Analysis of LT-codes and Raptor Codes In Binary Erasure Channels with Benchmark Simulations and Real-world Applications

Dhyey Thummar, Neeraj Jadhav, Pratyush Sahu

December 12, 2024

1 Introduction

In the modern age of the Internet, the usage and consumption of audio and video as types of media has become a necessity for common activities (entertainment, information, critical research, etc.). Ranging from transmission of small media files within a LAN to using satellite communication for transmitting live media content across the world, such files communicated must ensure two important attributes for ease of use:

1. Quality of Service (QoS)
2. Efficiency (Speed)

With the growing usage of wireless networks, there has been a lot of work done on ensuring reliable transmission through Binary Erasure Channels (BECs) that result in noisy erasure of bits, leading to either longer times to receive and decode, or the inability to decode messages altogether. Erasure Correction Codes such as LT-codes and Raptor Codes are now prevalently used for such communication, as they allow (by being part of the family of Digital Fountain codes) recovery of lost symbols without the need of retransmission. Digital Fountain codes are a class of erasure correcting codes that are known to be rateless: can potentially generate an infinite amount of source symbols to be transmitted. This property ensures that given a subset of these symbols originating from the source (derived from original symbols through some method), the original message can be recovered by the receiver with the restriction of the size of the subset being only a small percentage larger than the number of original source symbols.

In other words, with k source symbols, only $k(1+e)$ encoded symbols will be required to recover the original source symbols with high probability, where e is the extra symbols.

2 LT Codes

LT Codes are a type of Forward Error Correction (FEC) code, specifically categorized as Fountain Codes, known for being rateless. Introduced in 2002, LT Codes were the first practical implementation of Fountain Codes. These codes are relatively simple to understand and implement.

Algorithm 1 A general LT encoding algorithm

```
1: procedure LTENCODE
2: repeat
3:     Choose a degree $d$ from degree distribution $\rho(d)$
4:     Choose uniformly at random $d$ input nodes $m(i_1), m(i_2), \ldots, m(i_d)$
5:     $c(i) \leftarrow m(i_1) \oplus m(i_2) \oplus \cdots \oplus m(i_d)$
6: until enough output symbols are sent
7: end procedure
```

Algorithm 2 A general LT decoding algorithm

```
1: procedure LTDECODE
2: repeat
3:     repeat
4:         Receive a packet
5:     until a degree-one check node $c_n$ is available
6:     $m_k \leftarrow c_n$ {Message node $m_k$ is the same as $c_n$}
7:     for all $c_i$ connected to $m_k$ do
8:         $c_i \leftarrow m_k \oplus c_i$
9:     end for
10:    Remove all edges between $m_k$ and check nodes
11: until the original message is fully recovered
12: end procedure
```

2.1 Optimization of LT Codes

Apart from the implementation and/or hardware optimizations, from theoretical standpoint the performance of LT Codes depends on the degree distribution function used. In order to optimize the encoding process to remove the redundant blocks one can use:

- **Ideal Soliton Distribution:** Minimizes redundancy but has higher chances of decoding failure. The probability mass function is given by : **??**

$$p(1) = \frac{1}{K},$$
$$p(i) = \frac{1}{i(i-1)}, \quad (i = 2, 3, \ldots, K).$$

- **Robust Soliton Distribution:** Introduces modifications to improve reliability by increasing the number of degree-one packets. The extra set of values, t(i), are defined in terms of an additional real-valued parameter δ (which is interpreted as a failure probability) and c, a constant parameter. Define R as $R = cln(K/\delta)\sqrt{K}$. Then the values added to p(i), before the final normalization, are **??**

$$t(i) = \frac{R}{iK}, \qquad (i = 1, 2, \ldots, K/R - 1),$$
$$t(i) = \frac{R\ln(R/\delta)}{K}, \qquad (i = K/R),$$
$$t(i) = 0, \qquad (i = K/R + 1, \ldots, K).$$

2.2 Implementation

The LT code implementation (`github.com/anrosent/LT-code`) that we used for our benchmarking utilizes Python for simplicity with Robust Soliton Distribution optimization.

2.2.1 Channel Simulation

In order to benchmark we tried to develop our own channel simulator (`FEC-code-Bench`) that aims to test LT Code performance under various noisy conditions (refer to Fig. 1). The simulation involves encoding input data into blocks, passing them through a simulated noisy channel, and reconstructing the original data from the received blocks.

We used `aff3ct` toolbox as our inspiration.

The channel types simulated:

1. **No Drop (Baseline):** All blocks are received in order without losses.
2. **Random Block Rearrangement:** Blocks are received in a different order.

