition, $\langle Z_n, +_n \rangle$, where we have the set $Z_n = \{0, 1, ..., n-1\}$ and the operation $+_n$ defined by $a +_n b = a + b \pmod{n}, a, b \in Z_n$. $+$ is the normal addition over the integers.

2.1.2 Rings

Definition 2.3. A ring $\langle R, +, \cdot \rangle$ is a set R closed under two binary operations $+$ and $\cdot$ which we call addition and multiplication satisfying the following conditions

1. $\langle R, + \rangle$ is an abelian group with identity 0.
2. Multiplication $\cdot$ is associative
3. Addition and multiplication are distributive. $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$ and $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

We will often denote multiplication with juxtaposition, so that ab will mean $a \cdot b$. The integers modulo n form a ring over addition and multiplication. We have $\langle R, +_n, \cdot_n \rangle$ over the set Z_n and $\cdot_n$ is defined by $a \cdot b = a \cdot b \pmod{n}$.

If R is a ring, then a polynomial over R is an infinite formal sum $\sum_{i=0}^{\infty} a_i x^i, a_i \in R$ where we have a finite number of nonzero a_i coefficients. The highest i, such that a_i is nonzero is the degree of this polynomial. Together with the operations polynomial addition and polynomial multiplication, the set of all these polynomials form the polynomial ring we call R[x]. Polynomial addition is defined as follows: Let $a, b \in R[x]$, where $n = deg(a) \geq deg(b) =$ m, then $a + b = \sum_{i=0}^{n} (a_i + b_i) x^i$. Polynomial multiplication is defined as $a \cdot b = d_0 + d_1 x + ... + d_n x^n + ...$ where $d_n = \sum_{i=0}^{n} a_i b_{n-i}$

If multiplication in R is commutative, then polynomial multiplication in R[x] is commutative as well, and if R has a multiplicative identity $1 \neq 0$, then R[x] has the same multiplicative identity 1. We will not use any rings without multiplicative identity.

2.1.3 Fields

Definition 2.4. A field $\mathbb{F} = \langle F, +, \cdot \rangle$ is a ring where the multiplicative operation is commutative, a multiplicative identity exist, and every nonzero element has a multiplicative inverse.

We will focus on finite fields, fields with finitely many elements. There are two types: prime fields and extension fields.

Prime fields are fields with p elements, where p is prime and can be constructed by the integers modulo p. We call them $\mathbb{F}_p$.

Extension fields have $q = p^m$ elements and consist of polynomials of degree strictly smaller than m in $\mathbb{F}_p[x]$. First, we need to find an irreducible polynomial P of degree m in $\mathbb{F}_p[x]$. A polynomial P is irreducible in $\mathbb{F}_p[x]$ when it cannot be factorised into two non-constant polynomials in $\mathbb{F}_p[x]$ of lower degree. When we have chosen a polynomial P, we define x as a zero of P, in other words, P(x)=0, and we generate all the nonzero elements of $\mathbb{F}_q$ by multiplying x by itself mod P.

Two different irreducible polynomials P_1 and P_2 of degree m, will yield slightly different fields, but they are isomorphic, so we will not be too concerned about which irreducible polynomial we choose as long as we are consistent. Two fields, $\mathbb{F}_q = \langle F, +, \cdot \rangle$ and $\mathbb{F}'_q = \langle F', +', \cdot' \rangle$ are isomorphic when there exists a bijective map $\phi : \mathbb{F}_q \to \mathbb{F}'_q$ such that $\phi(a + b) = \phi(a) +' \phi(b)$ and $\phi(a \cdot b) = \phi(a) \cdot' \phi(b)$ for all $a, b \in F$.

For $\mathbb{F}_2[x]$ we can store our polynomials efficiently as integers. We start by representing the polynomials as $\sum_{i=0}^{m} a_i x^i, a_i \in \mathbb{F}_2, m \in \mathbb{N}$. We then let a_i be the i-th digit counting from least significant to most significant digit in the base-2 representation of an integer. For example $19 = (10011)_2$ will represent $1 + x + 0x^2 + 0x^3 + x^4 = 1 + x + x^4$ in $\mathbb{F}_2[x]$. We can now add two polynomials by using xor on the integers. For $\mathbb{F}_p, p > 2$ we can still convert our polynomials to integers, but our additions won't be simple xors any more, so we should should probably store our polynomials $\mathbb{F}_p[x]$ as vectors over $\mathbb{F}_p$ instead.

2.1.4 Linear Algebra

Definition 2.5. A vector space V over a field $\mathbb{F}$ is an abelian group V together with a scalar multiplication mapping $\cdot : \mathbb{F} \times V \to V$ so that $a, b \in \mathbb{F}$, $\alpha, \beta \in V$ the following conditions are satisfied

- $a\alpha \in V$
- $a(b\alpha) = (ab)\alpha$
- $(a + b)\alpha = (a\alpha) + (b\alpha)$
- $a(\alpha + \beta) = (a\alpha) + (a\beta)$
- $1\alpha = \alpha$

We call the elements of V vectors, and the elements of $\mathbb{F}$ scalars. We will use 0 for both the zero-vector and the additive identity of $\mathbb{F}$.

We will most of the time talk about finite vector spaces that consist of an ordered list(n-tuples) of elements from a field $\mathbb{F}$. Let $\alpha = [a_1, a_2, ..., a_n]$, and $\beta = [b_1, b_2, ..., b_n]$ for $\alpha, \beta \in V$ and $a_i, b_i \in \mathbb{F}$. We define scalar multiplication $a_1\beta = [a_1b_1, a_1b_2, ..., a_1b_n]$ and vector addition as $\alpha + \beta = [a_1 + b_1, a_2 + b_2, ..., a_n + b_n]$.

A k-dimensional vector space has k linear independent vectors that span the space, and these form a basis β. This means that any vector $\alpha \in V$ can be described as $\alpha = c_1b_1 + c_2b_2 + ... + c_kb_k$ for $b_i \in \beta$. The vectors of β are linear independent if and only if $\alpha = 0 \implies c_1 = c_2 = ... = c_k = 0$. If the vectors has length n, then the vector space has dimension $k \leq n$. If a vector space V_1 contains all the vectors of another vector space V_2 then we say that V_2 is a subspace of V_1

A linear map $f : V_1 \to V_2$ between two vector spaces V_1, V_2 defined over the same field $\mathbb{F}$ is a mapping that preserves addition and scalar multiplication. In other words $f(a_1\alpha_1 + a_2\alpha_2) = f(a_1\alpha_1) + f(a_2\alpha_2) = a_1f(\alpha_1) + a_2f(\alpha_2)$ for $a_1, a_2 \in \mathbb{F}$, $\alpha_1, \alpha_2 \in V_1$.If V_1 has degree n and V_2 has degree m, then any linear map $V_1 \to V_2$ can be described by an $m \times n$ matrix.

