NWCRG
Internet-Draft
Intended status: Experimental
Expires: September 6, 2018

J. Detchart
E. Lochin
J. Lacan
ISAE
V. Roca
INRIA

March 5, 2018

Tetrys, an On-the-Fly Network Coding protocol
draft-detchart-nwcrg-tetrys-04

Abstract

This document describes Tetrys, an On-The-Fly Network Coding (NC) protocol that can be used to transport delay and loss sensitive data over a lossy network. Tetrys can recover from erasures within a RTT-independent delay, thanks to the transmission of coded packets. It can be used for both unicast, multicast and anycast communications.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 6, 2018.

Copyright Notice

Table of Contents

1. Introduction

This document describes Tetrys, a novel network coding protocol. Network codes were introduced in the early 2000s [AHL-00] to address the limitations of transmission over the Internet (delay, capacity and packet loss). While the use of network codes is fairly recent in the Internet community, the use of application layer erasure codes in the IETF has already been standardized in the RMT [RMT] and the FECFRAME [FECFRAME] working groups. The protocol presented here can be seen as a network coding extension to standards solutions. The

current proposal can be considered as a combination of network erasure coding and feedback mechanisms [Tetrys].

The main innovation of the Tetrys protocol is in the generation of coded packets from an elastic encoding window periodically updated with the receiver's feedbacks. This update is done in such a way that any source packets coming from an input flow is included in the encoding window as long as it is not acknowledged or the encoding window did not reach a size limit. This mechanism allows for losses on both the forward and return paths and in particular is resilient to acknowledgement losses.

With Tetrys, a coded packet is a linear combination over a finite field of the data source packets belonging to the coding window. The choice of the finite field of the coefficients is a trade-off between the best performance (with non-binary coefficients) and the system constraints (binary codes in an energy constrained environment) and is driven by the application.

Thanks to the elastic encoding window, the coded packets are built on-the-fly, by using an algorithm or a function to choose the coefficients. The redundancy ratio can be dynamically adjusted, and the coefficients can be generated in different ways along a transmission. Compared to FEC block codes, this allows to reduce the bandwidth use and the decoding delay.

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Definitions, Notations and Abbreviations

The terminology used in this document is presented below. It is aligned with the FECFRAME terminology as well as with recent activities in the Network Coding Research Group.

Source symbol: a symbol that has to be transmitted between the ingress and egress of the network.

Coded symbol: a linear combination over a finite field of a set of source symbols.

Source symbol ID: a sequence number to identify the source symbols.

Coded symbol ID: a sequence number to identify the coded symbols.

Encoding coefficients: elements of the finite field characterizing the linear combination used to generate a coded symbols.

Encoding vector: set of the encoding coefficients and input source symbol IDs.

Source packet: a source packet contains a source symbol with its associated IDs.

Coded packet: a coded packet contains a coded symbol, the coded symbol's ID and encoding vector.

Input symbol: a symbol at the input of the Tetrys Encoding Building Block.

Output symbol: a symbol generated by the Tetrys Encoding Building Block. For a non systematic mode, all output symbols are coded symbols. For a systematic mode, output symbols can be the input symbols and a number of coded symbols that are linear combinations of the input symbols + the encoding vectors.

Feedback packet: a feedback packet is a packet containing information about the decoded or received source symbols. It can also bring additional information about the Packet Error Rate or the number of various packets in the receiver decoding window

Elastic Encoding Window: an encoder-side buffer that stores all the non-acknowledged source packets of the input flow that are involved in the coding process.

Coding Coefficient Generator Identifier: a unique identifier that define a function or an algorithm allowing to generate the encoding vector.

Code rate: Define the rate between the number of input symbols and the number of output symbols.

3. Architecture

-- Editor's note: The architecture used in this document should be aligned with the future NC Architecture document [NWCRG-ARCH]. --

3.1. Use Cases

Tetrys is well suited, but not limited to the use case where there is a single flow originated by a single source, with intra stream coding that takes place at a single encoding node. Note that the input stream can be a multiplex of several upper layer streams.

Transmission can be over a single path or multiple paths. In addition, the flow can be sent in unicast, multicast, or anycast mode. This is the simplest use-case, that is very much inline with currently proposed scenarios for end-to-end streaming.

3.2. Overview

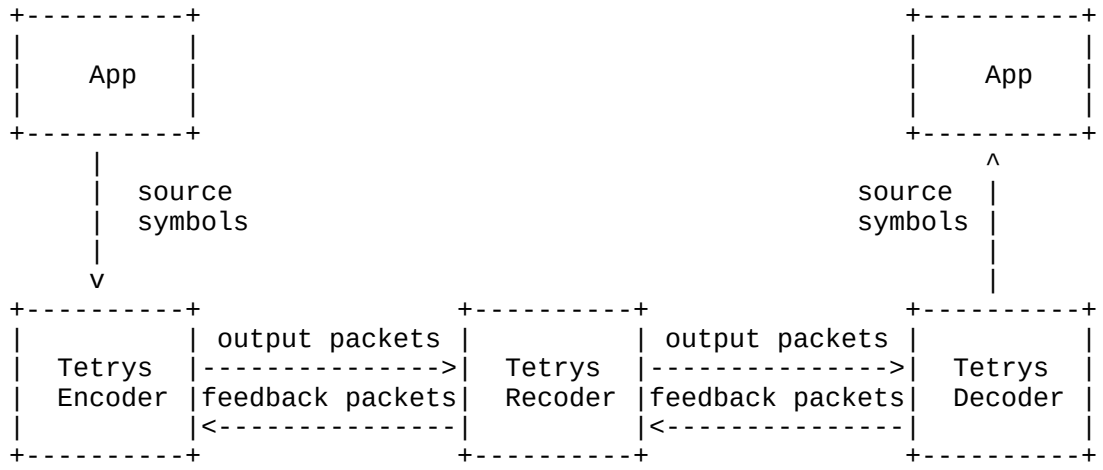

Figure 1: Tetrys Architecture

The Tetrys protocol features several key functionalities:

- On-the-fly encoding;
- Recoding;
- Decoding;
- Signaling, to carry in particular the symbol identifiers in the encoding window and the associated coding coefficients when meaningful, in a manner that was previously used in FEC;
- Feedback management;
- Elastic window management;
- Channel estimation;
- Dynamic adjustment of the code rate and flow control;
- Congestion control management (if appropriate);

-- Editor’s note: must be discussed --

- Tetrys packet header creation and processing;
- -- Editor’s note: something else? --

These functionalities are provided by several building blocks:

- The Tetrys Building Block: this BB is used during encoding, recoding and decoding processes. It must be noted that Tetrys does not mandate a specific building block. Instead any building block compatible with the elastic encoding window feature of Tetrys can be used.
- The Window Management Building Block: this building block is in charge of managing the encoding encoding window at a Tetrys sender.