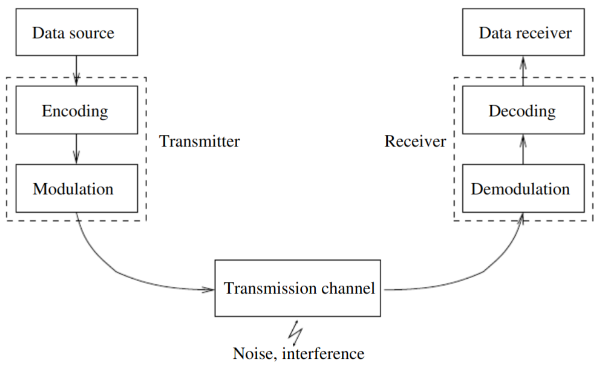


Figure 1: Simulation Channel

3. **Binary Erasure Channel (BEC):** Randomly drops blocks based on a uniform probability threshold.
$$Y_i = \begin{cases} \text{erased}, & \text{if } e = 1, \\ X_i, & \text{else,} \end{cases} \quad \text{with } P(e=1) = p_e \text{ and } P(e=0) = 1 - p_e$$
4. **Additive White Gaussian Noise (AWGN):** Drops blocks based on Gaussian noise thresholds.
$$Y = X + Z \quad \text{with } Z \sim \mathcal{N}(0, \sigma^2)$$
5. **Rayleigh Fading Channel:** Models the effect of a propagation environment on wireless signals, causing some blocks to be dropped.
$$Y = X \cdot H + Z \quad \text{with } Z \sim \mathcal{N}(0, \sigma) \text{ and } H \sim \mathcal{N}\left(0, \frac{1}{\sqrt{2}}\right)$$

2.3 LT-Code Benchmark

The LT-Code benchmark evaluates the efficiency and effectiveness of LT Codes under different scenarios. The evaluation is performed using a Python-based simulation setup.

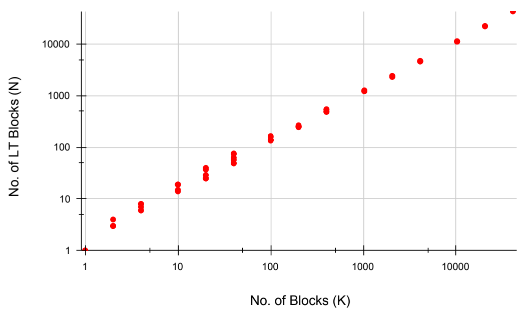
2.3.1 Parameters

Our simulation tests the following parameters:

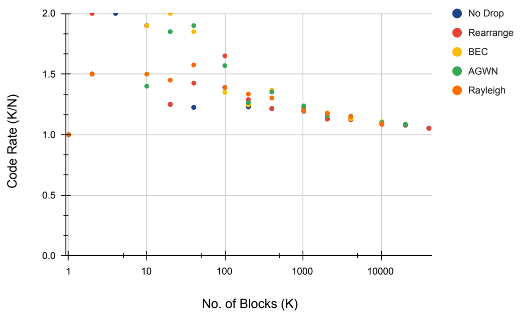
- **File Size:** Tested with files of sizes 1KB, 10KB, 100KB, 1MB, and 10MB.
- **Block Size:** Configured as 256, 512, or 1024 bytes, with a maximum of 50K blocks.
- **Channel Modes:** No Drop, Random Block Rearrangement, BEC, AWGN, and Rayleigh.
- **Metrics:** Number of LT blocks required to decode, encoding time, decoding time, and code rate (efficiency).
- **Iterations:** Each test was repeated 5 times to calculate the average results.

2.3.2 Results And Observations

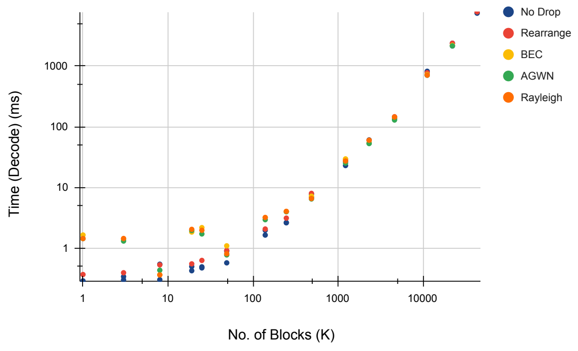
- Increasing the number of blocks reduces the code rate (K/N) (refer to Fig. 2b), thereby improving efficiency. Minimum observed efficiency was 1.05.
- Encoding and decoding time are dependent on the number of blocks (refer to Fig. 2c). However, due to the small time units (microseconds), a correlation with block size could not be conclusively determined.
- The order of blocks affects the decoding process. Random rearrangement or dropping blocks alters the number of blocks required to decode.
- The number of blocks required for decoding is primarily influenced by the percentage of dropped blocks and is agnostic to the channel type.



(a) No.of Blocks (K) v/s No. of LT Blocks Required



(b) No.of Blocks (K) v/s Code Rate (K/N)



(c) No.of LT Blocks (N) v/s Time to Decode

Figure 2: Simulation Channels for Different Scenarios

3 Raptor Codes

Raptor codes are a type of forward erasure correction code. They're also fountain codes (or rateless erasure codes). The specialty of Raptor codes is that they support linear time encoding and decoding, which is better than LT codes, having an additional log dependency on the number of source symbols.

There are two RFCs on raptor codes (RFC 5053 and RFC 6330). We proceed with using RFC 5053 for our project. Since there wasn't any available implementation in Python, we translated a Golang implementation to suit our use case.

3.1 Theoretical background

3.1.1 Encoding

Raptor codes are based on LT codes. They use two layers of codes namely Pre-code (or outer-code) and Inner-code. Pre-code is a fixed-rate erasure code (with a fairly high code rate) for example, HDPC from binary gray sequence along with some LDPC. Inner-code is just a LT code layer.
Pre-coding is done in such a way that the following constraint holds:
`Source symbol[i] = LTEnc[K, Intermediate Symbols, Triplet(Source symbol[i])]`
After adding the inner-code layer, we get:
`Encoding Symbols = LTEnc[..., Intermediate Symbols, ...]`

3.1.2 Decoding

There are two steps involved in decoding i.e. removing the inner code and the outer code. From the encoding part, we have a set of equations amongst encoding symbols and intermediate symbols. These can be represented as a binary matrix inversion problem. We solve these to obtain the intermediate symbols from encoding symbols (i.e. removing inner-code)

Finally, as discussed in pre-coding, retrieval of source symbols from intermediate symbols is a straight-forward LT encoding layer.