2.2 Coding theory

In this section, we will give a short overview of linear codes in general and convolutional codes defined over finite fields in particular. This section is for the most part based on a combination of [3] and [4]. We are mainly interested in convolutional codes over finite fields $GF(2^m)$ with $m \leq 14$, but the theory presented here should still work for any finite field.

Given an information sequence u, we want to generate a sequence v from u in such a way that when we transmit it over a specific channel, the receiver can reconstruct u as quickly and accurately as possible from the received sequence. We do this by dividing the sequence u into blocks of length k and encode them into blocks of length n.

A k-dimensional code over a finite field $\mathbb{F}_q$ is a set of codewords consisting of q^k n-tuples over $\mathbb{F}_q$ for $k < n$ such that there exist a mapping from the set of k-tuples into the set of n-tuples. If this mapping is linear, then we have a linear code, and the codewords form a k-dimensional vector space over $\mathbb{F}_q$. We say that these codes have rate R=k/n.

We have two main classes of linear codes, linear block codes, and convolutional codes. For block codes we use the notation $u = (u_1, u_2, ..., u_k)$ to denote a single block. An encoder for a given linear block code is given by an $k \times n$ generator matrix G such that $uG = v = (v_1, ..., v_n)$. The code described by G will then be the set of all possible v given all possible u. Equivalently we can also describe the same code as an $(n - k) \times n$ parity check matrix H with the property that $vH^\top = 0$ for all codewords v in the code.

For convolutional codes we have multiple ways of denoting the sequences. We can either consider them as a semi-infinite vectors $u = (u_1^{(0)}, u_2^{(0)}, ..., u_k^{(0)}, u_1^{(1)}, u_1^{(1)}, ...)$ and $v = (v_1^{(0)}, v_2^{(0)}, ..., v_n^{(0)}, v_1^{(1)}, v_2^{(1)}, ...)$ or polynomials over the polynomial ring of a finite field $\mathbb{F}_q[x]$. The corresponding polynomials will then be $u(x) = (\sum_i u_1^{(i)} x^i, ..., \sum_i u_k^{(i)} x^i)$, and $v(x) = (\sum_i v_1^{(i)} x^i, ..., \sum_i v_n^{(i)} x^i)$.

We still encode blocks of k symbols at a time, but this time the generated sequence depends on the previous D information blocks as well. We say that a code consists of all possible sequences v generated by a corresponding encoder.

An encoder can be described as a shift register with k inputs and n outputs. With memory D, we need at most kD shift register stages. For binary codes, we only need exclusive-or gates, shift register stages and multiplexers. For non-binary codes, we use more generic addition gates instead of exclusive-or and also need gates for multiplying our inputs with elements from the finite field our code is defined over.

The shift register for an example code collected from [4] is drawn in Figure 2.1. Because the information only moves forward and never backwards through the shift registers, we call it a feed-forward convolutional code. Because each input sequence is identical to a corresponding output sequence, we call it systematic. If it is not systematic, then it is non-systematic. For the example code $u_1 = v_1$ and $u_2 = v_2$.

We can also describe our encoder by a $k \times n$ transform-domain generator matrix G(x), a matrix where our elements are polynomials in $\mathbb{F}_q[x]$, and encode as v(x) = u(x)G(x). We can then interpret our indeterminate x as a delay operator, i.e. $\alpha^4 x^2$ can be interpreted as α^4 multiplied by the element we multiplied this polynomial with two time-steps ago. We have that x^2 at time t in row i is $u_i^{(t-2)}$, which means that for $k \geq 2$ x^2 in one row of our polynomial matrix does not have to represent the same element as x^2 in another row.

Example: The code in Figure 2.1 can be represented as a polynomial matrix over $\mathbb{F}_{16}[x]$ with primitive element α. We will refer to this code again throughout this thesis.

$$G(x) = \begin{pmatrix} 1 & 0 & 1 + \alpha x + x^2 + \alpha^7 x^3 \\ 0 & 1 & 1 + x + \alpha^4 x^2 + \alpha x^3 \end{pmatrix} \tag{2.1}$$

Any systematic encoder contains the identity matrix I_k in the columns of G(x). From Equation 2.1, we see that our transform domain generator matrix G(x) is on the form (I | -R(x)) where I is the identity matrix, which means that it is systematic. Because we have characteristic 2, we have that -R(x)=R(x). The corresponding transform domain parity check matrix is on the form $(R(x)^\top | I)$ giving us Equation 2.2

$$H(x) = \begin{pmatrix} 1 + \alpha x + x^2 + \alpha^7 x^3 & 1 + x + \alpha^4 x^2 + \alpha x^3 & 1 \end{pmatrix} \tag{2.2}$$

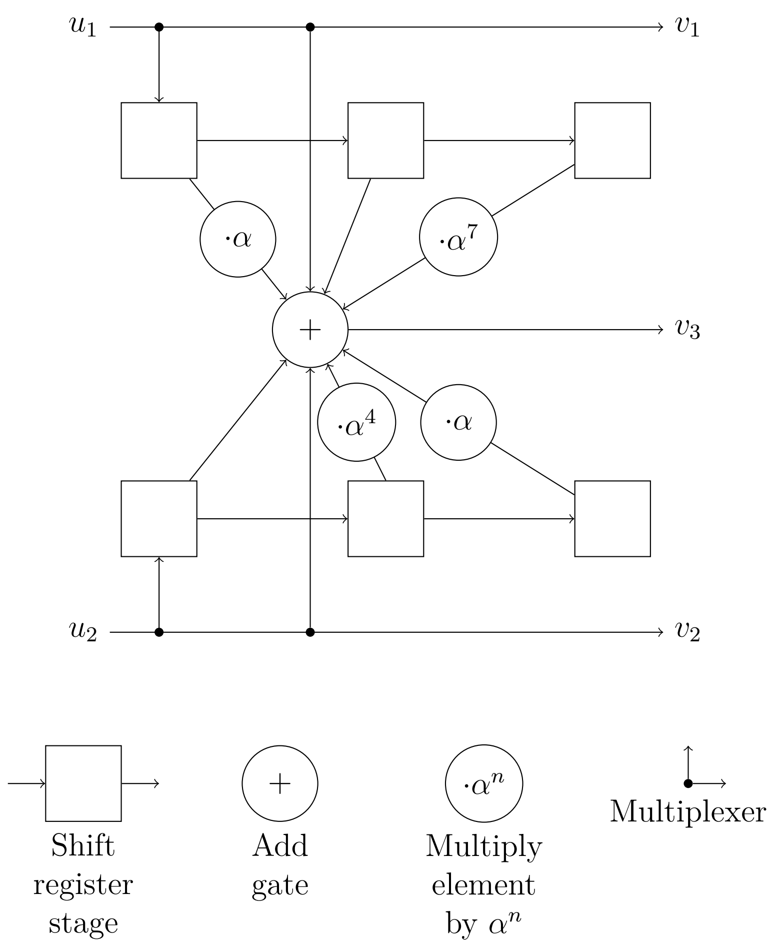


Figure 2.1: Shift register describing an encoder for a convolutional code defined over $\mathbb{F}_{16}$ with k=2, n=3, D=3. These shift registers are normally drawn with v_3 below v_2, but v_3 is drawn in the middle to make it easier to follow the arrows

