Reliable, Efficient Large-File Delivery over Lossy, Unidirectional Links

William L. Van Besien
IOMAXIS, LLC.
10700 Ballantraye Drive Suite 106
Fredericksburg, VA 22407
wvanbesien@iomaxis.com

Benjamin Ferris
IOMAXIS, LLC.
10700 Ballantraye Drive Suite 106
Fredericksburg, VA 22407
bferris@iomaxis.com

Jim Dudish
IOMAXIS, LLC.
10700 Ballantraye Drive Suite 106
Fredericksburg, VA 22407
jdudish@iomaxis.com

***Abstract*—Many mission-specific network architectures inhibit or delay bidirectional connectivity among hosts. Such constraints are common in Delay-Tolerant Networks, cyber-physical systems, and secured networks, where crucial network protocols – principally the Transmission Control Protocol (TCP) – cannot be used. In this paper, we present a process based upon Low-Density Parity Check (LDPC) codes for efficiently and reliably bridging TCP connections across network segments whose architecture enforces one-way data flows and prevents Automatic Repeat Request (ARQ) prompts. We demonstrate its performance, measured by the time required to complete a file transaction, as comparable to *scp* (Secure Copy), a common file transfer utility.**

TABLE OF CONTENTS

1. INTRODUCTION

Using Automatic Repeat Requests (ARQ) is a common technique in network protocols to achieve reliable communication over an unreliable data link. In protocols built upon ARQs, peers exchange acknowledgements and negative-acknowledgement (ACKs and NAKs) to repeatedly prompt for retransmission of missing data segments until they successfully complete the transaction [16]. The Transmission Control Protocol (TCP) [8] is perhaps the most well-known protocol using ARQ, though is it far from alone: Quick UDP Internet Connections (more popularly known as QUIC), Licklider Transmission Protocol (LTP) [7] and others [18] largely rely on the technique as well.

However, using ARQ to facilitate bulk data transfer is inefficient or infeasible in certain networking environments. This is particularly true in links characterized by extreme and/or asymmetric latency between hosts, and situations where a return link does not exist to the transmitting host [15]. When round-trip latency can be measured in minutes or hours, as is common in Deep-Space communications, the field of Delay Tolerant Networking has developed to address these and other challenges [15]. However, when a return link is entirely absent, as is the case for certain specialized network environments, we must develop novel techniques to provide reliability without ARQ. This challenge becomes particularly acute when message sizes become large and the effects of interference, congestion, and data buffering cause unpredictable patterns of data loss.

The study of encodings to recover unpredictable communication errors and loss is central to Information Theory, and of Coding Theory in particular. Turbo Codes, Reed-Solomon Codes, Low-Density Parity Check (LDPC) codes, and others provide efficient and powerful mechanisms for fixing bit-errors and erasures. These schemes are applied ubiquitously: from Interplanetary telecoms to 5G cellular networks, Forward Error Correction (FEC) encodings substantially enhance the likelihood that a receiver correctly retrieves a given data block. The LDPC [1] coding scheme is the backbone of the protocol we present in this paper, which detailed in Section 5.

Much of the existing literature regarding the use of error- and erasure-codes, however, is concerned with radio communication and errors at the physical and link-layers of the network stack [4, 5, 12]. This focus on the low-levels of the network stack leaves higher-level concerns such as transmission control and message integrity to higher-level protocols. This contrasts with the intended use-case we present where loss occurs at the level of network and transport-layer packets, which are either received correctly in their entirety, or dropped by the network. Each packet contains thousands of bytes of payload, and so the minimum length of missing data we should expect is on the order of tens of thousands of consecutive bits. Furthermore, we must be resilient to a burst of consecutively dropped packets that may amount to millions of consecutive bits missing inside a

978-1-7281-7436-5/21/$31.00 ©2021 IEEE

2021 IEEE Aerospace Conference | 978-1-7281-7436-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/AERO50100.2021.9438494

single transaction. This presents a somewhat unusual use-case for error correcting and erasure codes. While some transport and application-layer protocols do leverage erasure codes and forward error correction, it is normally used in conjunction with ARQ [19] or used for streaming multimedia data where some loss of content is acceptable [21].

The principal question we seek to answer in this paper is as follows: can we create a protocol that throttles transmissions without adaptive feedback, provides TCP-like reliability over a lossy channel without ARQ, and maintains high link utilization and throughput? Sections 5 and 6 detail how to structure and enable such a process, Section 7 dicusses its performance in operational settings, and finally Sections 8 and 9 detail remaining development activities.

2. OBJECTIVES

The protocol defines an interaction between a data producer and consumer situated on either end of a unidirectional data path in which traffic can only flow from the producer to consumer, as illustrated in **Figure 1**. Its objective is to bridge a TCP stream carrying an arbitrary, binary file, across this network segment that would otherwise inhibit TCP (or similar ARQ-based protocols) from opening a connection to the consumer.

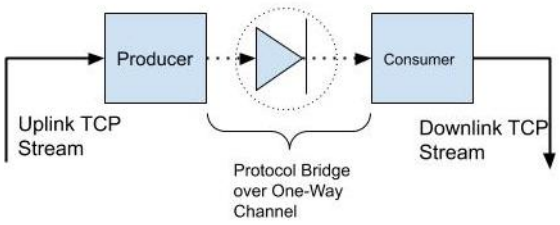


Figure 1 - Notional Archicture

Specifically, it must facilitate the transfer of the incoming TCP data stream in order, and with complete integrity; each uplink stream (uplinked file) must precisely match the corresponding downlink stream. Moreover, it must assume the route between the producer and consumer is subjected to rate throttling, congestion, interference, and other effects that may cause intermittent data loss and out-of-order delivery.