4 Raptor code Benchmark

The Raptor code benchmark evaluates the efficiency and effectiveness of Raptor codes under different scenarios. The evaluation is performed using a Python-based simulation setup.

4.1 Parameters

The simulation tests the following parameters:

- **File Size:** Tested with files of sizes 1KB, 10KB, 100KB, 1MB, and 10MB.
- **Block Size:** Configured as 256, 512, or 1024 bytes, with a maximum of 50K blocks.
- **Channel Modes:** No Drop, Random Block Rearrangement, BEC, AWGN, and Rayleigh.
- **Metrics:** Number of Raptor blocks required to decode, encoding time, decoding time, and code rate (efficiency).
- **Iterations:** Each test was repeated 5 times to calculate the average results.

4.2 Results

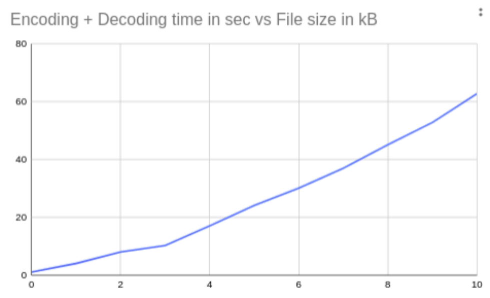


Figure 3: Linearity of Raptor codes (Block size = 256, File size in multiples of 100kB)

Key Observations:

- Encoding and decoding time are dependent on the number of blocks and increase linearly with the number of blocks.
- The order of blocks affects the decoding process. Random rearrangement or dropping blocks alters the number of blocks required to decode.
- The number of blocks required for decoding is primarily influenced by the percentage of dropped blocks and is agnostic to the channel type.
- Although Raptor encoding+decoding time grows slowly with increase in number of source symbols, the constant factors are still pretty high reducing its usability compared to LT codes for small file sizes (since video frames are still pretty small in size).

5 Frame-Dependent Video Stream Encoding: Efficient MP4 Video Codec Transmission Using LT-Codes and Frame Scene Changes

5.1 Initial Approach

Through the observations from the LT-Code benchmark, as well as theoretical limits described in papers Luby (2002) Wu, Guan, and Cui (2021) Silas (2021), it has been observed that the LT encoding and decoding, in the worst case, is $O(k^2)$ (k being the number of input symbols) and on average is not linear (for Belief Propagation Decoding, it is expected $O(klog(k))$). This leads to suboptimal time complexity for video streaming or transfer of video files. Hence, with growing k (proportional to the number of frames of a video file), it is necessary to bring down the time complexity while still restricting the loss of

frames (for optimal QoS).
The initial idea to enforce a more efficient scheme of erasure correction was to use a subset of the k input symbols, such that, the number of encoded symbols $(t) \leq 0.01k$, leading to a constant time reduction in k.
However, instead of randomly choosing $t \subset k$, we pick the optimal frames that are the most important for the video's QoS.
To get started with a standard approach, we chose to use the mp4 container Fernandez (2020) template, and the H264 Video Codec, both used as standards, and hence, allowing for best compatibility.
The H264 video codec (also known as Advanced Video Coding) Wikipedia contributors (2024a), is the video compression standard that utilizes a partially lossless compression that divides videos into separate frames of categorized into different types. For loss resilience and better organization, it uses NAL units that correspond to at most a single frame of the video. These frames are primarily classified into 3 categories OTTVerse Team (2022):-

1. **I-Frames**: These are standalone frames that are not compressed. This implies they contain the entire information (pixel values, for instance) of the frame, and can be decoded into a complete standalone image. They are sparsely present, with about 5% to 15% of the total number of frames.

2. **P-Frames**: These are frames that are extensively compressed, and only contain the changes from the previous frame in order. They cannot be decoded into standalone images.

3. **B-Frames**: These are further compressed as they contain only changes from both the previous frame and the following frame. They again cannot be decoded into standalone frames.

Evidently, the I-frames are the most important frames affecting the QoS of the video to be decoded. Moreover, the sparsity of the frames implies there encoding and decoding only I-Frames will have a reduction in time complexity as compared to encoding and decoding all frames.

5.2 Current Approach

While the above idea lends itself to an efficient process that is optimized for QoS, it presented some challenges, making it infeasible to implement it:-

1. There doesn't exist a well-defined tool (such as FFMPEG) that allows users to extract raw frame information (or raw NAL units) that contains the frame type information without significant utilization of the AVCodec library. This would mean non-trivial changes in the source-code of FFMPEG itself.

2. The definition of the I-Frames (and P,B-Frames) are not defined as a standard, leading to variations depending on what tool is used to inspect the videos.

Given the above issues, we implement a variation of the above idea. We make use of scene changes, defined as:-

1. **High scene-change frames**: Frames that have a high difference (with respect to a given threshold) from the previous frame in sequence.

2. **Low scene-change frames**: Frames that have a low difference (with respect to a given threshold), in comparison to the previous frame in sequence.