The time-domain parity check matrix H for a convolutional code is a matrix over the finite field $\mathbb{F}_q$ such that $vH^\top = 0$ where v is the output sequence of the encoder and 0 is the all-zero sequence. This matrix is semi-infinite and we add rows to it until it fits the length of our output sequence. We construct H as in Equation 2.3. A semi-infinite time-domain generator matrix G also exist, but we will not use it, so we will not describe it here.

$$\begin{pmatrix} H_0 \\ H_1 & H_0 \\ \vdots & \vdots & \ddots \\ H_D & H_{D-1} & \dots & H_1 & H_0 \\ & H_D & H_{D-1} & \dots & H_1 & H_0 \\ &&&&&& \ddots \\ &&&&&&& H_D & H_{D-1} & \dots & H_1 & H_0 \end{pmatrix} \tag{2.3}$$

Each H_i is an $(n-k) \times n$ matrix and is shifted n position to the right compared with the previous $n-k$ rows. To simplify the description of the construction we will limit ourselves to systematic feedforward encoders where n=k+1. This simplification means that $R(x) = (\sum_{i=0}^{D} r_{i,1}x^i, ..., \sum_{i=0}^{D} r_{i,n-1}x^i)$. We then have that $H_i = (r_{i,1}, ..., r_{i,n})$. The last column of H_0 is 1, and the last column of H_i is 0 for $i > 0$. We can then construct the parity check matrix corresponding to G(x) in Equation 2.4.

$$H = \begin{pmatrix} 1 & 1 & 1 \\ \alpha & 1 & 0 & 1 & 1 & 1 \\ 1 & \alpha^4 & 0 & \alpha & 1 & 0 & 1 & 1 & 1 \\ \alpha^7 & \alpha & 0 & 1 & \alpha^4 & 0 & \alpha & 1 & 0 & 1 & 1 & 1 \\ &&& \alpha^7 & \alpha & 0 & 1 & \alpha^4 & 0 & \alpha & 1 & 0 & 1 & 1 & 1 \\ &&&&&& \ddots &&& \vdots &&& \vdots &&& \ddots \end{pmatrix} \tag{2.4}$$

If we let v' contain the received symbols of $D+i$ blocks, and let H' be the sub-matrix of H with row D to row $D+i$ from H, we can solve $H'v' = 0$ to find the missing symbols assuming we don't have too many erasures.

We can construct an Lth truncated block code $C^{(L)}$ as in Equation 2.5 where H_L is the zero matrix of dimension $(n-k) \times n$.

$$H^{(L)} = \begin{pmatrix} H_0 \\ H_1 & H_0 \\ \vdots & \vdots & \ddots \\ H_L & H_{L-1} & \dots & H_0 \end{pmatrix} \tag{2.5}$$

The free distance d_{free} of a convolutional code is defined by the minimum distance between two generated sequences given that the input sequences are different. The Column Distance Profile (CDP) is a non-decreasing sequence denoting the minimum distance of a codeword generated by the Lth truncated block matrix. For rate $n - 1/n$ systematic codes with memory D, we have $d_{free} \leq D + 2$ and the best column distance profile is $[2, 3, .., D + 2, D + 2, ...]$ with D+2 repeating. Codes with that CDP are Memory Maximum Distance Separable (mMDS) codes. An important property of these codes is that we can correct any j erasures in j blocks with a delay of j blocks when $j < D + 2$ [4]. Our example code G(x) is an mMDS code with column distance profile [2, 3, 4, 5, 5, ...].

Chapter 3

Methods

3.1 Transport protocol

3.1.1 Network Coded TCP

The paper [5] introduced a protocol for combining TCP with coding called Network Coded TCP (CTCP). This subsection gives an introduction to this protocol based on their paper.

Given a fixed packet size, and a fixed number of packets per block, the sender and receiver negotiate a maximum number of packets to hold in memory at any given time to ensure that the buffers at either end do not grow too large. The sender divides the data stream into packets and starts filling the first block until it has the desired number of packets before they start on the next block. If there is not enough data in the stream, then the sender can pad the last packet with zeros to make it fit.

This protocol uses a congestion control mechanism controlling the number of packets the sender is allowed to send. It uses a combination of Round-trip time (RTT) and packet loss to determine when the sender is permitted to send packets. This combination is done to distinguish between a packet loss caused by full queues and packet loss caused by interference on a noisy channel. If the link is underutilised, then the current round-trip time is likely to equal to the minimum round-trip time.

When the sender is permitted to send a packet, it picks a packet from one of the blocks in memory, favouring the first block if it estimates that the receiver needs more packets from that block to decode it. When sending a packet, the sender combines it with a header denoting the block number, a seed for a pseudo-random number generator used for generating coding coefficients, and packet sequence number.

The receiver sends acknowledgements for the smallest block not yet decoded, along with information about the number of degrees of freedom in this block, and the sequence number of the packet it has received.

CTCP uses systematic block codes. It starts by sending uncoded packages first, i.e., the information packets, and then add additional linear combinations of these with random coefficients, the parity check packets. It decides the number of parity check packets by estimating the packet loss and sends enough that the receiver probably can decode.

3.1.2 Our protocol

Our protocol uses systematic convolutional codes. The advantage convolutional codes have over block codes, is that we can use the memory to get the desired minimum distance with smaller block lengths. Because the receiver cannot recover from erasures until they have received enough parity check symbols, keeping the block size small can be beneficial for quicker recovery. Keeping it systematic means that we can decode instantly in the absence of erasures. When the channel only erases a few symbols, the receiver can recover quickly.

By combining this convolutional code with an ARQ, the code does not have to be strong enough to correct every likely error pattern, we just have to correct the most frequent patterns. This ARQ means that the protocol will need sequence numbers so that the receiver can refer to which symbols they want the sender to retransmit. For simplicity, these sequence numbers start at 0. The protocol uses selective repeat, where the receiver sends negative acknowledgements with the sequence number of the symbols they need to decode.

Another use for sequence numbers is so that the receiver can detect erasures. If there is an erasure, then they will detect a gap in the sequence number. For channels that reorder data, the receiver can use these numbers to put the received data in the right order.

3.2 Software

To test latency, we have been writing software using Python 3.7 and the Numpy library. We have also made a python module for finite field arithmetic in characteristic 2, and a module for solving linear equations over these fields, implementing Gaussian elimination.

We have a modular pipeline consisting of multiple processes: a data generator, an encoder, a sender/channel, a receiver/decoder. There is a queue between each process, and each process is running concurrently. Even though the channel and the ARQ sender are different concepts, the code for this is running in the same process because the operations they do are quick enough that it does not slow down the operations. The decoder and the ARQ receiver are also in the same process because they interact a lot with each other. In this section, we merge the description of things running in the same process into the same subsection because it describes the dataflow better.