A final important constraint is that this protocol must seek to maximize link throughput [16], as measured by the proportion of the size of the data file to the volume of data required to successfully provide its reliable delivery. I.e., it must seek to minimize the protocol overhead (consisting of headers, checksums, and error coding blocks), L_{ovr}, in order to maximize the following expression for throughput:

$$Throughput = \frac{L_{file}}{L_{file} + L_{ovr}}$$

Equivalently, throughput is the measure of how efficiently the protocol uses the link. Reduced throughput causes transactions to take longer to traverse from producer to consumer, and generally adversely affects system performance and stability.

In this paper, we present the following strategy to provide reliable delivery with high throughput. First, we seek to understand patterns of packet loss in various networking environments to discover the most important drivers of packet loss. Second, we identify and apply suitable erasure coding schemes that best suit the large anticipated file sizes and loss profile. Finally, having identified several parameters governing the behavior of the protocol, we show how to optimize and balance them to achieve high throughput, minimal end-to-end transaction latency, and provide strong reliability in certain networking environments.

3. DRIVERS OF PACKET LOSS

Measuring data loss in various networking environments under differing conditions is the first step in our analysis. In particular, in this section we seek to describe the frequency and distribution of packet loss, particularly to see if loss is dominated by scattered, errant dropped packets, or if packets were more likely to be dropped in bursts. To measure and assess this loss profile, we developed a pair of benchmarking utilities written in C to maximize runtime performance. The first was a client to generate a sequence of *n* packets, each containing its index in the sequence followed by a payload of 1,000 randomly-generated bytes. The client transmitted these packets as quickly as possible. The second utility was a server which consumed the stream of packets and identified gaps in indexing, indicating dropped packets. The client had a configurable option we designate the throttling rate, or the pause we introduce between consecutive packet transmissions. Increasing the throttling rate causes the overall transmission time to rise, as the producer spends more time paused between transmitting UDP packets.

We initially ran the benchmarking utility in the following network environment scenarios:

1. Workstation connected to a server through a Virtual Private Network (VPN);
2. Between two workstations in a wireless LAN (Local Area Network);
3. Between two servers hosted in a private cloud running on separate hypervisors with heavy network utilization by other cloud hosts.
4. Between two servers hosted in a private cloud running on separate hypervisors absent of any other network traffic;

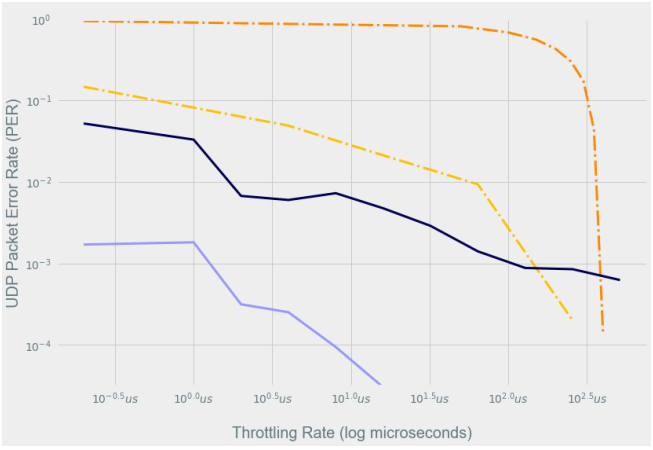


Figure 2 - Packet loss as function of throttle rate for scenarios A, B, C and D.

The results show that transmission rate is an important driver of packet loss, and this result held across each of the networking environments sampled. **Figure 2** relates **Packet Error Rate (PER)**, the proportion of missing packets, to the throttling rate (shown along the horizontal axis), both in logarithmic scale.

Under most circumstances, increasing the throttling (and thus slowing down the transmission) resulted in substantial reduction in packet loss, even in situations where the unthrottled packet loss rate was as high as 94%. In general, slowing the transmission speed by increasing the throttling rate suppressed the rate of packet loss in all environments. We summarize each of the environments represented in **Figure 2** as follows:

Scenario A: Workstation connected to the internet over a VPN to a cloud server (dashed orange line). The sharp elbow of the curve indicates a clear rate-limiting functionality somewhere along the route, such as by the Internet Service Provider (ISP) or VPN. This rate limiting behavior inhibits almost all traffic until the throttling rate brings the transmission rate below the threshold. This threshold occurs when the intra-packet delay is approximately 250µs ($10^{2.4}\mu s$ in **Figure 2**), corresponding to a maximum bitrate of 2 MByte/sec. After data transmission rate falls below this rate limit, the overall PER vanishes due to the large pauses between packet transmissions.

Scenario B: Between two workstations on a wireless Local Area Network (dashed yellow line). Major reductions of the transmission rate produce continuous but marginal improvements in link quality. This becomes particularly pronounced when the throttling rate hits approximately $10^{2.1}$ µs, or about 125 µs, corresponding to about 3 MB/s. However, even major reductions in the transmission rate does not eliminate the likelihood of dropped packets.

Scenario C: Between two servers hosted in a private cloud running on separate hypervisors with heavy network utilization by other cloud hosts (dark blue line). At nearly all throttling rates, observed packet loss was at least an order of magnitude higher than in an uncongested setting. Reducing the transmission speed produced modest, but diminishing returns improving link quality. Even at extremely low data rates, a PER of about 10^{-4} (about 0.01%) was present, which guarantees that data streams larger than several megabytes will encounter some level of loss.

Scenario D: Between two servers hosted in a private cloud running on separate hypervisors, absent of any other network traffic (light blue line). With no other network traffic it was rare to observe packet loss at any data rate. Loss was only observed for very large sample sizes such as 10 GB or greater. It is possible even the small observed loss is a function of a backed-up buffer on the receiver, rather than loss introduced by the network. When the per-packet throttling rate exceeded 10 µs per packet, it became impossible to observe any packet loss whatsoever.

Consequences for Lower Data Rates

Simply increasing the throttling rate (thereby slowing the rate of packet transmission) appears to be a simple means to suppress packet loss, but causes substantial increases in the length of time needed to complete a file transaction. In our benchmarks below, for example, introducing a 200µs (0.0002 seconds) gap between packet transmission caused a *five-fold increase* in the overall transaction duration. This corresponds to transmitting at less than 20 % of the rate of a corresponding unthrottled transmission.