5.3 Proposed Algorithm

We propose the following algorithm:-

Algorithm 3 EncodeVideo

```
Require: $video\_file$: String
Require: $fps$: Integer
Require: $t$: Float
Ensure: $k' \subset total\ number\ of\ frames(k)$
1: $k = \emptyset$
   $k \leftarrow$ SplitVideoIntoFrames($video\_file$, $fps$)
   $k' \leftarrow$ FilterLowSceneChangeFrames($k$, $t$)
2: for $i \in k$ and $i \notin k'$ do
     LTEncodeAndTransmit($i$)
3: end for
4: for $i \in k'$ do
     RawTransmit($i$)
5: end for
```

Algorithm 4 DecodeVideo

```
Require: $LTDecodedFrames$: [String]
Require: $NonLTEncodedFrames$: [String]
   $Frames \leftarrow$ Sort($LTDecodedFrames$, $NonLTEncodedFrames$)
   $RenumberedFrames \leftarrow$ ReNumberFrom0($Frames$)
   $Output\_Video \leftarrow$ AssembleVideoFromFrames($RenumberedFrames$)
```

The functions `SplitVideoIntoFrames` and `FilterLowSceneChangeFrames` are implemented using the FFMPEG tool FFmpeg Team (2024a) (which in turn makes use of the AVCodec library FFmpeg Team (2024b) implemented in C).

The algorithm works as follows: -

1. We take in as input the video file, frames-per-second (fps) parameter, and the threshold parameter.
2. Given the video file and fps, we split the video into frames with the number of frames given by fps. Each frame is stored in the JPEG format in a set k.
3. All the frames are then passed into a function that inspects each consecutive frame and calculates the scene change between them. If the scene change (output as a ratio) is larger than the threshold parameter, it is classified as a high-scene change frame, otherwise it is a low-scene change frame and is added to another set k'.
4. For all the frames in k and not in k' (high-scene changing frames), we encode them using the LT-code implementation described in section 2, while the remaining frames (in k') are transmitted with encoding (low-scene changing frames).
5. Once all frames are decoded using the LT-decoding algorithm, we merge the two sets of frames (with some frames that might have been dropped in transmission) using the serial numbers (and renumber if needed) and then assemble the frames to form the output video.

5.4 Implementation using Video Analysis tool: FFMPEG

For implementing our algorithm, we make use of the open-source command line tool FFMPEG, which consists of various low-level libraries that efficiently handle audio, video, and other digital multimedia files Wikipedia contributors (2024b). Based on the command-line arguments and parameters passed to it, the tool uses overloaded functions to handle different formats of a file (MP4 in our case). Below is a diagram that illustrates the flow of splitting a video into frames.
Note that we do not exactly follow this pipeline, as we require the raw video frames for which FFMPEG does not provide any interface to view and edit. Instead we allow FFMPEG to split into frames according to a user-defined standard and arguments, and then utilize the output as frames for defining and encoding.

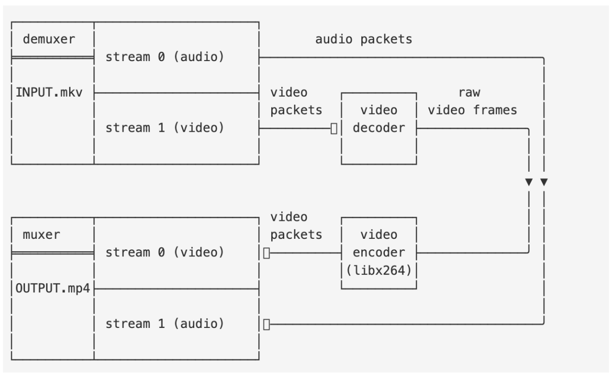


Figure 4: End-to-end flow of Transcoding using FFMPEG FFmpeg Team (2024a)

In our specific implementation, we make use of the "`fps`" argument to specify the number of frames the video has to be broken down into (per second) and the "`select=lt(scene,threshold)`" argument to specify the threshold (lt stands for less than). Using these arguments, the video is broken down into multiple frames (represented as JPEG images) and a subset is extracted as low-scene change frames.

Our implementation is available in the following repository
`https://github.com/Dhyey-Thummar/FEC-code-Bench`

5.5 Metrics

We test on the following parameters:-

1. **Frames-per-second (FPS)**: Range defined by the video, and determines how many frames to divide the video per second.
2. **Scene-change threshold(x)**: Frames with higher than x% change from previous scenes are defined as high-scene change frames and lower than x% are defined as low-scene change frames.
3. **Channel Type**: There are 3 main types of channels that we look at (a) *Perfect channel* (no dropping), (b) *BEC channel* with 50% drop probability, and (c) *Rayleigh Fading Channel* that has the highest noise (close to Shannon limit), leading to most number of frames being dropped *Aff3ct Toolbox* (n.d.).

For each of these parameters, we test **efficiency** (time taken in seconds) and the **quality of service (QoS)**. It is important to note that due to the lack of a formal definition on how to compare QoS, we manually inspect (eye-test) the received video through the BEC channel for jitter and duration of video and quantitatively measure the number of frames received. All other base setup parameters are identical to the LT benchmarking simulation.

Please also note that the subsequent readings are made against a **single** video source. The choice of video was due to the following reasons: -

- The video chosen has sufficient scene changes (in a rapid fashion (per unit time)), leading to a large subset of high-scene changing frames, based on the threshold values between 0.002 to 0.01.
- The duration is 8 seconds, optimal for the time taken to benchmark with multiple FPS, threshold, and channel distributions.
- Viewing the video with frames dropped versus no frames dropped provides a recognizable difference, allowing for judging QoS by looking at the decoded video.