3.2.1 Data generator

The data generator calculates the arrival time of new data by either a geometric distribution, a Poisson distribution, an exponential distribution, or at a constant rate satisfying a given arrival rate λ. The data itself is a random element from a given finite field. The generator also makes a timestamp. The element and the timestamp are passed on as input to the encoder.

3.2.2 Encoder

The encoder is set up using a predefined systematic convolutional code over the same finite field as the data from the generator. It uses the polynomial generator matrix, represented by a 3-dimensional $k \times n \times (D + 1)$ array of integers, where each integer represents elements of the finite field. During the encoding, it calculates the dot product between the coefficients of each polynomial in the generator matrix and the corresponding memory. It then sums up the rows. Because this encoder is for systematic codes only, it does not calculate the systematic part of each block, but rather append the parity check symbols to our information blocks before they are passed on.

If the encoder process picks up symbols to be encoded but has to wait longer than a predetermined time to get a full block, then it will add additional zero elements until it fits, so that it can encode them, and pass them on to the channel. These padding symbols have timestamp zero so that we can exclude them in our latency measurements, but the symbols from the data generator are included and delayed by this timeout. In retrospect there isn't any good reason for why the encoder can't pass through the information symbols immediately and only delay the generation of the parity check symbol, but with the parameters we run our code with, we don't add many zeros.

The output symbols are passed on together with the timestamps from before. The parity check symbol has timestamp zero.

3.2.3 Channel/ARQ Sender

The code for the channel logic and the ARQ sender logic run in the same Python process. It takes elements from the output queue of the encoder. First, it has a given probability of marking each symbol as erased. It marks the symbol as erased, rather than erase it to make the code for detecting erasures in the receiver simpler. Second, it adds a constant delay to each symbol. The constant delay is not ideal, but for simulating underutilised networks, it should be acceptable.

The channel puts a tuple of the arrival time and the packet in a minimum heap. If the first packet to arrive has a timestamp than is earlier than the current time it transmits the packet to the receiver. The packet consists of the timestamp, an element, and a marker for whether the packet is erased or a retransmission.

There is also a feedback queue from the ARQ-receiver to the ARQ-sender. This feedback queue is lossless for simplicity. Whenever there is a negative acknowledgement, it retransmits the requested symbol. Because the feedback channel has no delay, the channel double the RTT here to compensate, i.e. add a full RTT rather than a half to arrival time. The heap ensures they are delayed appropriately.

3.2.4 Decoder/ARQ Receiver

The decoder is implemented as an object with a decode method and a state in the ARQ receiver process. In order to detect retransmissions as early as possible, it processes the input queue as fast as possible. Retransmissions are processed immediately, updating the memory of the decoder, while it puts the new symbols into a local queue that is processed when the input queue is empty. Any information symbol received in sequence outside an erasure pattern is passed on immediately along with a timestamp saying when it was passed on.

The decoder is based on the semi-infinite parity check matrix of the code. It is initialised with the n(D+1) interesting columns of the first full row so that it can construct the semi-infinite parity check matrix later. The ARQ receiver feeds it one block at a time in sequence, and each symbol in a block is either the correct element or the separate symbol telling the decoder that the symbol has been erased. If the decoder is in a good state, and there are no erasures in the information symbols, then it only updates the memory before it returns.

It keeps track of information debt each time it gets a new block. The debt is increased by one for each erasure in the information symbols, and decreased by one for each parity check symbol not being erased, but never below zero. These debt calculations do not check for linear independence. Instead, they tell the decoder when it definitely cannot decode, preventing it from wasting time trying. It continuously keeps track of how many rows the parity check matrix would have to be to decode. When it gets a block, the debt is zero and there is at least 1 unknown information symbol, it then generates the parity check matrix with the correct number of rows, removes known variables from the equations, and use Gaussian elimination to see if it can recover the unknown variables. If the number of rows is at least 30, it tries to solve as little as possible at a time. Otherwise, it tries to solve everything at once. If it recovers the unknown variables, then it returns every information symbol it knows about and removes the parts of the memory not needed for subsequent decoding. If it solves some unknown symbols, but not all, it updates the memory, so that they will still be known at the start of the next decoding attempt.

If it fails to decode, then it gives the ARQ a list of unknown symbols that would have solved the equations if they were known. The ARQ receiver then sends negative acknowledgements back to the ARQ sender over the feedback channel, asking for these symbols.

When decoding succeeds it pass on the now known information symbols along with a timestamp of when the decoding finished. It does not check if any of these symbols were passed on earlier, but it passes on these symbols in a slightly different way so we can remove the duplicates in post-processing.

Chapter 4

Results

4.1 Error Patterns

In this section, we will discuss error patterns and discuss what a decoder can recover without retransmission.

Assume we receive the output sequence in Equation 4.1, where e means the channel has erased this symbol. The first and second columns are information symbols, while the third column consists of parity check symbols that are linear combinations of the information symbols from the same row, and the three rows above. Each row is a block encoded by an encoder described by the code in Equation 2.1 and reproduced here in 4.2.

$$\begin{bmatrix} \alpha^{14} & \alpha^{12} & \alpha^{3} \\ \alpha^{5} & \alpha^{10} & \alpha^{5} \\ \alpha^{13} & \alpha^{12} & \alpha^{4} \\ e & e & e \\ \alpha^{14} & \alpha^{3} & e \\ \alpha^{6} & e & \alpha^{6} \\ \alpha & e & e \\ \alpha^{12} & \alpha^{5} & \alpha^{12} \\ \alpha^{9} & \alpha^{4} & \alpha^{11} \\ \alpha^{3} & \alpha^{10} & \alpha^{14} \end{bmatrix} \tag{4.1}$$

$$G(x) = \begin{pmatrix} 1 & 0 & 1 + \alpha x + x^2 + \alpha^7 x^3 \\ 0 & 1 & 1 + x + \alpha^4 x^2 + \alpha x^3 \end{pmatrix} \tag{4.2}$$

Because we have 7 erasures, we get 7 equations with 7 unknowns. The 23 other symbols are known so we can remove them from the system of equations. Because the parity check symbol in the last row is a linear combination of 7 knowns, and 1 unknown information symbol, we can recover the information symbol in the 7th row. Having recovered that, we can also recover the information symbol in the 6th row by using the parity check symbol in the 9th row. We still have 3 parity check symbols and 2 information symbols that are unrecoverable. However, finding any of these will give us enough information to recover all of them. If we find the value of one of the missing information symbols, then the known parity check symbol in the 6th row will allow us to recover the other. If we find one of the parity check symbols, we will have 2 parity check symbols where both are linear combinations of 6 known and 2 unknown information symbols. Because this code has a column distance profile of [2, 3, 4, 5], we know that we can recover any pattern of 4 errors with a delay of at most 4 blocks.