  -- Editor’s note: Is it worth moving it in a dedicated BB? To be discussed --

- Other ?

In order to enable future components and services to be added dynamically, Tetrys adds a header extension mechanism, compatible with that of LCT, NORM, FECFRAME [REFS].

4. Packet Format

4.1. Common Header Format

All types of Tetrys packets share the same common header format (see Figure 2).

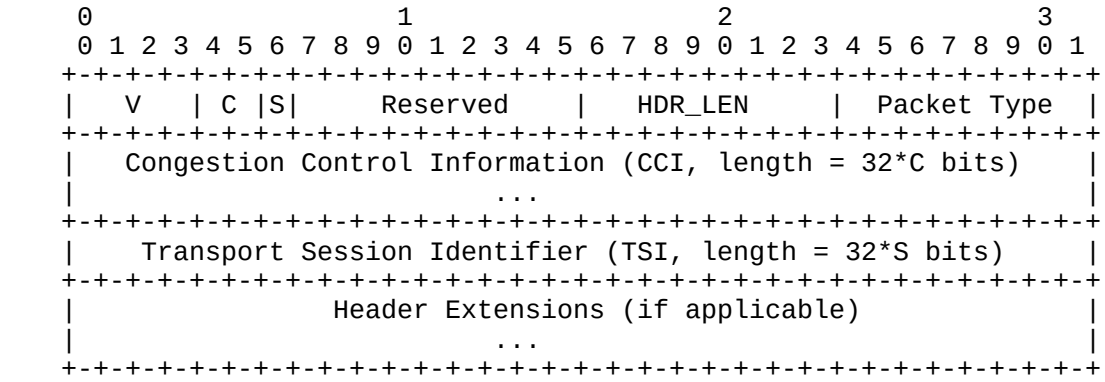

Figure 2: Common Header Format

-- Editor’s note: this format inherits from the LCT header format (RFC 5651) with slight modifications. --

- Tetrys version number (V): 4 bits. Indicates the Tetrys version number. The Tetrys version number for this specification is 1.

- Congestion control flag (C): 2 bits. C=0 indicates the Congestion Control Information (CCI) field is 0 bits in length. C=1 indicates the CCI field is 32 bits in length. C=2 indicates the CCI field is 64 bits in length. C=3 indicates the CCI field is 96 bits in length.

  -- Editor’s note: version number and congestion control to be discussed --

- Transport Session Identifier flag (S): 1 bit. This is the number of full 32-bit words in the TSI field. The TSI field is 32*S bits in length, i.e., the length is either 0 bits or 32 bits.

- Reserved (Resv): 9 bits. These bits are reserved. In this version of the specification, they MUST be set to zero by senders and MUST be ignored by receivers.

- Header length (HDR_LEN): 8 bits. Total length of the Tetrys header in units of 32-bit words. The length of the Tetrys header MUST be a multiple of 32 bits. This field can be used to directly access the portion of the packet beyond the Tetrys header, i.e., to the first other header if it exists, or to the packet payload if it exists and there is no other header, or to the end of the packet if there are no other headers or packet payload.

- Packet Type: 8 bits. Type of packet.

- Congestion Control Information (CCI): 0, 32, 64, or 96 bits Used to carry congestion control information. For example, the congestion control information could include layer numbers, logical channel numbers, and sequence numbers. This field is opaque for the purpose of this specification. This field MUST be 0 bits (absent) if C=0. This field MUST be 32 bits if C=1. This field MUST be 64 bits if C=2. This field MUST be 96 bits if C=3.

- Transport Session Identifier (TSI): 0 or 32 bits The TSI uniquely identifies a session among all sessions from a particular sender. The TSI is scoped by the IP address of the sender, and thus the IP address of the sender and the TSI together uniquely identify the session. Although a TSI in conjunction with the IP address of the sender always uniquely identifies a session, whether or not the TSI is included in the Tetrys header depends on what is used as the TSI value. If the underlying transport is UDP, then the 16-bit UDP source port number MAY serve as the TSI for the session. If there is no underlying TSI provided by the network, transport or any other layer, then the TSI MUST be included in the Tetrys header.

4.1.1. Header Extensions

Header Extensions are used in Tetrys to accommodate optional header fields that are not always used or have variable size. The presence of Header Extensions can be inferred by the Tetrys header length (HDR_LEN). If HDR_LEN is larger than the length of the standard header, then the remaining header space is taken by Header Extensions.

If present, Header Extensions MUST be processed to ensure that they are recognized before performing any congestion control procedure or otherwise accepting a packet. The default action for unrecognized Header Extensions is to ignore them. This allows the future introduction of backward-compatible enhancements to Tetrys without changing the Tetrys version number. Non-backward-compatible Header Extensions CANNOT be introduced without changing the Tetrys version number.

There are two formats for Header Extensions, as depicted in Figure 3. The first format is used for variable-length extensions, with Header Extension Type (HET) values between 0 and 127. The second format is used for fixed-length (one 32-bit word) extensions, using HET values from 128 to 255.

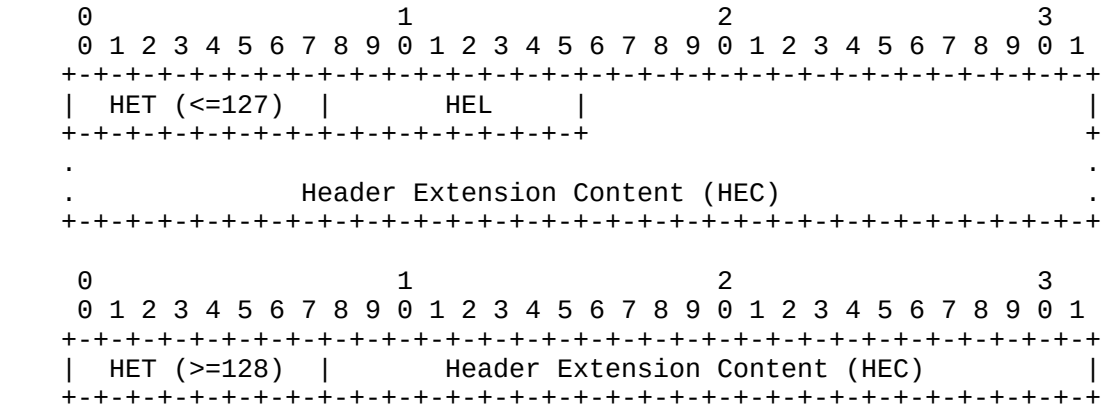

Figure 3: Header Extension Format

- Header Extension Type (HET): 8 bits The type of the Header Extension. This document defines a number of possible types. Additional types may be defined in future versions of this specification. HET values from 0 to 127 are used for variable-length Header Extensions. HET values from 128 to 255 are used for fixed-length 32-bit Header Extensions.