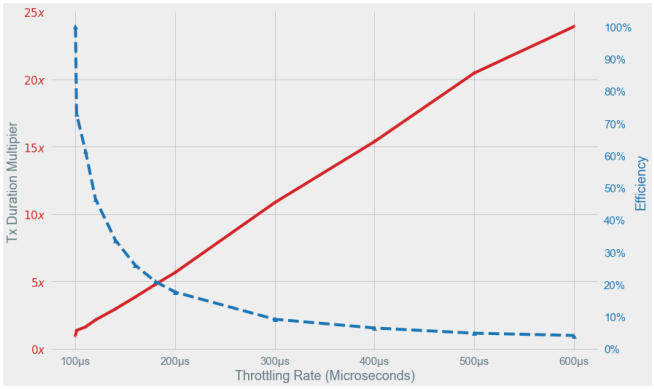


Figure 3 - Performance consequences of rate throttling

Figure 3 relates the throttling rate to the overall transaction duration to a benchmark unthrottled transaction. Increasing the throttling rate to more than a few microseconds violates our system's performance requirements and introduce further design complications.

4. Characterizing Packet Loss

In section 3, we related the throttling rate to overall packet loss under four different network environment scenarios. It was observed that introducing brief pauses of a few microseconds between packet transmission can improve link quality, but continued application produced diminishing returns and caused unacceptable transaction latency. In this section, we characterize the pattern of packet loss, with

particular focus observing whether these losses appear randomly scattered, or in bursts (i.e., clustered together). Understanding the distribution of packet loss points us toward the optimal methods for enabling reconstruction and repair of dropped data segments.

Throughout the rest of this paper, we focus on the third scenario detailed in section 3: large file transfer between two virtual machines in a VPC with high network usage. In this environment, we recorded the packet loss for a 90 MB transaction, consisting of 90,000 UDP packets, each with 1,000 bytes of file data. 90 MB was selected as it is a perfect square that fits into a small image. **Figure 4** visualizes the data with packet loss. Each UDP packet represented by a pixel in a 300 x 300 image. Dark blue pixels represent packets that were received while mis-colored pixels represent dropped packets. Measurements were taken with 0, 10, 20, and 40 µs delays per-packet, respectively.

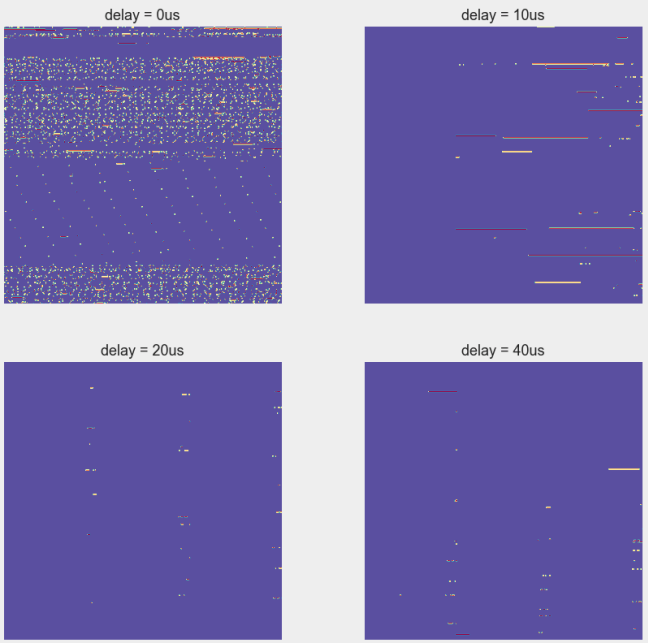


Figure 4 – Visualization of packet loss at various throttling rates

The results show that introducing a modest 10 µs per-packet pause cut packet loss approximately in half. The results further show that the loss consists of both significant bursts as well as very frequent isolated dropped packets. Therefore, introducing a brief pause reduces, but does not eliminate, the presence of these isolated packet losses. Furthermore, as observed under these networking conditions, we should expect to observe error bursts of approximately 10-100 KB of consecutive file data.

Since simply reducing the transmission speed by increasing the throttling rate cannot eliminate the likelihood of errors, and simply duplicating data causes the link throughput to rapidly approach zero[1], the following considerations will be important as we move toward the application of techniques for repairing these gaps. The smallest segment of missing data is approximately 1,000-1500 bytes. This corresponds to the loss of a single UDP packet. The largest segment we should anticipate is on the order of 100 UDP packets, or 100,000-150,000 bytes.

While this visualization pertains to a single transaction, each transaction will have a slightly different loss profile, as measured by the overall packet drop rate and its distribution. However, at the level of 90 MB blocks and larger, these differences become homogenized; while the average loss rate per block remains the same regardless of block size, its variance decreases as block sizes become larger.

5. Applying Forward Error Correction

Having described patterns of packet loss in the sampled networking environments, in this section we identify optimal methods for reconstructing dropped packets. As repeating a transmission multiple times rapidly causes link throughput to approach zero, as well as creating unnecessary backlog on the consumer, we must look toward Forward Error Correction (FEC) schemes to provide resiliency for less overhead. Much of the literature regarding FEC, and its sub-discipline Erasure Coding (EC), considers errors (ie., flipped bits in received data) or erasures (i.e., missing data) at the bit-level. However, our challenge presents a situation where *smallest* erasure will be 8,000 to 12,000 bits long, corresponding to the payload of a UDP packet inside an Ethernet Maximum Transmission Unit (MTU) frame. Moreover, we expect typical blocks lengths to be up to several hundred megabytes long, rendering many common FEC and EC codes impossible or computationally impractical to apply. Finally, these constraints, in addition to expected file sizes up to 10 GB, make it imperative to select coding schemes relying on simple arithmetic operations.