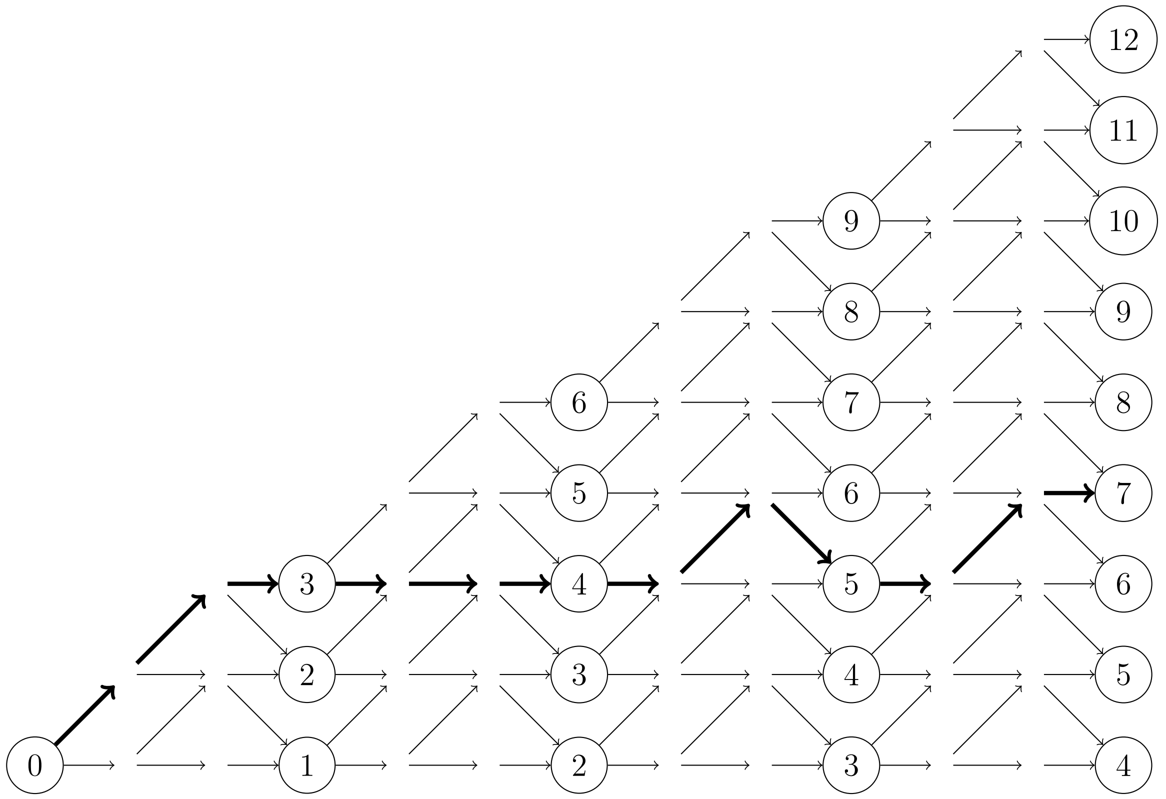


Figure 4.1: Trellis of erasure patterns for a (3,2) code with erasure probability ϵ . The bold path is the beginning of the non-recoverable erasure pattern in Equation 4.1

We have a trellis over the erasure patterns of (3,2) codes over a probabilistic erasure channel with erasure probability ϵ in Figure 4.1. Starting at a state 0 where we have decoded everything up to this point, we travel through the trellis based on the presence or absence of erasures. For each state vertex, we pick the topmost edge if we have an erasure, or the bottom edge if we do not. Notice that the parity check vertices on the bottom row only have one edge. We follow the same edge in both situations because the parity check symbol does not give us anything new. Whenever we reach a node marked t on the bottom, we have finished the current error pattern and go back to state 0.

We can then calculate the erasure probabilities for decoding an error pattern in time t: $P_e(t = T), T < d_{free}$. For t=1 we have 3 alternatives with 1 erasure and 2 non-erasures, and 1 alternative without erasures. For $t > 1$ we count all paths from the zero state to the t on the bottom that does not visit the bottom after the first erasure, so 2 or 3 of the first 3 symbols must be erased. If $t > d_{free}$, we will have to check if the parity check symbols are linear independent of the symbols we have from before.

For $t = 2$ we have two ways of reaching the second row, and 1 way to reach the third row counting from the bottom, so we get $P_e(t = 2) = 3\epsilon^2(1 - \epsilon)^4$. Counting the other paths the same way as t=2 gives us the following:

$$\begin{aligned}
&P_e(t = 1) = 3\epsilon(1 - \epsilon)^2 + (1 - \epsilon)^3 = 2\epsilon(1 - \epsilon)^2 + (1 - \epsilon)^2 \\
&P_e(t = 2) = 3\epsilon^2(1 - \epsilon)^4 \\
&P_e(t = 3) = 10\epsilon^3(1 - \epsilon)^6 \\
&P_e(t = 4) = 42\epsilon^4(1 - \epsilon)^8 \\
&P_e(t = 5) = 198\epsilon^5(1 - \epsilon)^{10}
\end{aligned}$$

So far, this match the following pattern:

$$P_e(t) = \frac{2}{t}\binom{3t-3}{t-1}\epsilon^t(1 - \epsilon)^{2t}, t \geq 2 \tag{4.3}$$

Using software to count the paths the the erasure trellis for $n \in \{2, 3, 4, 5, 6\}$ we see that the formula 4.4 looks correct for any code with $k = n - 1$ as long as t is lower than $\mathrm{d_{free}}$. With n=2, k=1, we see that the Catalan number $C_{t-1} = \frac{1}{t}\binom{2(t-1)}{t-1}$ shows up as one of the factors.

$$P_e(t) = \frac{k}{t}\binom{n(t-1)}{t-1}\epsilon^t(1 - \epsilon)^{kt}, t \geq 2 \tag{4.4}$$

4.2 Simulations

In this section, we will present some results from some simulation. We perform all simulations on a laptop with Intel i5-4210H processor and 8GB ram.

Our first simulation uses a data generator generating symbols at a constant rate 4400 symbols/second with a constant erasure probability 0.1. We use a constant channel delay of 25ms using an optimal code over $\mathbb{F}_{16}$ with the following generator matrix.

$$G(x) = \begin{pmatrix} 1 & 0 & 1 + \alpha x + x^2 + \alpha^7 x^3 \\ 0 & 1 & 1 + x + \alpha^4 x^2 + \alpha x^3 \end{pmatrix} \tag{4.5}$$

The data generator terminates after 30.076 seconds after generating a total of 132,176 elements. The decoder finish decoding them after 30.090s. Our simulated channel transmits 178,511(90.04%) symbols successfully and erase 19,753(9.96%) when we we do not count retransmissions.

Information about the latency distribution of the run is given in Table 4.1 and plotted in Figure 4.2. During this simulation, we have 14 retransmitted symbols. These are plotted as red dots in our latency plots, using the latency measured for this symbol.