- Header Extension Length (HEL): 8 bits The length of the whole Header Extension field, expressed in multiples of 32-bit words. This field MUST be present for variable-length extensions (HETs between 0 and 127) and MUST NOT be present for fixed-length extensions (HETs between 128 and 255).

- Header Extension Content (HEC): variable length The content of the Header Extension. The format of this sub-field depends on the Header Extension Type. For fixed-length Header Extensions, the HEC is 24 bits. For variable-length Header Extensions, the HEC field has variable size, as specified by the HEL field. Note that the length of each Header Extension MUST be a multiple of 32 bits. Also note that the total size of the Tetrys header, including all Header Extensions and all optional header fields, cannot exceed 255 32-bit words.

4.2. Source Packet Format

A source packet is the encapsulation of a Common Packet Header, a Source Symbol ID and a source symbol (payload). The source symbols can have variable sizes.

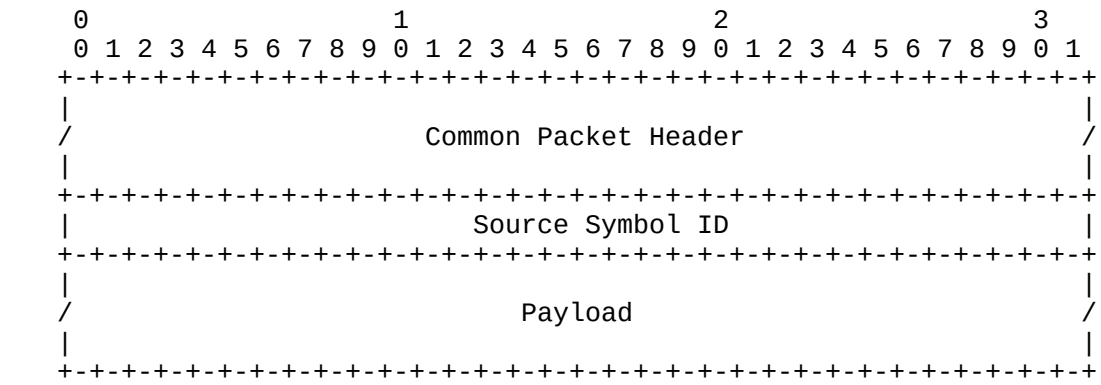

Figure 4: Source Packet Format

Common Packet Header: a common packet header (as common header format) where Packet Type=0.

Source Symbol ID: the sequence number to identify a source symbol.

Payload: the payload (source symbol)

4.3. Coded Packet Format

A coded packet is the encapsulation of a Common Packet Header, a Coded Symbol ID, the associated Encoding Vector and a coded symbol (payload). As the source symbols CAN have variable sizes, each source symbol size need to be encoded, and the result must be stored in the coded packet as the Encoded Payload Size (16 bits): as it is an optional field, the encoding vector MUST signal the use of variable source symbol sizes with the field V (see Section 6.1.1.2).

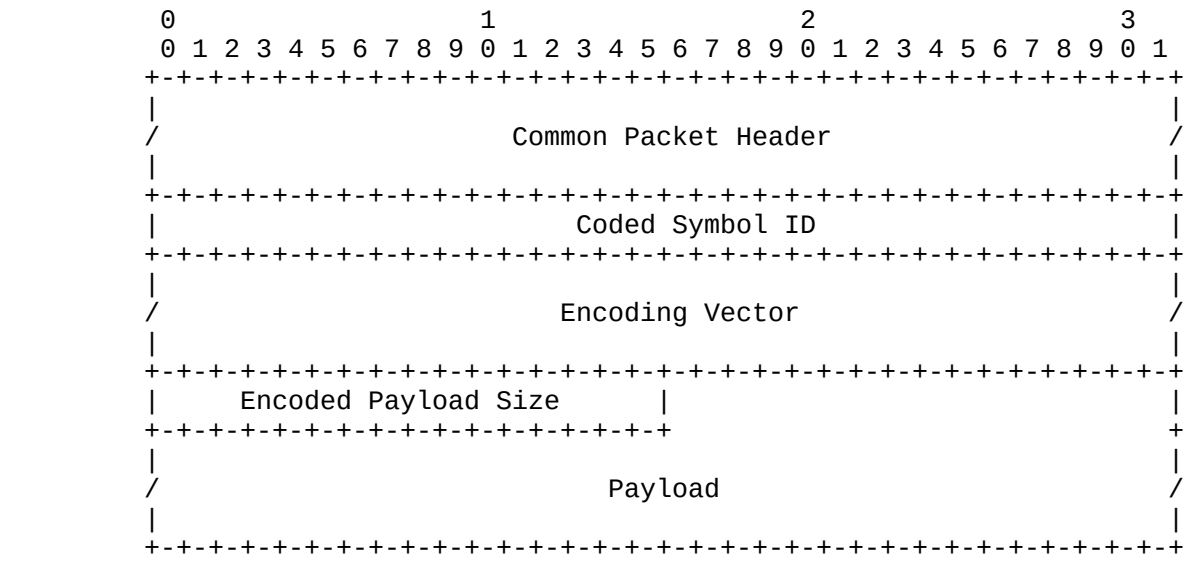

Figure 5: Coded Packet Format

Common Packet Header: a common packet header (as common header format) where Packet Type=1.

Coded Symbol ID: the sequence number to identify a coded symbol.

Encoding Vector: an encoding vector to define the linear combination used (coefficients, and source symbols).

Encoded Payload Size: the coded payload size used if the source symbols have variable size (optional, Section 6.1.1.2)).

Payload: the coded symbol.

4.4. Acknowledgement Packet Format

A Tetrys Decoding Building Block or Tetrys Recoding Building Block MAY send back to another building block some Acknowledgement packets. They contain information about what it has received and/or decoded, and other information such as a packet loss rate or the size of the decoding buffers. The acknowledgement packets are OPTIONAL hence they could be omitted or lost in transmission without impacting the protocol behavior.