We selected the Low-Density Parity Check codes (LDPC) as the starting point for providing reliability. First presented by Gallager [1] in 1960, LDPC is a linear, block error correction code, in which every error coding block is a linear combination of data blocks. The construction of parity check blocks is easily defined by a sparse parity check matrix, and details for their construction may be found in [1, 2, 3, 4].

We considered other encodings to provide reconstruction of missing data, including Reed-Solomon (RS) [20] and Turbo Codes [5]. However, they are better solutions for lower code rates in which higher densities of errors are expected, and Andrews *et al* from JPL noted that "LDPC codes have performance and complexity advantages over Turbo Codes at high code rates" [5]. From our observations in the prior section, we choose to throttle our transmission to target an

[1] Duplicating data once will cause the throughput on the link to be cut in half, duplicating three times cuts throughput by two thirds, and so on.

erasure rate well below 10%, which allows us to run at a higher code rate with less overhead.

LDPC codes provided the starting point for our erasure coding scheme, but our implementation diverged in several key respects from textbook specifications. We opted to use a novel, non-standard variant of LDPC for two main reasons. First, our changes yielded a very compact software implementation: our prototype, written in Python, consisted of fewer than two-dozen lines each for the encode and decode functions, respectively. Second, the changes permitted us to optimize for the only arithmetic operation used: vectored *exclusive-or (xor)* operations over kilobyte-length binary blocks. The primary change from LDPC is removing explicit matrix operations $x = mG$ (with G being the generator matrix) in favor of selecting random subsets of size S of payload blocks over which simple *xor* parity checks are computed. Blocks are embedded inside PDUs that contain information such as their offset within the overall code block.

Our coding scheme is driven by two primary parameters: Symbols-per-parity-check, S, which must be a factor of the length of the sequence to be encoded; and the number of "layers", L. Together, one may compute the code rate, $r = (1 + {}^{L}/_{S})^{-1}$. The block size is the sum of the number of data and parity blocks, which is $N = {}^{K}/_{r}$ blocks.

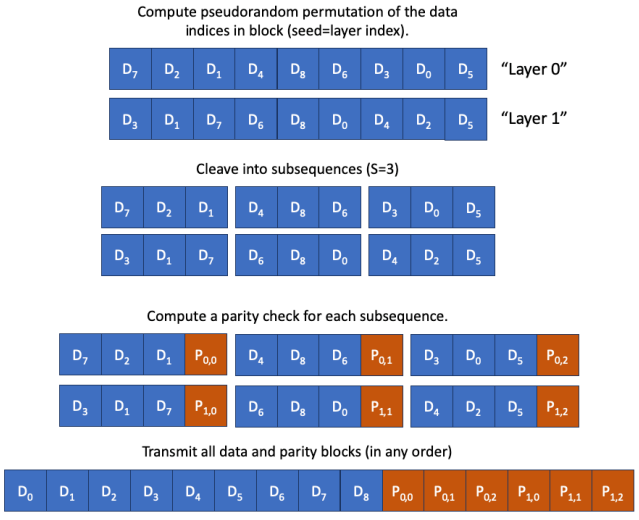


Figure 5 – Encoding example with *L*=2 and *S*=3

The encoding function accepts a sequence of fixed-sized data blocks, and produces a sequence of fixed-size parity check blocks. The sizes of all parity and data blocks have equal bit-width, and correspond to the size of an Ethernet MTU (minus certain protocol overhead). Each of these blocks are prepended with protocol metadata, such as transaction ID, transaction size, offset, and – if a parity block – the layer and index that it corresponds to. **Figure 5** demonstrates an example encoding, and **Figure 6** contains the Python-like Pseudocode for the encoder.

```
def encode(data_blocks, S, L):
    K = len(data_blocks)
    B = N / S
    parity_blocks = array(K*L/S)
    for l in 0..L:
        parity_blocks[B*l..B*(l+1)] = \
            encode_layer(S, data_blocks, l)
    return parity_blocks

def encode_layer(S, data_blocks, layer_index):
    set_random_seed(layer_index)
    perm = random_permutation(0..len(data_blocks))
    B = len(data_blocks) / S
    parity_blocks = array(B) ## P is length B
    for b in 0..B:
        ## vec_xor computes the xor of the
        ## data_blocks at the given indices.
        parity_blocks[b] = \
            vec_xor(data_blocks, perm[b*S..b*(S+1)])
    return parity_blocks
```

Figure 6 - Encoder Python-like Psuedocode

The decode function accepts an ordered sequence of data and parity check blocks, with Null-values representing absent blocks. The decode function then repairs absent data and parity blocks in an iterative fashion, raising an error if the count of missing blocks levels off after multiple repair cycles. Similar to the encoder, the decoder creates a pseudorandom permutation of the indices, using the current layer as the seed for the random number generator (this allows the encoder and decoder to generate identical permutations). It walks over each of B subsequences, each consisting of S data blocks and a parity check block, and if exactly one data block is missing it computes a vectored *xor* of the blocks that are present to reconstruct it. **Figure 7** presents the Python-like pseudocode to implement the decode, with the intricate array indexing written as comments.

```
def decode(data_blocks, p_blocks, S, L, C):
    K = len(data_blocks)
    B = K / S

    ## The decode step is iterative, virtually all
    ## sequences can be reconstructed in at most 3
    ## cycles.
    for cycle in 0..C:
        for l in 0..L
            decode_cycle(data_blocks, p_blocks, S, l)

def decode_cycle(data_blocks, p_blocks, S, l):
    K = len(data_blocks), B = K/S
    set_random_seed(l)
    perm = random_permutation(0..len(data_blocks))

    ## For-each B-length subsequence
    for b in 0..B:
        m = count_missing(data_blocks,
                          perm[B*b..B*(b+1)],
                          p_blocks[l*B+b])
        if m == 1:
            ## If only one block is missing, whether
            ## it's a data or parity block, compute
            ## the xor of all existing blocks and
            ## repair it.
        else:
            ## There are too many gaps, bypass
            ## as it cannot be repaired in this
            ## cycle
```

Figure 7 - Decoder Python-like Pseudocode

Since PDUs encode important metadata about the block, the software accepting these PDUs off of the socket can place the data or parity block in the proper place within the code block before passing it to the decode function.