Notice that one of the red dots has lower latency than 1.5RTT=75ms. This means that we requested a retransmission, but managed to decode the symbol we asked for before the retransmitted symbol arrived.

We also measure the discrete recovery time by counting how many information symbols the decoder gives us for each block. We compare this with the calculated results from Equation 4.3 in the previous section giving us the erasure probabilities in Table 4.2 and the counts in Table 4.3. We see that the values we calculate are relatively close to the formula for small t, but when t grows larger there are two sources of error. One is that for $t > d_{free} = 5$ we do not consider the linearly dependent parity check symbols in our formula. This means our probabilities are likely to be too optimistic. We also have a decreasing number of observations when t gets larger. Just because we measure equally many error patterns recovered in time 8 as 10 does not mean they are equally likely. Finally, these measurements does not distinguish between an error pattern that was recovered by a retransmission and an error pattern that was recovered by parity checks in the code.

	u_1	u_2
min latency	25.597	25.434
max latency	275.421	275.194
avg latency	44.578	44.443
median latency	37.693	37.574

Table 4.1: Recorded statistics on the latency distribution

	$P_e(1)$	$P_e(2)$	$P_e(3)$	$P_e(4)$	$P_e(5)$	$P_e(6)$
simulated	0.972104	0.020000	0.005091	0.001576	0.000536	0.000299
calculated	0.972000	0.019683	0.005314	0.001808	0.000690	0.000283
	$P_e(7)$	$P_e(8)$	$P_e(9)$	$P_e(10)$	$P_e(11)$	$P_e(12)$
simulated	0.000189	0.000047	0.000032	0.000047	0.000063	0.000000
calculated	0.000121	0.000054	0.000025	0.000011	0.000005	0.000003

Table 4.2: Table comparing expected results with measured results

Length	1	2	3	4	5	6	7	8	9	10	11	12-15	16
Count	61681	1269	323	100	34	19	12	3	2	3	4	0	1

Table 4.3: Table of erasure pattern lengths

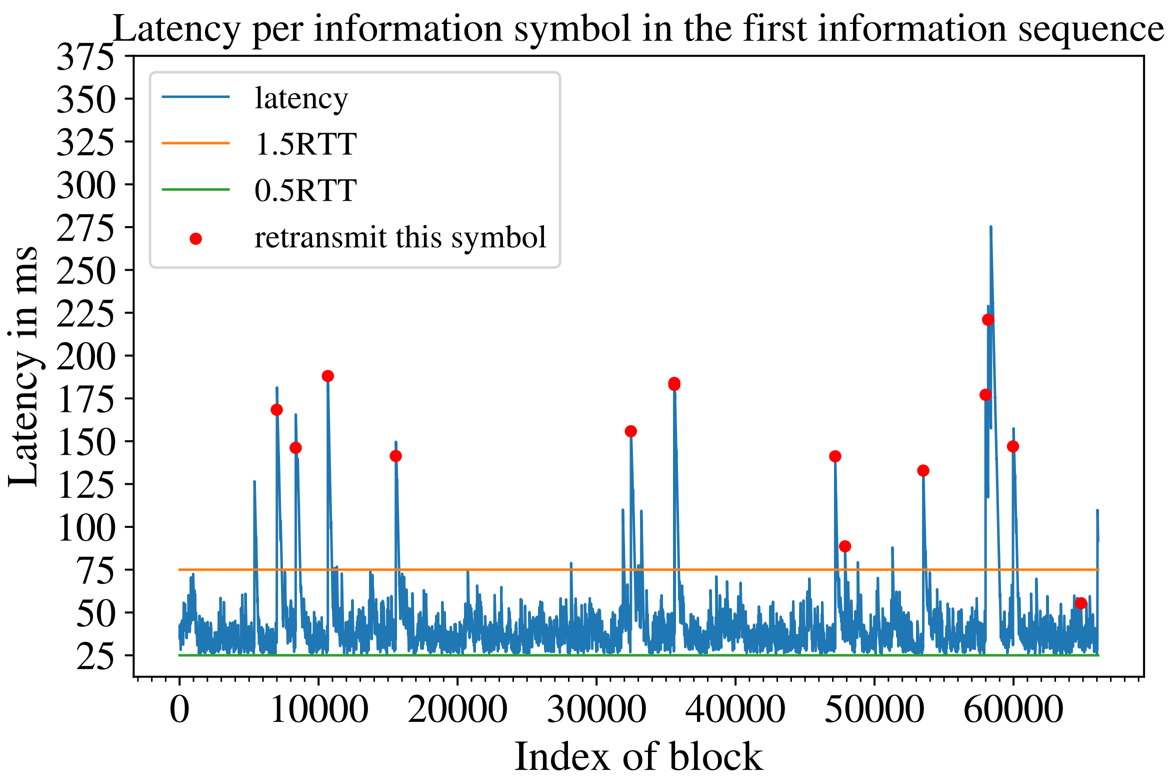


Figure 4.2: Plot of simulation on an optimal code

4.2.1 Comparison with random code

We also have a way to run our simulations for codes with random coefficients. In the following simulations, we use identical parameters with the exception of the code coefficients used. Both codes are defined over the same field $\mathbb{F}_{16}$. We simulate a channel with a constant one-way latency of 25ms. The latency distributions are plotted in Figure 4.3. We use the mMDS code from Equation 4.5 in Figure 4.3a while we use a code with random coefficients in Figure 4.3b. The generator matrix for this random code is given in Equation 4.6. Further results are given in Table 4.4.

	random u_1	random u_2	optimal u_1	optimal u_2
min latency	26.770	26.645	26.414	26.187
max latency	252.603	252.603	203.429	203.202
avg latency	50.010	49.915	51.297	51.199
median latency	42.213	42.130	44.585	44.496