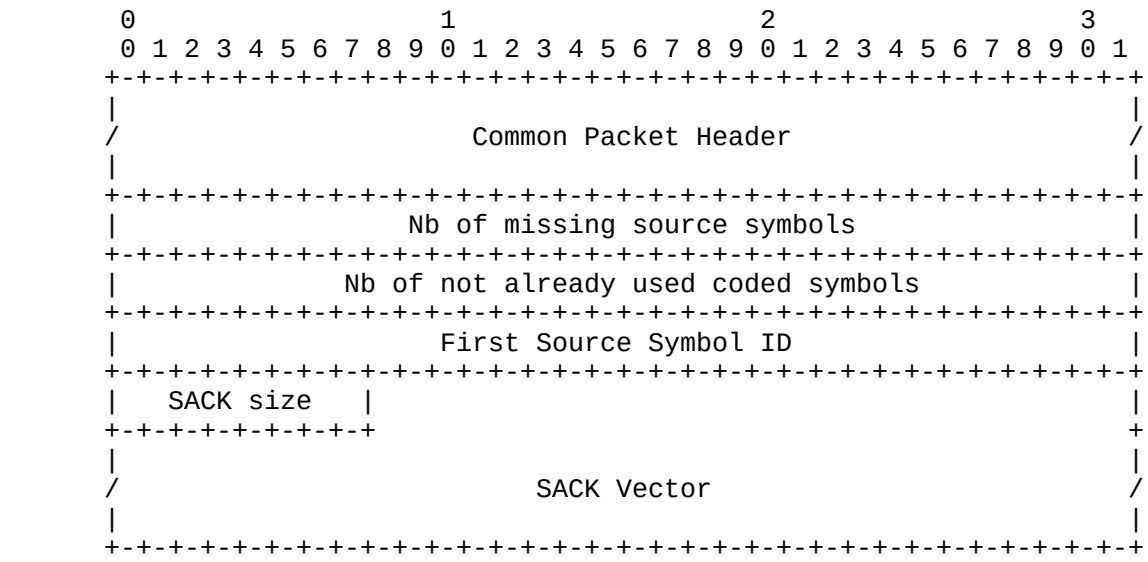

Figure 6: Acknowledgement Packet Format

Common Packet Header: a common packet header (as common header format) where Packet Type=2.

Nb missing source symbols: the number of missing source symbols in the receiver since the beginning of the session.

Nb of not already used coded symbols: the number coded symbols at the receiver that have not already been used for decoding (e.g., the linear combinations contains at least 2 unknown source symbols).

First Source Symbol ID: ID of the first source symbol to consider for acknowledgement.

SACK size: the size of the SACK vector in 32-bit words. For instance, with value 2, the SACK vector is 64 bits long.

SACK vector: bit vector indicating the acknowledged symbols from the first source symbol ID. The "First Source Symbol" is included in this bit vector. A bit equal to 1 at position i means that the source symbol of ID equal to "First Source Symbol ID" + i is acknowledged by this acknowledgment packet.

5. The Coding Coefficient Generator Identifiers

5.1. Definition

The Coding Coefficient Generator Identifiers define a function or an algorithm to build the coding coefficients used to generate the coded symbols. They MUST be known by all the Tetrys encoders, recoders or decoders.

5.2. Table of Identifiers

0000: GF256 (or GF(2**8)) Vandermonde based coefficients. Each coefficient is built as alpha^((source_symbol_id*coded_symbol_id) % 255).

0001: GF16 (or GF(2**4)) Vandermonde based coefficients. Each coefficient is built as alpha^((source_symbol_id*coded-symbol_id) % 15).

0010: SRLC.

Others: To be discussed.

6. Tetrys Basic Functions

6.1. Encoding

At the beginning of a transmission, a Tetrys Encoding Building Block or MUST choose an initial code rate (added redundancy) as it doesn’t know the packet loss rate of the channel. In steady state, depending on the code-rate, the Tetrys Encoding Building Block CAN generate coded symbols when it receives a source symbol from the application or some feedback from the decoding or recoding blocks.

When a Tetrys Encoding Building Block needs to generate a coded symbol, it considers the set of source symbols stored in the Elastic Encoding Window. These source symbols are the set of source symbols which are not yet acknowledged by the receiver.

A Tetrys Encoding Building Block SHOULD set a limit to the Elastic Encoding Window maximum size. This allows to control the algorithmic complexity at the encoder and decoder by limiting the size of linear combinations. It is also needed in situations where acknowledgment packets are all lost or absent.

At the generation of a coded symbol, the Tetrys Encoding Building Block generates an encoding vector containing the IDs of the source symbols stored in the Elastic Encoding Window. For each source

symbol, a finite field coefficient is determined using a Coding Coefficient Generator. This generator CAN take as input the source symbol ID and the coded symbol ID and CAN determine a coefficient in a deterministic way. A classical example of such deterministic function is a generator matrix where the rows are indexed by the source symbol IDs and the columns by the coded symbol IDs. For example, the entries of this matrix can be built from a Vandermonde structure, like Reed-Solomon codes, or from a sparse binary matrix, like Low-Density Generator Matrix codes. Finally, the coded symbol is the sum of the source symbols multiplied by their corresponding coefficients.

6.1.1. Encoding Vector Formats

Each coded packet contains an encoding vector. The encoding vectors CAN contain the ID and/or coefficient of each source symbol contained in the coded symbol.

6.1.1.1. Transmitting the source symbol IDs

The source symbol IDs are organized as a sorted list of 32-bit unsigned integers. Depending on the feedback, the source symbol IDs can be successive or not in the list.

If they are successive, only the boundaries CAN be stored in the encoding vector: it just needs 2*32-bit of information.

If not, the edge blocks CAN be stored directly, or a differential transform to reduce the number of bits needed to represent an ID CAN be used.

6.1.1.1.1. Compressed list of Source symbol IDs

Assume the symbol IDs used in the combination are: [1..3],[5..6],[8..10].

1. Keep the first element in the packet as the first_source_id: 1.

2. Apply a differential transform to the others elements ([3,5,6,8,10]) which removes the element i-1 to the element i, starting with the first_source_id as i0, and get the list L => [2,2,1,2,2]

3. Compute b, the number of bits needed to store all the elements, which is ceil(log2(max(L))): here, 2 bits.

4. Write b in the corresponding field, and write all the b * [(Nb IDs) - 1] elements in a bit vector, here: 10 10 01 10 10.