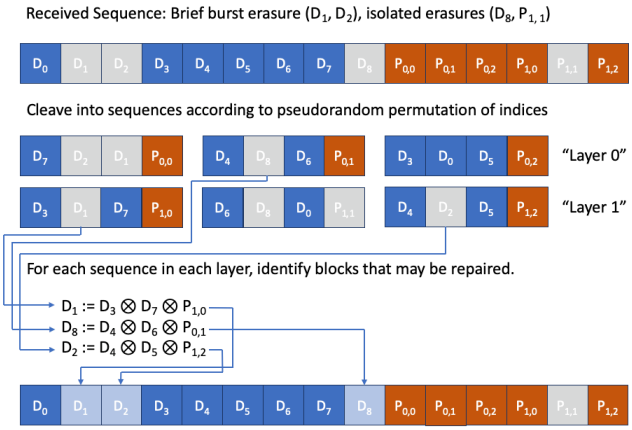


Figure 8 – Decoder Example (*L*=2, *S*=3)

Figure 8 walks through an example decoding procedure, with data length $K = 9$, symbols-per-parity-check $S = 3$, and the number of "layers" $L = 2$. Grey boxes indicate dropped data and parity blocks. For each layer, it generates the proper permutation of data indices to produce $B = {}^{K}/_{S}$ subsequences, each with its associated parity check block. When precisely one of these blocks are absent, the remaining blocks can be summed together with *xor* to solve for the missing block. If two or more symbols from any sequence are absent, simple parity checking will be unable to reconstruct the missing data. However, the use of additional layers provides extra resiliency, as each data symbol has coverage from multiple parity check blocks. Reconstructing missing data segments within a data block is therefore an iterative process – missing symbols are reconstructed with each step, and those reconstructed symbols can then be used to reconstruct more symbols in further iterations.

In the given example, even though the first subsequence in the first layer is missing two data blocks, they can both be repaired in just one repair cycle (i.e., with one invocation of the "decode" function). The only remaining missing block is one of the parity checks, which can be ignored since all the data is present – however that too may be repaired quickly with a second call of decode.

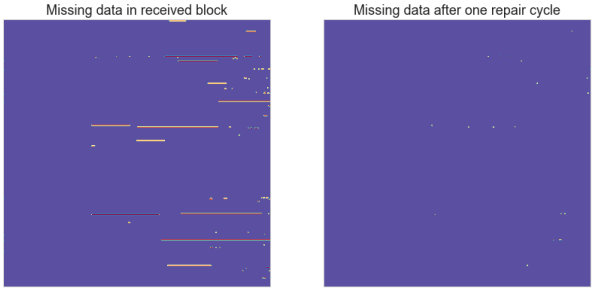


Figure 9 – Visualization of repairs after one cycle

Figure 9 visualizes one step of repairing a 90 MB code block transmitted over a congested network with 2% loss (90,000 packets again rendered in a 300x300 pixel image). Each pixel represents a PDU containing 1,000 bytes of data. After just one cycle of repairs (taking approximately 0.33s on a MacBook workstation), our decoder repaired nearly all missing segments, with all of the few remaining segments repaired in the second iteration.

Since we will transmit at a speed yielding relatively low Packet Error Rates, we aim to operate at the highest code rate possible sufficient to repair any code block whose error density is beneath our threshold. A higher code rate implies higher link throughput as fewer transmissions of PDUs containing parity checks need to be sent. A higher code rate also reduces the burden on the producer to compute parity checks.

6. Optimizing System Parameters

Section 5 discussed the selection and operation of our atypical LDPC encoding scheme, which we chose primarily because of its simplicity of implementation, reliance on the *xor* arithmetic operation, and suitability for large erasures. However, we are presented with a range of parameters governing transmission rates, encode/decode complexity, and throughput. Selection of these parameters must be balanced to provide reliability for a given loss profile, while minimizing encoding operations, network traffic overhead for parity-checks, and overall transaction duration.

More formally, we must select the highest code rate sufficient to repair any code block with an erasure rate below the given threshold. Note that the code rate itself is computed from two parameters: the number of data symbols per parity symbols, and the number of *layers*. **Table 1** details the configurable parameters and the specific values for them that we selected. The parameters in the table are defined as the following:

- **PDU size.** The protocol data unit contained within a single UDP packet, which itself should be contained in a single Ethernet frame.
- **Throttle Rate (Per-Packet transmission delay).** The time paused between transmission of any two PDUs. Note that when this pause is only a few microseconds, inaccuracies begin to arise. As an implementation detail, we batch the pause. For example, if the delay is 2 µs, we transmit ten PDUs, and then pause for 20 µs, where effects of function calls and context switching become less apparent.
- **Block size.** The fundamental unit over which error correction is applied. Though we intend to support files of multiple gigabytes, using block sizes this large may begin to exceed the memory of either the data producer or consumer. Furthermore, block sizes on the order of gigabytes prevent the producer from flushing old data and introduces further design constraints. Conversely, code blocks too small are more likely to be rendered

unrecoverable by a sufficiently large burst of errors. We must therefore select the size of each code block to balance these considerations.

- **Symbols per parity check (*S*).** This indicates to the erasure coding scheme the number of blocks to use for each parity check symbol. For example, if this value is 10, it will create a parity check symbol for every 10 symbols of data. Generally having fewer symbols per parity check is more resilient to loss, but at the cost of substantially higher overhead.

- **Encoding layers (*L*).** Number of times to compute the parity checks for different permutations of symbols. Equivalently, this governs the how many parity checks cover each data symbol.