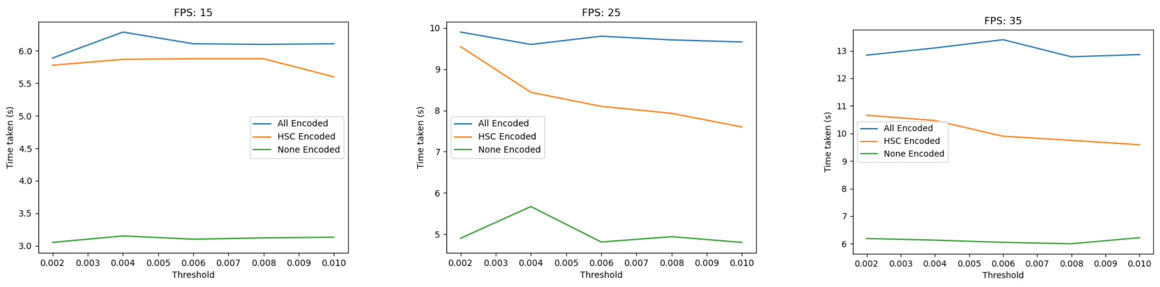
5.6 Results

To begin, we first look at the perfect channel, and the **end-to-end** time taken for 3 different categories: -

1. **Encode All frames**: All frames (high-scene changing and low-scene changing) are encoded using our LT-Code implementation, hence leading to a 100% recovery rate for the frames.

2. **Encode High-Scene Changing (HSC) frames**: Our algorithm, that only encodes high-scene changing frames.

3. **Encode No frames**: No frames are encoded using LT-Code, leading to best case time-complexity.



(a) FPS 15 (b) FPS 25 (c) FPS 35

Figure 5: Time taken for Transmission in Channel 0 (no dropping, perfect channel)

As shown in **Figure 5**, in a perfect setting with a channel that can transmit every single frame without any drops, encoding all frames with LT-codes results in approximately 2x seconds in comparison to not encoding any frames. This is consistent over any threshold value as well as FPS. It is also important to note that with growing FPS, the number of frames significantly increase leading to higher end-to-end time taken for transmission (more frames need to be transmitted and received serially). However, the ratio between encoding all frames and encoding no frames remains approximately the same (≈ 2).
Coming to our proposed idea, the time taken is still higher than what we expect when no frames are encoded, however is asymptotically bound by encoding all frames. Furthermore, with increase in threshold (within a constant FPS), the time taken reduces (non-linearly) due to a decrease in high-scene changing frames (HSC frames).
Please note that there is no point of tracing the number of frames received and comparing them for channel 0 as no matter how many we encode, the number of received frames will always be 100%.

In **Figure 6**, the simulation shows the time taken for end-to-end transmission for all three cases and FPS 15, as well as the percentage of received frames (100 - percentage of frames dropped). As expected, the time taken for encoding all frames is ~ 7 seconds in Channel 2 (Binary Erasure Channel with drop probability 50% and σ as 1) and ~ 6.5 in Channel 4 (Rayleigh Fading Channel). With none encoded, the times are significantly less, with values ≤ 2s with varying threshold values. However, encoding all frames results in 100% of frames received while encoding no frames results in only $\sim 50\%$ in Channel 2 and $\sim 30\%$ in Channel 4.
Our proposed idea manages to consider this trade-off by only encoding HSC frames. This results in the end-to-end time taken to be ~ 6s (lower than encoding all frames) but managing to recover $\sim 95\%$ of the frames, significantly higher than when no frames are encoded. Furthermore, the increase in threshold leads to a lower proportion of HSC frames (the increase in threshold implies we consider a larger change to be a high-scene changing frame), leading to lower time complexity but also lesser subset of frames received.
It is important to consider the implications of a lower number of received frames here. While we approximate the number of received frames to the quality of experience (QoS), this is still an overapproximation. In fact, we only allow the dropping of low-scene changing frames in our algorithm, leading to QoS being much better than dropping the same number of frames randomly (which might include high-scene changing frames). Our calculations suggest that on average, the ratio of HSC to total frames tend to lie in the range 0.6 - 0.9 (for optimal time efficiency in relation to current standards). We will delve deeper into this shortly, but with an antagonistic channel capable of dropping 50% of frames, would lead to 5% to 20% of LSC frames dropped. This correlates to a small perceivable change in the quality of video (subject to manual inspection of the different videos).

Figure 7 and **Figure 8** further give evidence of robustness as well as consistency for our algorithm in comparison to not encoding any frames and encoding all frames.