6.1.1.1.2. Decompressing the Source symbol IDs

When a Tetrys Decoding Building Block wants to reverse the operations, this algorithm is used:

1. Rebuild the list of the transmitted elements by reading the bit vector and b: [10 10 01 10 10] => [2,2,1,2,2]

2. Apply the reverse transform by adding successively the elements, starting with first_source_id: [1,1+2,(1+2)+2,(1+2+2)+1,...] => [1,3,5,6,8,10]

3. Rebuild the IDs using the list and first_source_id: [1..3],[5..6],[8..10].

6.1.1.2. Encoding Vector Format

The encoding vector CAN be used to store the source symbol IDs included in the associated coded symbol, the coefficients used in the combination, or both. It CAN be used to send only the number of source symbols included in the coded symbol.

The encoding vector format uses a 4-bit Coding Coefficient Generator Identifier to identity the algorithm to generate the coefficients, and contains a set of IDs from the source symbol used in the combination. In this format, the number of IDs is stored as a 8-bit unsigned integer. To reduce the overhead, a compressed way to store the symbol IDs CAN be used: the IDs are not stored as themselves, but stored as the difference between the previous.

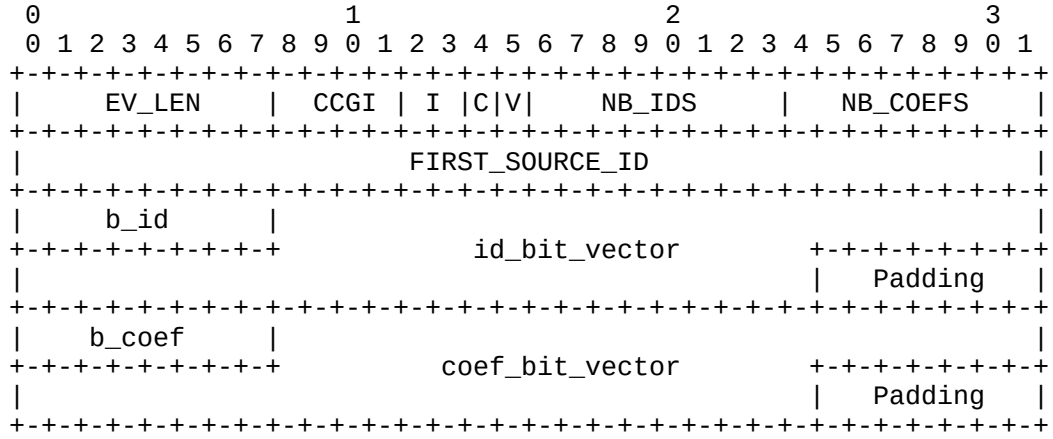

Figure 7: Encoding Vector Format

- Encoding Vector Length (EV_LEN) (8-bits): size in units of 32-bit words.
- Coding Coefficient Generator Identifier (CCGI): 4-bit ID to identify the algorithm or the function used to generate the coefficients (see Section 5). As a CCGI is included in each encoded vector, it can dynamically change between the generation of 2 coded symbols.
- Store the Source symbol IDs (I) (2 bits):
  - 00 means there is no source symbol ID information.
  - 01 means the encoding vector contains the edge blocks of the source symbol IDs without compression.
  - 10 means the encoding vector contains the compressed list of the source symbol IDs.
  - 11 means the encoding vector contains the compressed edge blocks of the source symbol IDs.
- Store the coefficients (C): 1 bit to know if an encoding vector contains information about the coefficients used.
- Having source symbols with variable size (V): set V to 1 if the combination which refers the encoding vector is a combination of source symbols with variable sizes. In this case, the coded packets MUST have the 'Encoded Payload Size' field.
- Number of IDs used to store the source symbol IDs (NB_IDS): the number of IDs stored in the encoding vector (depending on I).
- Number of coefficients (NB_COEFS): The number of the coefficients used to generate the associated coded symbol.
- The first source Identifier (FIRST_SOURCE_ID): the first source symbol ID used in the combination.
- Number of bits for each ID (b_id): the number of bits needed to store the IDs (see Section 6.1.1.1).
- Information about the source symbol IDs (id_bit_vector): if I=01, store the edge blocks as b_id * (NB_IDS - 1). If I=10, store in a compressed way the edge blocks.
- Number of bits needed to store each coefficient (b_coef): the number of bits used to store the coefficients.

- The coefficients (coef_bit_vector): The coefficients stored (as a vector of b_coef * NB_COEFS).

- Padding: padding to have an Encoding Vector size multiple of 32-bit (for the id and coefficient part).

6.2. The Elastic Encoding Window

When an input source symbol is passed to a Tetrys Encoding Building Block, it is added to the Elastic Encoding Window. This window MUST have a limit set by the encoding building Block (depending of the use case: unicast, multicast, file transfer, real-time transfer, ...). If the Elastic Encoding Window reached its limit, the window slides over the symbols: the first (oldest) symbols are removed. Then, a packet containing this symbol can be sent onto the network. As an element of the coding window, this symbol is included in the next linear combinations created to generate the coded symbols.

As explained below, the receiver or the recoder sends periodic feedback indicating the received or decoded source symbols. In the case of a unicast transmission, when the sender receives the information that a source symbol was received and/or decoded by the receiver, it removes this symbol from the coding window.

In a multicast transmission:

- If the acknowledgement packets are not enabled, the coding window grows up to a limit. When the limit is reached, the oldest symbols are removed from the coding window.

- If the acknowledgement packets are enabled, a source symbol is removed from the coding window when all the receivers have received or decoded it or when the coding window reaches its limit.

6.3. Recoding

6.3.1. Principle

A Tetrys Recoding Block maintains a list of the ID of the source symbols included in the Elastic Coding Window of the sender. It also stores a set of received source and coded symbols able to regenerate the set or a subset of the symbols of the Elastic Coding Window. In other words, if R1, ..., Rt represent t received symbols and S1, ..., Sk represent the set or a subset of the source symbols of the Elastic Coding Window, there exists a t*k-matrix M such that (R1, ..., Rt).M = (S1, ..., Sk).