- **Code rate.** The ratio of the size of the data unit to the size of the overall code block. It is computed according to the following expression: $r = (1 + \frac{L}{S})^{-1}$, where L is the number of layers, and S is the symbols per parity check.

Table 1 – Protocol parameter selections

Parameter	Value
PDU Size	1,000 bytes
Throttle Rate	10 µs
Block Size	100 MB
Symbols per Parity Check	25
LDPC Layers	2
Code Rate	0.925

The motivation for selecting the parameters in **Table 1** for experimentation in our prototypical networking environment, as described in section 3 (large file transfer between two virtual machines in a VPC with high network usage), is as follows. First, the brief 10 µs per-packet pause reduces the baseline erasure rate from $10^{-1.3}$, or about 6.2%, to $10^{-2.1}$, or about 0.7% – an 89% reduction. However, this causes the transmission time to increase by 40%. Second, since we can expect an erasure rate generally below 1%, we can afford to use a relatively high value for the number of symbols per parity check, implying lower overhead. The selected value of 25 means that for every 25 symbols, chosen according to a random permutation, our LDPC-like encoding scheme will produce one parity check. We further chose two layers of this parity checking, which means this process is applied twice, and provides additional resilience against errors. Together, we transmit with a code rate of $(1 + {}^{2}/_{25})^{-1} = {}^{25}/_{27} \cong 0.925$.

We measured that the average loss across a 100 MB code block to be 0.7% (std=0.2%), and this encoding rate is sufficient to repair any code block with loss several standard deviations above the mean.

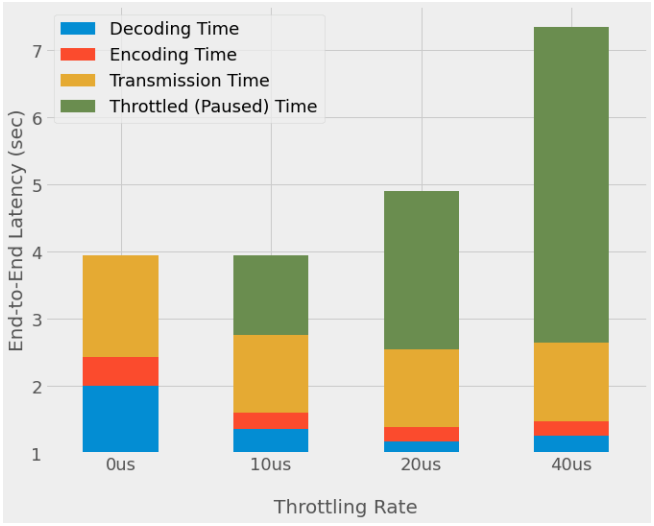


Figure 10 - Transaction Times for various throttling rates

Figure 10 illustrates this end-to-end latency for a 100MB code block in terms of time to transmit, time to encode/decode, and time spent paused between packet transmission. The use of a 10 µs per-packet delay at the expense of a 40% increase in transmission delay may strike one as excessive. However, introducing this brief pause actually decreases end-to-end transaction latency for the following reasons. First, running with a 10 µs pause permits us to run at a higher code rate with less overhead (higher throughput). Second, with lower overhead there are fewer computations required by the producer for parity checks, and likewise on the consumer there are fewer missing packets to reconstruct (less time needed to decode and perform repairs). Together, these produce a slight performance enhancement over using unthrottled transmission. Performance quickly degrades, however, as the throttling rate extends beyond 20 µs.

To summarize, the producer ingests about 92.5 MB of data off of the uplink TCP stream. It expands into a 100 MB code block via application of our LDPC-like encoding scheme, and then transmits as a series of UDP packets. The payload for each contains a PDU with header data and some amount of payload data. It repeats this process until the stream is empty. Methods for trailing data segments less than the block size and for handling small uplink streams is discussed in Section 9.

We identified 100 MB as the optimal block size for the following reason: as block sizes become smaller, variance in erasure rates increases due to smaller sample size and requires additional parity checks to mitigate. Much larger block sizes begin to risk using up available memory and also add excessive latency as the code block cannot be flushed

from memory until every parity check computation is complete and they are all transmitted. This procedure remains vulnerable to decoding failures for any code block for a given uplinked data stream. However, with the given parameterization we did not observe any failures for transaction sizes up to 100GB.

7. Results

Having selected the erasure coding scheme and chosen its parameters to match the expected link conditions, we confirm this provides adequate reliability within acceptable performance boundaries. To benchmark our protocol, we performed 10 GB file transactions over the same networking environment. We measured the time to encode, decode, and transmit each code block, and for the cumulative file transaction.

To provide a basis for comparison, we performed a file transfer using *scp* of the same file between the same two hosts and under identical networking conditions. The results, as illustrated in **Figure 11**, indicated that our protocol generally matches the performance of *scp*. The figure shows the cumulative amount of time spent transmitting, encoding, and decoding for the given 10GB transaction.

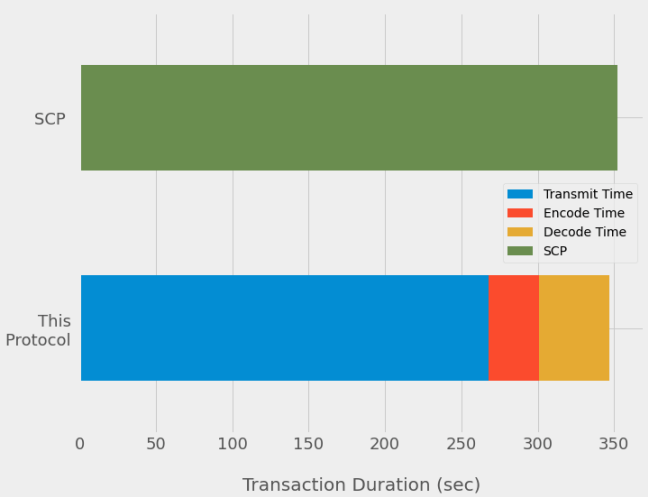


Figure 11 – Time to complete 10GB transaction, compared to *scp*

We also measured the latency introduced by encoding and decoding for each 100 MB block within the overall 10 GB transaction. These measurements, plotted in **Figure 12**, show decoding operations took roughly 33% longer than the corresponding encoding operations: 450 ms average decode time per 100 MB versus 330 ms time to encode per 100 MB. Larger decoding time is not surprising considering packet loss ranging from 0.4% to 1.2% per block. Every code block was able to be repaired within a half of a second. We believe further software engineering optimizations will both reduce this gap and lower the overall latency introduced by the encode/decode operations.