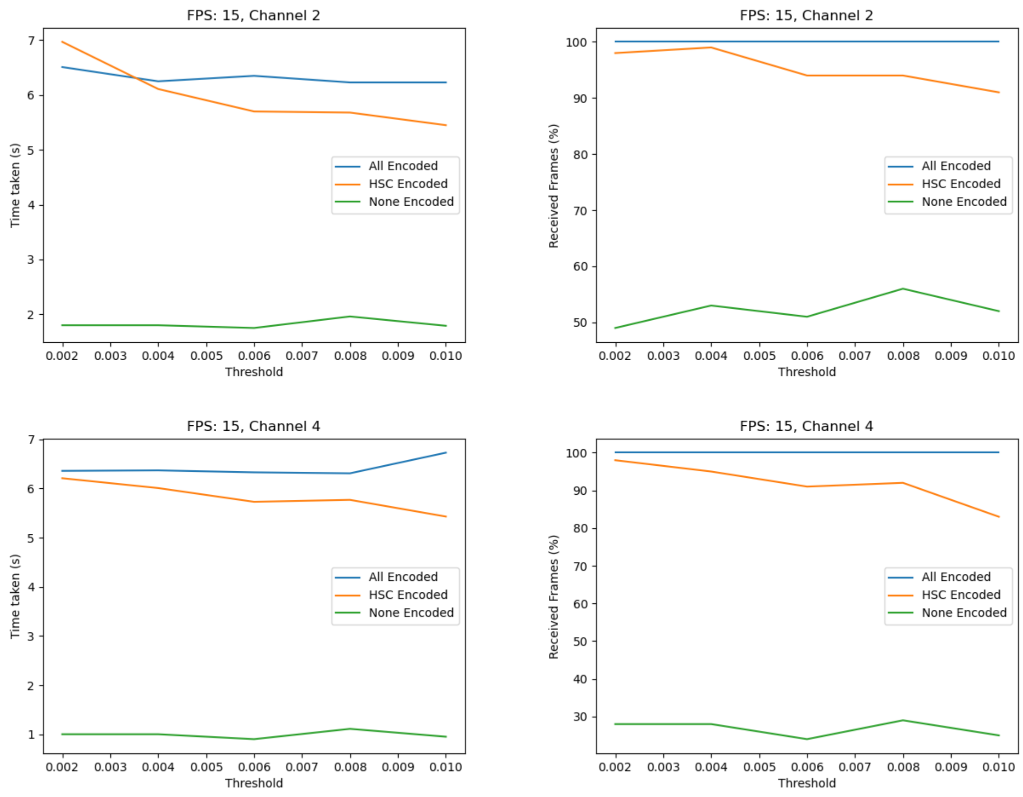


Figure 6: Time taken and Number of frames received with FPS 15

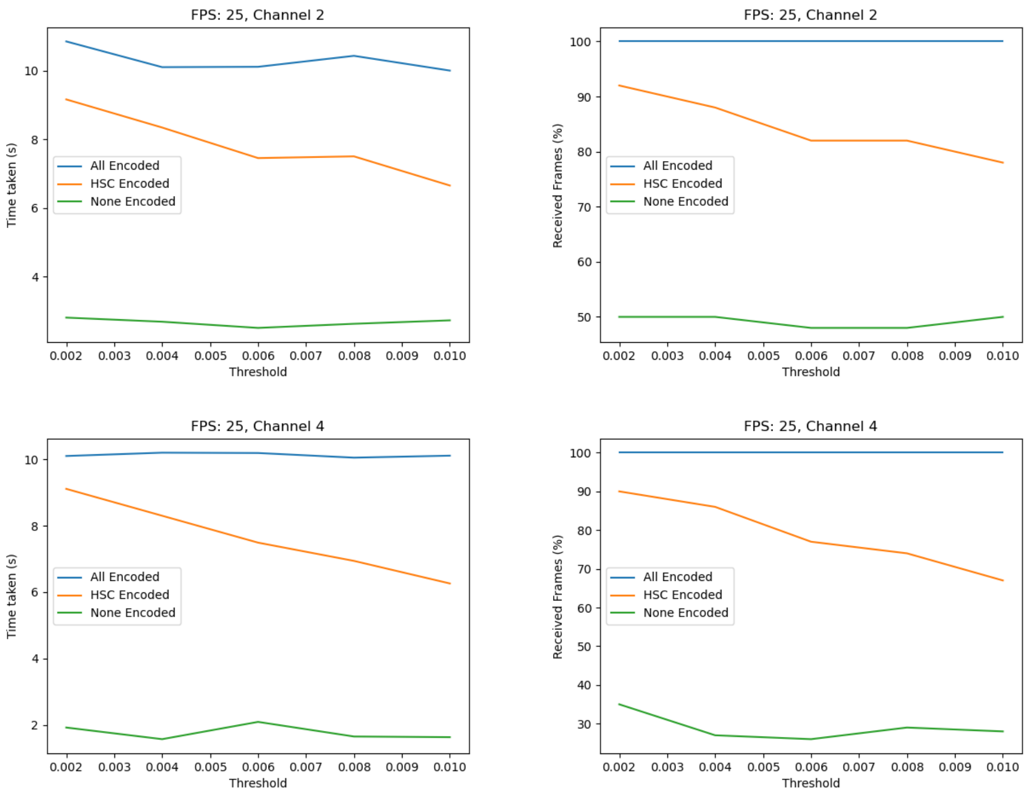


Figure 7: Time taken and Number of frames received with FPS 25

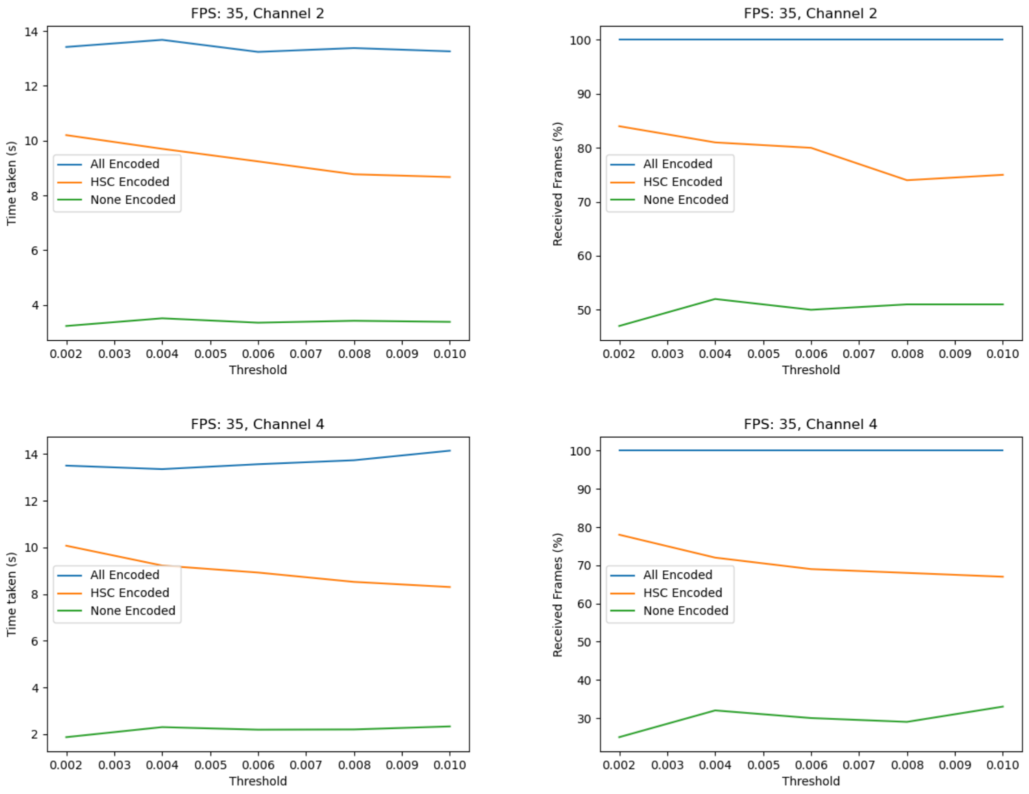


Figure 8: Time taken and Number of frames received with FPS 35

The output videos for Channels 2 and 4 for all FPS values are uploaded in the Dropbox link for reference.

5.6.1 FPS-Threshold Ratio

The previous results reveal a pattern for the relationship between the FPS parameter and the Threshold parameter that can be further studied. Specifically, we notice the following HSC ratio for different frames and threshold values.
HSC ratio: Defined as the number of High-scene changing frames over the total number of frames in the video.

FPS	Threshold	HSC Ratio
15	0.002	0.967
	0.004	0.935
	0.006	0.895
	0.008	0.88
	0.01	0.82
25	0.002	0.879
	0.004	0.758
	0.006	0.676
	0.008	0.603
	0.01	0.545
35	0.002	0.665
	0.004	0.617
	0.006	0.575
	0.008	0.541
	0.01	0.517

Table 1: HSC Ratio for different FPS and Thresholds

Taking F to be the FPS and t to be the threshold, F is inversely related to t (but not proportionate as the relationship is non-linear).
In other words, to maintain a constant HSC ratio, an increase in F implies a decrease in t and vice versa. To put into perspective the current standards, we note the transmission speeds of current high-definition videos ($\sim$ 25 fps) to be 5-8 Mbps Lightstream Team (2024). With the benchmarked video (5 MB), this would lead to around 8 seconds of transmission time to match the standard. With $F = 25$ and $t = 0.004$, we achieve the HSC Ratio (h) to be 0.758, and LSC drop (%) to be approximately 12.5%. **Figure 9** shows the relationship between FPS and threshold for this constant HSC ratio.