6.3.2. Generating a coded symbol at an intermediate node

At the generation of a coded symbol, the Tetrys Recoding Building Block generates an encoding vector containing the IDs of the source symbols stored in the Elastic Encoding Window or in the subset of the Elastic Encoding Window that it is able to regenerate. The Tetrys Recoding Building Block then generates a new coded symbol ID different from the received coded symbol IDs. From this coded symbol ID and the source symbol IDs of (S1, ..., Sk), a finite field coefficient is determined using a Coding Coefficient Generator. Let (a1, ...,ak) denote the obtained coefficients. To compute the linear combination (s1, ..., Sk).transpose(a1, ..., ak) the Tetrys Recoding Building block computes the vector v = (a1, ...,ak).transpose(M) and then computes the coded symbol R = (R1, ..., Rt).transpose(v). It can be verified that the new coded symbol is obtained without any decoding operation.

6.4. Decoding

A classical matrix inversion is sufficient to recover the source symbols.

7. Security Considerations

N/A

8. Privacy Considerations

N/A

9. IANA Considerations

N/A

10. Acknowledgments

N/A

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

11.2. Informative References

[AHL-00] Ahlswede, R., Ning Cai, Li, S., and R. Yeung, "Network information flow", IEEE Transactions on Information Theory vol.46, no.4, pp.1204,1216, July 2000.

[FECFRAME] Watson, M., Begen, A., and V. Roca, "Forward Error Correction (FEC) Framework", Request for Comments 6363, October 2011.

[NWCRG-ARCH] NWCRG, "Network Coding Architecture", TBD TBD.

[RMT] Vicisano, L., Gemmel, J., Rizzo, L., Handley, M., and J. Crowcroft, "Forward Error Correction (FEC) Building Block", Request for Comments 3452, December 2002.

[Tetrys] Lacan, J. and E. Lochin, "Rethinking reliability for long-delay networks", International Workshop on Satellite and Space Communications 2008 (IWSSC08), October 2008.

Authors' Addresses

Jonathan Detchart
ISAE
10, avenue Edouard-Belin
BP 54032
Toulouse CEDEX 4 31055
France

Email: jonathan.detchart@isae.fr

Emmanuel Lochin
ISAE
10, avenue Edouard-Belin
BP 54032
Toulouse CEDEX 4 31055
France

Email: emmanuel.lochin@isae.fr

Jerome Lacan
ISAE
10, avenue Edouard-Belin
BP 54032
Toulouse CEDEX 4 31055
France

Email: jerome.lacan@isae.fr

Vincent Roca
INRIA
655, av. de l'Europe
Inovallee; Montbonnot
ST ISMIER cedex 38334
France

Email: vincent.roca@inria.fr

NWCRG
Internet-Draft
Intended status: Experimental
Expires: August 10, 2018

J. Heide
Steinwurf Aps
S. Shi
M. Medard
Code On Network Coding LLC
V. Chook
Inmarsat PLC

February 6, 2018

Random Linear Network Coding (RLNC)-Based Symbol Representation draft-heide-nwcrg-rlnc-00

Abstract

This document describes the use of Random Linear Network Coding (RLNC) schemes for erasure correction in data transfer, with an emphasis on RLNC encoded symbol representations in data packets. Both block and sliding window RLNC code implementations are described. By providing erasure correction using randomly generated repair symbols, such RLNC-based schemes offer advantages in accommodating varying frame sizes and dynamically changing connections, reducing the need for feedback, and lowering the amount of state information needed at the sender and receiver.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 10, 2018.

Table of Contents

1. Introduction

Network Coding is an emerging coding discipline that jointly improves network reliability and efficiency. In general communication networks, source coding is performed as a compression mechanism to reduce data redundancy and to reduce resources necessary for data transportation over the network. Channel coding, on the other hand, introduces redundancy for data transmission reliability over lossy channels. Network coding adds another layer of coding in-between these two. Random Linear Network Coding (RLNC) is an efficient network coding approach that enables network nodes to generate

independently and randomly linear mappings of input to output data symbols over a finite field.

This document describes the use of RLNC schemes, with an emphasis on RLNC encoded symbol representations in data packets. Specifically, a block RLNC scheme and a sliding window RLNC scheme are discussed in the context of varying data frame sizes.

1.1. Random Linear Network Coding (RLNC) Basics

Unlike conventional communication systems based on the "store-and-forward" principle, RLNC allows network nodes to independently and randomly combine input source data into coded symbols over a finite field [HK03]. Such an approach enables receivers to decode and recover the original source data as long as enough linearly independent coded symbols, with sufficient degrees of freedom, are received. At the sender, RLNC can introduce redundancy into data streams in a granular way. At the receiver, RLNC enables progressive decoding and reduces feedback necessary for retransmission. Collectively, RLNC provides network utilization and throughput improvements, high degrees of robustness and decentralization, reduces transmission latency, and simplifies feedback and state management.

To encode using RLNC, original source data are divided into symbols of a given size and linearly combined. Each symbol is multiplied with a scalar coding coefficient drawn randomly from a finite field, and the resulting coded symbol is of the same size as the original data symbols.

Thus, each RLNC encoding operation can be viewed as creating a linear equation in the data symbols, where the random scalar coding coefficients can be grouped and viewed as a coding vector. Similarly, the overall encoding process where multiple coded symbols are generated can be viewed as a system of linear equations with randomly generated coefficients. Any number of coded symbols can be generated from a set of data symbols, similarly to expandable forward error correction codes specified in [RFC5445] and [RFC3453]. Coding vectors must be implicitly or explicitly transmitted from the sender to the receiver for successful decoding of the original data. For example, sending a seed for generating pseudo-random coding coefficients can be considered as an implicit transmission of the coding vectors. In addition, while coding vectors are often transmitted together with coded data in the same data packet, it is also possible to separate the transmission of coding coefficient vectors from the coded data, if desired.

To reconstruct the original data from coded symbols, a network node collects a finite but sufficient number of degrees of freedom for solving the system of linear equations. This is beneficial over conventional approaches as the network node is no longer required to gather each individual data symbol. In general, the network node needs to collect slightly more independent coded symbols than there are original data symbols, where the slight overhead arises because coding coefficients are drawn at random, with a non-zero probability that a coding vector is linearly dependent on another coding vector, and that one coded symbol is linearly dependent on another coded symbol. This overhead can be made arbitrarily small, provided that the finite field used is sufficiently large.

A unique advantage of RLNC is the ability to re-encode or "recode" without first decoding. Recoding can be performed jointly on existing coded symbols, partially decoded symbols, or uncoded systematic data symbols. This feature allows intermediate network nodes to re-encode and generate new linear combinations on the fly, thus increasing the likelihood of innovative transmissions to the receiver. Recoded symbols and recoded coefficient vectors have the same size as before and are indistinguishable from the original coded symbols and coefficient vectors.