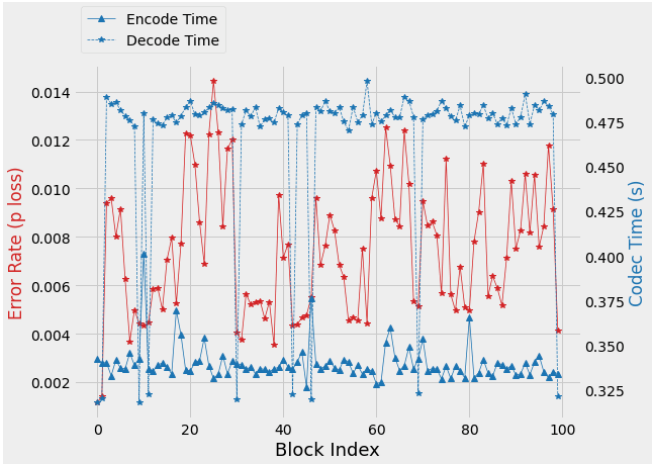


Figure 12 - Encode and decode time per code block

Overall, the reliable one-way transmission of the example 10 GB file took about 345 seconds and incurred an overhead of approximately 260 MB due to protocol headers, and an additional 860 MB of error correction PDUs. This brought our overall throughput to 90%, an acceptable outcome. Furthermore, although the 10 µs per-packet throttling rate does increase the transmission delay, the costs of encoding and decoding operations become excessive without its use. Finally, as its performance appears to be on-par with *scp*, we take this as indication that its performance approaches the maximum possible under the given networking conditions.

8. Security Considerations

Authentication, integrity and confidentiality are important parts of any networking protocol, and continues to be an important part of ongoing work. In a brief, yet informal security analysis, we consider the principal threat to be the injection of malicious PDUs into the data stream. Such malicious PDUs could overwrite portions of file data with compromised material, or cause an incorrect reconstruction of a missing segment. Such an attack is similar to DNS cache poisoning attacks [14].

We believe a common keyed-hash message authentication code (HMAC) per PDU sufficient to provide integrity and authentication over the entire file. The use of HMAC with sufficient password strength and authentication size substantially mitigates the risk of a malicious or errant PDU embedding data within the file on the consumer-side. Moreover, the use of LDPC erasure coding provides further integrity checking across the entire transaction, as the sum of each parity check block must sum to zero. We continue to investigate new extensions to this protocol, such as appending checksums or hashes as the final data symbol (over which parity checking is computed) in a transaction.

These properties can lead to a final security mitigation: in the event that a parity check computation fails (i.e., parity blocks do not add up to zero using *xor*) or HMAC validation begins to fail with high frequency, the consumer may flush the

transaction from memory, raise an alarm, and even shut itself down until system administrators isolate the source of the attack. While such behavior would otherwise be an invitation for a Denial of Service attack for a server on the internet, this protocol is initially intended for isolated networking environments.

The use of HMAC introduces new configuration items for the producer and consumer applications: a password from which a symmetric cryptographic key will be derived, the size of the HMAC authentic code in the header, and a flag to indicate whether to self-terminate if an attack is detected. System administrators may adjust this size to meet the security requirements; a shorter authentication code will make it easier for an adversary to mount an attack similar to a DNS cache poisoning attack, while longer ones mitigate that risk at the cost of higher header overhead and corresponding reduction in throughput.

PDU payload confidentiality is out of scope, and protecting file confidentiality is not a goal of these security extensions; encrypting the file before beginning the transaction would be advised in such cases.

9. Further Considerations

Our primary aim in this paper is to propose and demonstrate the use of a modified LDPC code to provide high reliability for large file transfers across lossy one-way links. However, there remain a number of ancillary issues and limitations that we have not fully discussed. Among these issues are the following:

Selection of LDPC versus alternatives. We selected LDPC following a brief trade study to investigate which erasure coding method was best suitable for our use-cases. Among the most important criteria was efficiency when using large code blocks (of hundreds of megabytes or more) with hundreds of thousands of symbols within them; such as a 100 MB code block, with k=100,000 symbols, and each symbol being 8,000-12,000 bits long. Our team investigated Reed-Solomon codes, and while we found strong runtime performance for large code blocks, we encountered difficulties when code blocks stretched to be hundreds of thousands of symbols.

Files substantially smaller than the block size. The procedures defined in this paper assume an uplink data stream equal to or larger than the chosen block size. Our approach involves an implicit zero-pad, where the UDP consumer allocates a 100 MB, zero-initialized buffer. The producer only transmits PDUs containing actual uplink data, and parity blocks over the entire code block are still sent. For very small uplinks, simple repetition is sufficient.

Link scheduling for concurrent uplinks. The model we presented in this paper assumes a single uplink at a time. However, in operations multiple uplinks can occur simultaneously, which opens new problems on scheduling link use, establishing stream and packet prioritization, and avoiding starvation. This work is ongoing and subject to further study.

Software engineering optimizations. We use a simple permutation of a sequence of integers to generate indices for each parity check. Using an identical random number seed on the producer and consumer, these permutations are likewise identical. While this yields an extremely compact, dependency-free implementation to both encode and decode a code block, we believe different approaches to constructing parity check sequences have better runtime performance, particularly when decoding.