Table 4.4: Table comparing latency of random code and optimal code

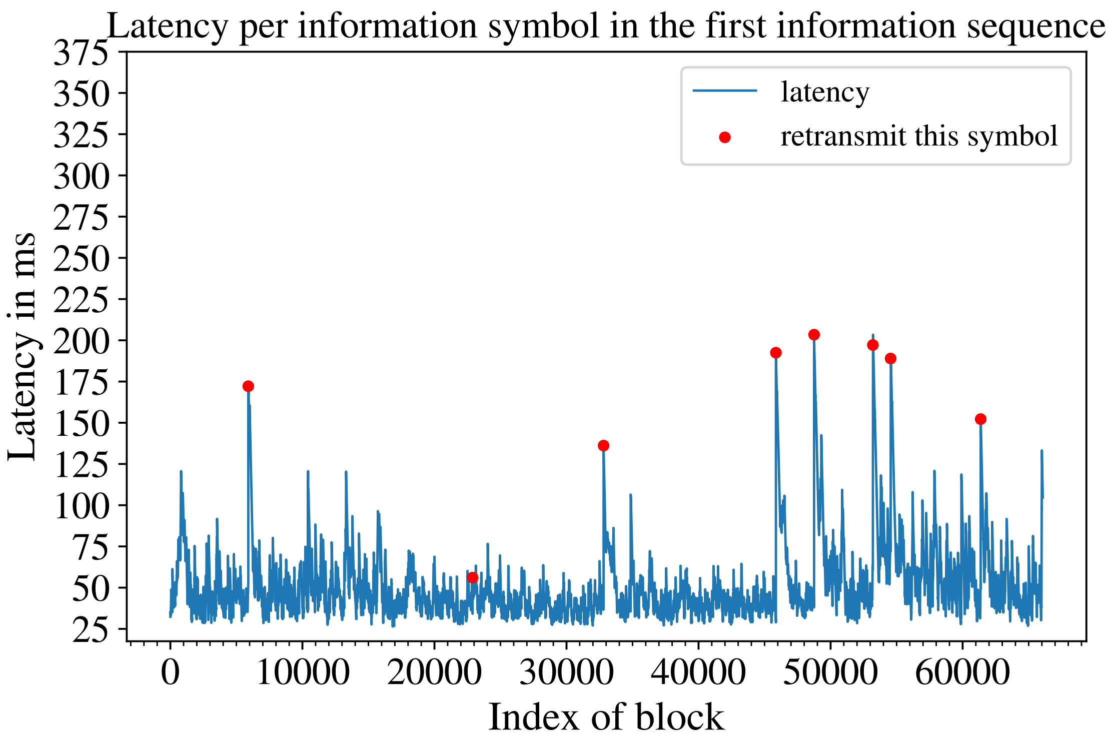
$$G(x) = \begin{pmatrix} 1 & 0 & \alpha + \alpha^9 x + \alpha^9 x^2 + \alpha^3 x^3 \\ 0 & 1 & \alpha^{10} + \alpha^4 x + \alpha^{10} x^2 + \alpha^3 x^3 \end{pmatrix} \tag{4.6}$$

As we can see from Figure 4.3, there are more retransmissions for the random code. This is because our optimal code is chosen for its distance properties given the [n, k, D] parameters in our chosen field. The random codes are less likely to generate linearly independent parity check symbols.

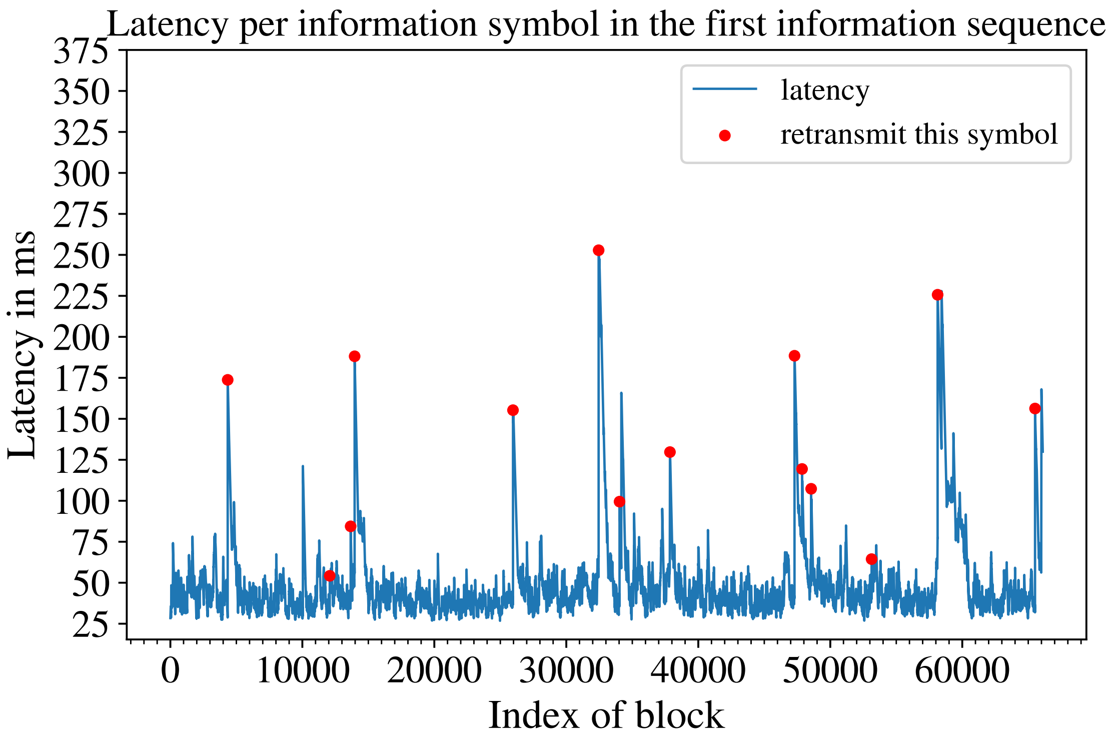
In Table 4.5 we see that the simulations with the mMDS code has more of error patterns of length 1 and 2, while the random code has more error patterns of length 4 to 6. This is what we expect to see, with a good column distance profile, we recover from erasures faster.

Length	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Count random	61522	1305	336	114	48	20	6	3	1	1	3		1	1
Count mMDS	61697	1258	336	107	43	12	7	2	1	1	1	2		1

Table 4.5: Table comparing erasure pattern lengths of different codes



(a) Optimal code



(b) Random code

Figure 4.3: Comparison of simulation random coefficients and optimal coefficients