In practical implementations of RLNC, the original source data are often divided into multiple coding blocks or "generations" where coding is performed over each individual generation to lower the computational complexity of the encoding and decoding operations. Alternatively, a convolutional approach can be used, where coding is applied to overlapping spans of data symbols, possibly of different spanning widths, viewed as a sliding coding window. In generation-based RLNC, not all symbols within a single generation need to be present for coding to start. Similarly, a sliding window can be variable-sized, with more data symbols added to the coding window as they arrive. Thus, innovative coded symbols can be generated as data symbols arrive. This "on-the-fly" coding technique reduces coding delays at transmit buffers, and together with rateless encoding operations, enables the sender to start emitting coded packets as soon as data is received from an upper layer in the protocol stack, adapting to fluctuating incoming traffic flows. Injecting coded symbols based on a dynamic transmission window also breaks the decoding delay lower bound imposed by traditional block codes and is well suited for delay-sensitive applications and streaming protocols.

When coded symbols are transmitted through a communication network, erasures may occur, depending on channel conditions and interactions with underlying transport protocols. RLNC can efficiently repair such erasures, potentially improving protocol response to erasure events to ensure reliability and throughput over the communication

network. For example, in a point-to-point connection, RLNC can proactively compensate for packet erasures by generating Forward Erasure Correcting (FEC) redundancy, especially when a packet erasure probability can be estimated. As any number of coded symbols may be generated from a set of data symbols, RLNC is naturally suited for adapting to network conditions by adjusting redundancy dynamically to fit the level of erasures, and by updating coding parameters during a session. Alternatively, packet erasures may be repaired reactively by using feedback requests from the receiver to the sender, or by a combination of FEC and retransmission. RLNC simplifies state and feedback management and coordination as only a desired number of degrees of freedom needs to be communicated from the receiver to the sender, instead of indications of the exact packets to be retransmitted. The need to exchange packet arrival state information is therefore greatly reduced in feedback operations.

The advantages of RLNC in state and feedback management are apparent in a multicast setting. In this one-to-many setup, uncorrelated losses may occur, and any retransmitted data symbol is likely to benefit only a single receiver. By comparison, a transmitted RLNC coded symbol is likely to carry a new degree of freedom that may correct different errors at different receivers simultaneously. Similarly, RLNC offers advantages in coordinating multiple paths, multiple sources, mesh networking and cooperation, and peer-to-peer operations.

A more detailed introduction to network coding including RLNC is provided in the books [MS11] and [HL08].

1.2. Generation-Based RLNC

This section describes a generation-based RLNC scheme.

In generation-based RLNC, input data as received from an upper layer in the protocol stack is segmented into equal-sized blocks, denoted as generations, and each generation is further segmented into equal-sized data symbols for encoding, with paddings added when necessary. Encoding and decoding are performed over each individual generation. Figure 1 below provides an illustrative example where each generation includes four data symbols, and a systematic RLNC code is generated with rate 2/3.

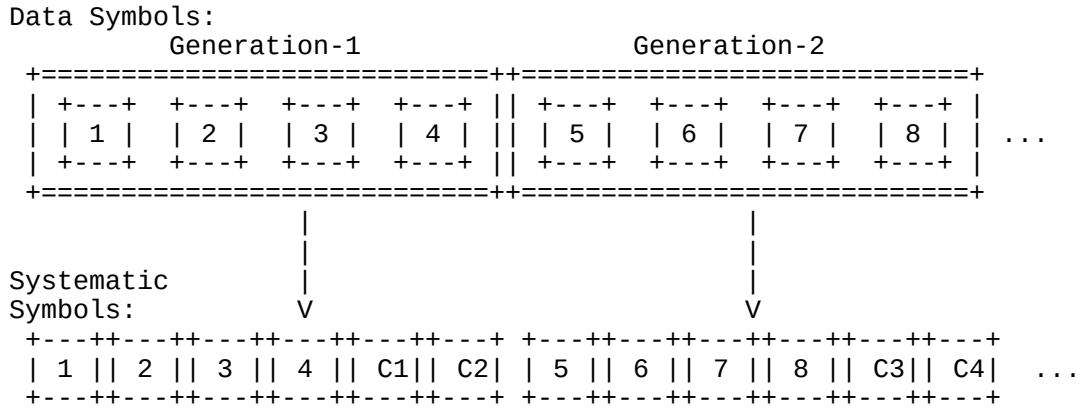

Figure 1: Generation-based RLNC with rate 2/3, systematic encoding performed on data symbols within each generation.

Symbols can be of any size, although symbol sizes typically depend on application or system specifications. In scenarios with highly varying input data frame sizes, a small symbol size may be desirable for achieving flexibility and transmission efficiency, with one or more symbols concatenated to form a coded data packet. In this context, existing basic FEC schemes [RFC5445] do not support the use of a single header for multiple coded symbols, whereas the symbol representation design as described herein provides this option.

For any protocol that utilizes generation-based RLNC, a setup process is necessary for establishing a connection and conveying coding parameters from the sender to the receiver. Such coding parameters can include one or more of field size, code specifications, index of the current generation being encoded at the sender, generation size, code rate, and desired feedback frequency or probability. Some coding parameters are updated dynamically during the transmission process, reflecting the coding operations over sequences of generations, and adjusting to channel conditions and resource availability. For example, an outer header can be added to the symbol representation specified in this document to indicate the current generation encoded within the symbol representation. Such information is essential for proper recoding and decoding operations, but the exact design of the outer header is outside the scope of the current document. At the minimum, an outer header should indicate the current generation, generation size, symbol size, and field size. Section 2 provides a detailed discussion of coding parameter considerations.

1.3. Sliding Window RLNC

This section describes a sliding-window RLNC scheme. Sliding window RLNC was first described in [SS09]

In sliding-window RLNC, input data as received from an upper layer in the protocol stack is segmented into equal-sized data symbols for encoding. In some implementations, the sliding encoding window can expand in size as new data packets arrive, until it is closed off by an explicit instruction, such as a feedback message that re-initiates the encoding window. In some implementations, the size of the sliding encoding window is upper bounded by some parameter, fixed or dynamically determined by online behavior such as packet loss or congestion estimation. Figure 3 below provides an example of a systematic finite sliding window code with rate 2/3.