10. Conclusion

We presented and demonstrated a protocol for providing reliable and efficient delivery of large files over lossy one-way data links where ARQ is not available. The performance of this one-way protocol is roughly on par with *scp*, a popular file transfer utility leveraging ARQ and transmission control capabilities of TCP to achieve reliability. We have demonstrated the success of this protocol with files as large as 10 GB in under congested network conditions characterized with patterns of data loss. We believe the analysis and techniques contained here have further significance for many types of specialized networking environments, and may serve to optimize file downlink across space links and in other Delay-Tolerant Networks.

References

[1] R. Gallager, "Low-density parity-check codes," in *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21-28, January 1962, doi: 10.1109/TIT.1962.1057683.

[2] P. H. Seigel. (2007). An Introduction to Low-Density Parity-Check Codes. [Slides]. Available: http://cmrr-star.ucsd.edu/static/presentations/ldpc_tutorial.pdf

[3] W. A. Geisel, "Tutorial on Reed Solomon Error Correction Coding," NASA Johnson Space Flight Center, Houston, TX, Technical Memorandum 102162, Aug, 1990.

[4] T. Richardson and R. Urbanke, "The renaissance of Gallager's low-density parity-check codes," in *IEEE Communications Magazine*, vol. 41, no. 8, pp. 126-131, Aug. 2003, doi: 10.1109/MCOM.2003.1222728.

[5] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones and F. Pollara, "The Development of Turbo and LDPC Codes for Deep-Space Applications," in *Proceedings of the IEEE*, vol. 95, no. 11, pp. 2142-2156, Nov. 200s7, doi: 10.1109/JPROC.2007.905132.

[6] R. Ransier, W. Van Besien, E. Birrane, D. Srinivasan and C. Sheldon, "Maximizing data return for the Europa lander: A trade study in the application of CCSDS protocols," *2017 IEEE Aerospace Conference*, Big Sky, MT, 2017, pp. 1-13, doi: 10.1109/AERO.2017.7943850.

[7] *Licklider Transmission Protocol (LTP) for CCSDS*. CCSDS 734.1-B-1. May 2015. [Online]. Available: https://public.ccsds.org/Pubs/734x1b1.pdf

[8] RFC 793—Transmission Control Protocol, accessed on Aug. 1, 2020. [Online].

[9] RFC 3366 – Advice to Link Designers on link Automatic Repeat reQuests (ARQ), accessed on Jul. 30, 2020. [Online]

[10] RFC 8085 — UDP Usage Guidelines, accessed on Aug. 1, 2020. [Online].

[11] RFC 4838 – Delay Tolerant Networking Architecture. Accessed on Jul. 30, 2020. [Online].

[12] Low Density Parity Check Codes for Use in Deep Space Applications (131.0 Orange Book), Consultative Committee on Space Data Systems, Sep. 2007.

[13] J. Thorpe, "Low Density parity-check codes constructed from protographs," Jet Propulsion Laboratory, Pasadena, CA, IPN Prog. Rep. 42-154, Aug. 2003.

[14] J. Trostle, W. Van Besien and A. Pujari, "Protecting against DNS cache poisoning attacks," *2010 6th IEEE Workshop on Secure Network Protocols*, Kyoto, 2010, pp. 25-30, doi: 10.1109/NPSEC.2010.5634454.

[15] "Rationale Scenarios and Requirements for DTN in Space", *CCSDS Report 734.0-G-1*, August 2010.

[16] J. Kurose and K. Ross. "Principles of Reliable Data Transfer," in *Computer Networking: A Top-Down Approach, 4th ed.* Boston, MA, USA: Pearson Addison Wesley, 2008.

[17] E. J. Birrane, D. J. Copeland and M. G. Ryschkewitsch, "The path to space-terrestrial internetworking," *2017 IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*, Montreal, QC, 2017, pp. 134-139, doi: 10.1109/WiSEE.2017.8124906.

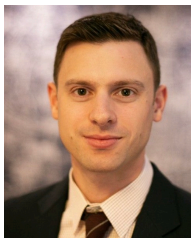
[18] Adam Langley, et al. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (*SIGCOMM '17*). Association for Computing Machinery, New York, NY, USA, 183–196. doi: https://doi.org/10.1145/3098822.3098842

[19] F. Michel, Q. De Coninck and O. Bonaventure, "QUIC-FEC: Bringing the benefits of forward erasure correction to QUIC," in *2019 IFIP Networking Conference (IFIP Networking)*, Warsaw, Poland, 2019 pp. 1-9. doi: 10.23919/IFIPNetworking46909.2019.8999475

[20] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," J. Soc. Ind. Math., vol. 8, pp. 260-269, Jun. 1960.

[21] RFC 6363 – Forward Error Correction (FEC) Framework. Accessed on Aug. 10, 2020. [Online].

BIOGRAPHY



William L. Van Besien *received a M.S. and B.S. in Computer Science from George Washington University. He is currently a Senior Engineer in the advanced concepts division at IOMAXIS, where his work focuses in distributed systems and protocol engineering. For almost a decade he served as an embedded software engineer and project manager in the flight software group at the Johns Hopkins University Applied Physics Lab, where he specialized in flight software architectures, Delay Tolerant Networking, and Autonomy/Fault-Management. His work contributed to several missions including MESSENGER, Van Allen Probes, and Parker Solar Probe.*

Benjamin J. Ferris *received his B.S. in Computer Science from DePaul University and currently works as a Senior Software Engineer in IOMAXIS' advanced concepts division. He is primarily occupied with building a high-security, cross-domain digital infrastructure. Previously, Benjamin worked at NASA Goddard Space Flight Center developing ground-system software supporting vehicle launches and payload tracking.*

Jim Dudish *received his B.S. in software engineering from Rochester Institute of Technology. He is the Director of Software Engineering for the advanced concepts division at IOMAXIS. He has 8 years' experience developing secure distributed systems, most recently leading the design and implementation of a cross-domain virtual infrastructure management system.*