4.2.2 Different erasure rates

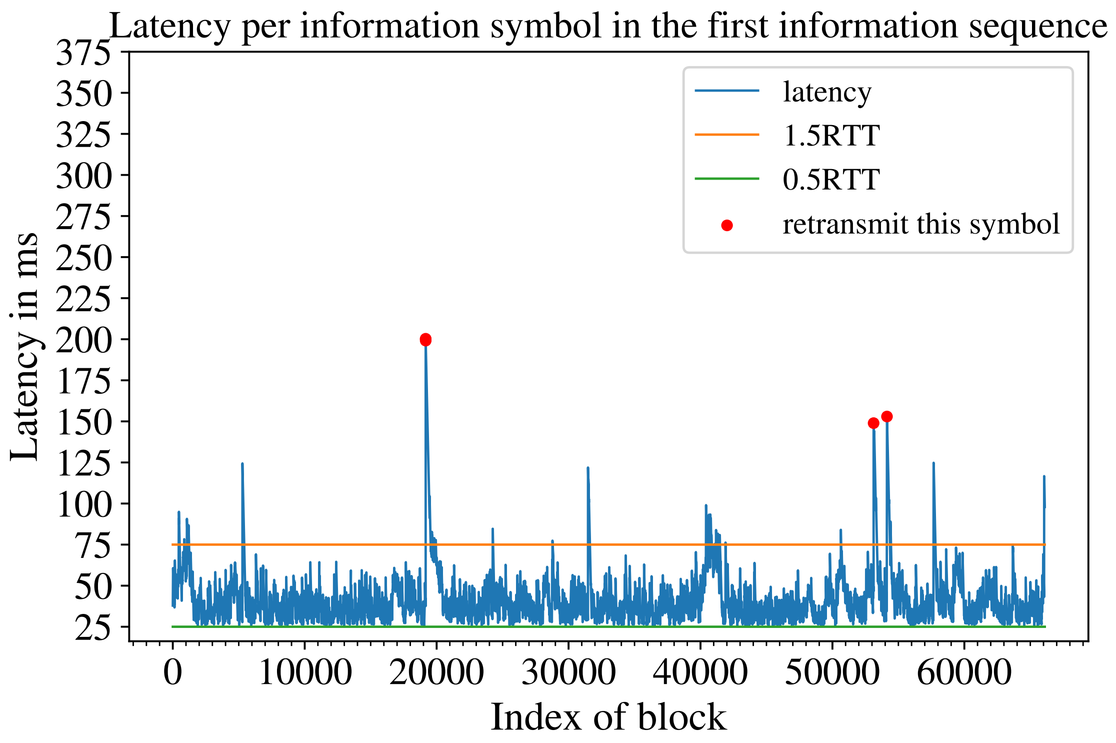
We can also run our simulations with different erasure rates. Again we use the mMDS code in our simulations. We compare the performance with erasure rate $\epsilon = 0.08$ with the performance with erasure rate $\epsilon = 0.12$. The $\epsilon = 0.08$ plot is smoother than the $\epsilon = 0.10$ plots from earlier, and the plot with $\epsilon = 0.12$ has more peaks. As we expect lower erasure rate give shorter recovery time. We give some statistics in Table 4.6 and we count the length of the erasure patterns in Table 4.7.

	u_1 for $\epsilon = 0.08$	u_1 for $\epsilon = 0.12$
min latency	25.595	26.018
max latency	200.289	367.761
avg latency	41.648	68.234
median latency	37.702	54.500

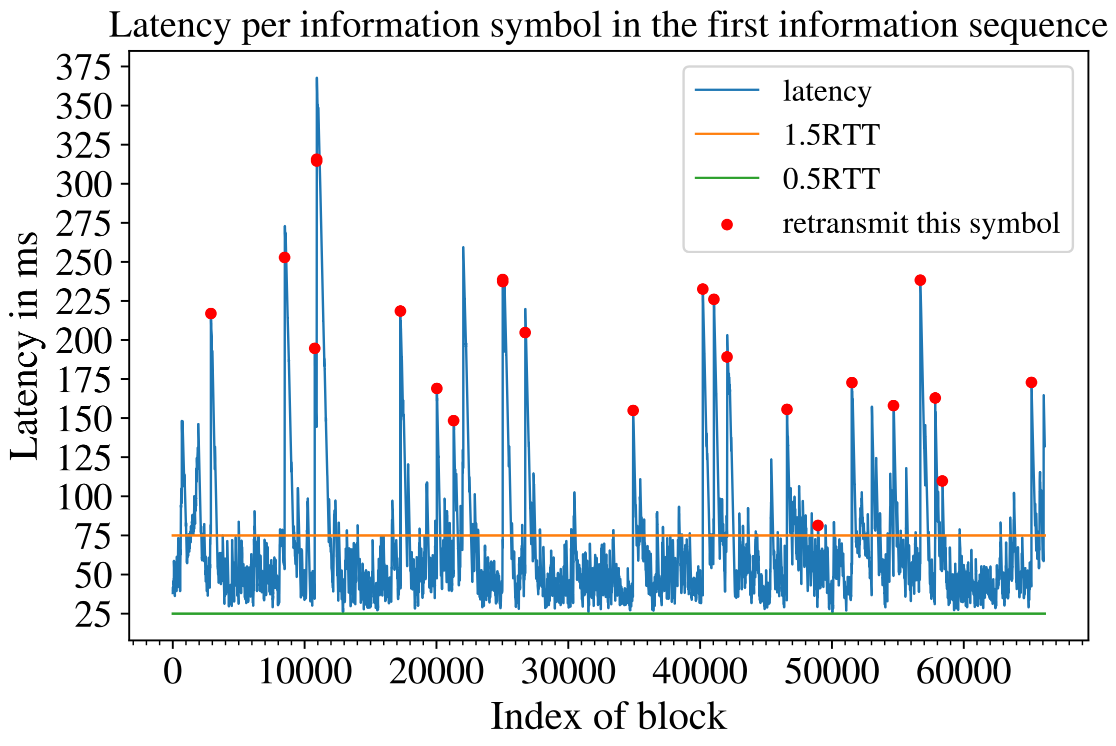
Table 4.6: Table comparing latency for different erasure rates

Length	1	2	3	4	5	6	7	8	9	10	11	13	14	18
Count $\epsilon = 0.12$	59636	1655	486	198	62	33	18	13	4	4	2	2	5	1
Count $\epsilon = 0.08$	63295	889	204	62	18	4	2		3		1			

Table 4.7: Table comparing erasure pattern lengths with different erasure rates



(a) Erasure rate 0.08



(b) Erasure rate 0.12

Figure 4.4: Comparison of simulation with different erasure rates

Chapter 5

Conclusion

In this thesis, we have tested the performance of some rate 2/3 convolutional codes on a relatively simple erasure channel with a few different erasure rates. We have seen that the mMDS code offer better latency, and require fewer retransmissions compared to similar random codes on channels with relatively high erasure rates.

There are a few improvements that would have made this project better. First, it would be a good idea to replace the flags marking data as erased or retransmitted and instead detect it at the receiver. With this fixed, we could split our software into two separate programs, a sender and a receiver that can run on different computers. Testing this on a real, realistic network with at least one wireless link between them would have given more relevant results. For this, we would not have a lossless feedback channel any more, so we would need a bit more logic to handle this. The main reason we did not do this was that implementing the decoder took more time than expected. For realistic networks, we should also implement a congestion algorithm, probably something similar to the one in CTCP.

Another potential improvement is to improve the way we decide when to ask for retransmissions. Because we had a few retransmissions that were not needed, we wasted bandwidth, but considering that the number of retransmissions was relatively low, this is not too important.

Bibliography

[1] Keith W. Ross James F. Kurose. *Computer Networking: A Top-Down Approach, International ed of 6th revised ed.* Pearson Education Limited, 2012. ISBN: 9780273768968.

[2] John B. Fraleigh. *First Course in Abstract Algebra, A: Pearson New International Edition, Seventh Edition.* Pearson Education Limited, 2013. ISBN: 9781292024967.

[3] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004. ISBN: 0130426725.

[4] Á. Barbero and Ø. Ytrehus. "Rate $(n-1)/n$ Systematic Memory Maximum Distance Separable Convolutional Codes." In: *IEEE Transactions on Information Theory* 64.4 (Apr. 2018), pp. 3018–3030. ISSN: 0018-9448. DOI: 10.1109/TIT.2018.2802540.

[5] Kim Minji et al. "Network Coded TCP (CTCP)." In: (Dec. 2012).