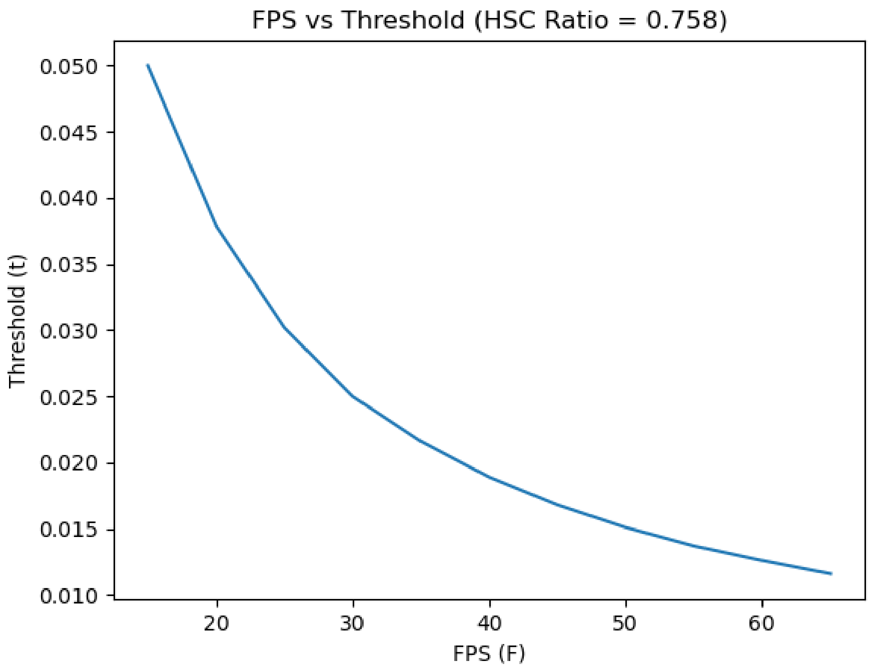


Figure 9: FPS v Threshold for HSC Ratio = 0.758

As expected, we see a hyperbola (representing an approximated inverse function), leading to the following conclusion regarding the FPS-Threshold ratio: With a given FPS F, and a required HSC ratio H, we calculate the threshold by an approximated inverse function $t = \frac{H}{F^k}$ where k is some constant. Please note that this is not an inversely proportional function ($t \not\propto \frac{1}{F^k}$ as the scene changes are inconsistent in a video, and cannot have a closed-form function. The above experiments only provide an example, and the values are experimentally obtained. Furthermore, every video would have a large difference from each other, making generalization about this ratio impossible.

6 Conclusion and Future Work

We provide a thorough analysis on erasure correcting digital fountain codes that allow for reliable and efficient transmission (over other competing standards such as retransmissions in TCP). We specifically look at LT codes and Raptor codes, and through our simulation that involves creating various antagonistic noisy channels (including the Binary Erasure Channel, channel with Additive White Gaussian Noise, and Rayleigh Fading Channel), and modifying existing implementations.

We also propose a new idea to use LT codes for transmitting H264 codec encoded videos using scene-changes between frames. We benchmark it using FPS, different thresholds and noisy channels, and compare the efficiency and drop rate against the standards of encoding all frames and not encoding any frames.

In future work, we aim to extend this analysis to include additional error correction codes, broadening the scope of comparison and performance evaluation. Furthermore, we plan to introduce and test newer implementations, such as sliding window LT codes, to explore their potential benefits in dynamic network environments. Another avenue for improvement involves incorporating the I, P, and B frames concept commonly used in video encoding. This would provide a more structured and efficient approach compared to our current focus on low-scene and high-scene changes. These extensions aim to further enhance the practical applicability and performance of fountain codes in diverse transmission scenarios.

References

Aff3ct toolbox. (n.d.). Retrieved from `https://aff3ct.readthedocs.io/en/latest/index.html`

Anrosent. (n.d.). *Anrosent/lt-code: Encoder/decoder for the luby transform fountain code.* Retrieved from `https://github.com/anrosent/LT-code`

Fernandez, A. (2020). *A quick dive into mp4.* Retrieved from `https://dev.to/alfg/a-quick-dive-into-mp4-57fo` ([Online; accessed 1-December-2024])

FFmpeg Team. (2024a). *Ffmpeg.* Retrieved from `https://www.ffmpeg.org/` ([Online; accessed 11-December-2024])

FFmpeg Team. (2024b). *Libavcodec - ffmpeg.* Retrieved from `https://www.ffmpeg.org/libavcodec.html` ([Online; accessed 9-November-2024])

Google. (n.d.). *Google/gofountain.* Retrieved from `https://github.com/google/gofountain`

Lightstream Team. (2024). *What is video bitrate?* Retrieved from `https://golightstream.com/what-is-video-bitrate/` ([Online; accessed 8-December-2024])

Luby, M. (2002, 01). Lt codes. In (p. 271-). DOI: 10.1109/SFCS.2002.1181950

Minder, L., Shokrollahi, A., Watson, M., Luby, M., & Stockhammer, T. (2011, August). *RaptorQ Forward Error Correction Scheme for Object Delivery* (No. 6330). RFC 6330. RFC Editor. Retrieved from `https://www.rfc-editor.org/info/rfc6330` DOI: 10.17487/RFC6330

OTTVerse Team. (2022). *I, p, b frames, idr keyframes: Differences and use cases.* Retrieved from `https://ottverse.com/i-p-b-frames-idr-keyframes-differences-usecases/` ([Online; accessed 1-December-2024])

Shokrollahi, A., Stockhammer, T., Luby, M., & Watson, M. (2007, October). *Raptor Forward Error Correction Scheme for Object Delivery* (No. 5053). RFC 5053. RFC Editor. Retrieved from `https://www.rfc-editor.org/info/rfc5053` DOI: 10.17487/RFC5053

Silas, S. (2021, July). Real-time oblivious erasure correction with linear time decoding and constant feedback. In *2021 ieee international symposium on information theory (isit)* (p. 1361-1366). DOI: 10.1109/ISIT45174.2021.9518168

Soliton distribution. (n.d.). Retrieved from `https://en.wikipedia.org/wiki/Soliton_distribution`

Tirronen, T. (n.d.). Optimizing the degree distribution of lt codes.

Wikipedia contributors. (2024a). *Advanced Video Coding – Wikipedia, The Free Encyclopedia.* Retrieved from `https://en.wikipedia.org/wiki/Advanced_Video_Coding` ([Online; accessed 1-December-2024])

Wikipedia contributors. (2024b). *FFmpeg – Wikipedia, The Free Encyclopedia.* Retrieved from `https://en.wikipedia.org/wiki/FFmpeg` ([Online; accessed 8-December-2024])

Wu, S., Guan, Q., & Cui, C. (2021). Research on the optimal decoding overhead of lt codes. In *2021 international wireless communications and mobile computing (iwcmc)* (p. 1829-1834). DOI: 10.1109/IWCMC51323.2021.9498769