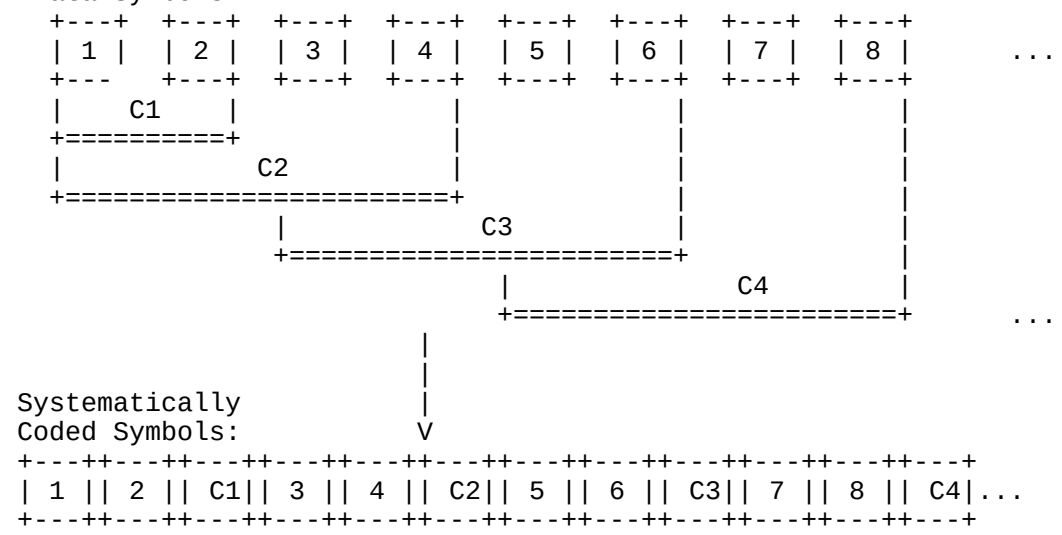

Figure 3: Finite sliding-window RLNC with code rate 2/3, systematic encoding performed on data symbols within the sliding coding window.

For any protocol that utilizes sliding-window RLNC, a setup process is necessary for establishing a connection and conveying coding parameters from the sender to the receiver. Such coding parameters can include one or more of field size, code specifications, symbol ordering, encoding window position, encoding window size, code rate, and desired feedback frequency or probability. Some coding parameters can also be updated dynamically during the transmission process in accordance to channel conditions and resource

availability. For example, an outer header can be added to the symbol representation specified in this document to indicate an encoding window position, as a starting index for current data symbols being encoded within the symbol representation. Again, such information is essential for proper recoding and decoding operations, but the exact design of the outer header is outside the scope of the current document. At the minimum, an outer header should indicate the current encoding window position, encoding window size, symbol size, and field size. Section 2 provides a detailed discussion of coding parameter considerations.

Once a connection is established, RLNC coded packets comprising one or more coded symbols are transmitted from the sender to the receiver. The sender can transmit in either a systematic or coded fashion, with or without receiver feedback. In progressive decoding of RLNC coded symbols, the notion of "seen" packets can be utilized to provide degree of freedom feedbacks. Seen packets are those packet that have contributed to a received coded packet, where generally the oldest such packet that has yet to be declared seen is declared as seen [SS09].

1.4. Recoding

Recoding is the process where coded or partially decoded symbols are re-encoded without first being fully decoded. To recode, both coded symbols and corresponding coding coefficient vectors are linearly combined, respectively, with new randomly generated recoding coefficients. Recoded symbols and recoded coefficient vectors generally have the same size as before and are indistinguishable from the original coded symbols and coding coefficient vectors. Recoding is typically performed at intermediate network nodes, in either an intra-session or an inter-session fashion. Intra-session coding refers to coding between packets of the same flow or session, while inter-session coding refers to combination of packets from different flows or sessions in a network.

As recoding requires the same operations on the coding coefficient vectors as applied to the coded symbols, coding coefficients must be updated by recoding coefficients. This is generally achieved by having coding coefficient accessible at recoding nodes so that they may be updated using the recoding coefficients. Thus, either the original coding coefficients or reversible representations of the coding coefficients need to be communicated from upstream nodes to the recoding nodes.

2. Coding Parameter Design Considerations

For any protocol that utilizes generation-based or sliding-window RLNC, several coding parameters must be communicated from the sender to the receiver as part of the protocol design. Without elaborating on all such parameters, this section examines those essential for symbol representation design, including field size, symbol size, maximum number of symbols, and maximum generation or window size.

As RLNC is performed over a finite field, field size determines the complexity of the required mathematical computations. Larger field sizes translate to higher probability of successful decoding, as randomly generated coding coefficient vectors are more likely to be independent from each other. However, larger field sizes may also result in higher computational complexity, leading to longer decoding latency, higher energy usage, and other hardware requirements for both the encoder and the decoder. A finite field size of 2 or the binary Finite Field FF(2) should always be supported since addition, multiplication, and division over FF(2) are equivalent to elementary XOR, AND, and IDENTITY operations respectively. It is also desirable to support a field size of 2^8, corresponding to a single byte, and where operations are performed over the binary extension field FF(2^8) with polynomial x^8+x^4+x^3+x^2+1.

The choice of symbol size typically depends on the application or system specification. For example, a symbol size may be chosen based on the size of a maximum transmission unit (MTU) so that datagrams are not fragmented as they traverse a network, while also ensuring no symbol bits are unnecessarily wasted. The current symbol representation design is flexible and can accommodate any symbol size in bytes. For example, an IP packet is typically in the range between 500 and 1500 bytes, while a much smaller datagram having a size of 90 bytes may be used by satellite communication networks. The current symbol representation can be configured to support either or both cases in different implementations.

The generation size or coding window size is a tradeoff between the strength of the code and the computational complexity of performing the coding operations. With a larger generation/window size, fewer generations or coding windows are needed to enclose a data message of a given size, thus reducing protocol overhead for coordinating individual generations or coding windows. In addition, a larger generation/window size increases the likelihood that a received coded symbol is innovative with respect to previously received symbols, thus amortizing retransmission or FEC overheads. Conversely, when coding coefficients are attached, larger generation/window size also lead to larger overheads per packet. The generation/window size to

be used can be signaled between the sender and receiver when the connection is first established.

Lastly, to successfully decode RLNC coded symbols, sufficient degrees of freedom are required at the decoder. The maximum number of redundant symbols that can be transmitted is therefore limited by the number of linearly independent coding coefficient vectors that can be supported by the system. For example, if coding vectors are constructed using a pseudo-random generator, the maximum number of redundant symbols that can be transmitted is limited by the number of available generator states.

3. Symbol Representation

This section provides a symbol representation design for implementing RLNC-based erasure correction schemes. In this symbol representation design, multiple symbols are concatenated and associated with a single symbol representation header.

The symbol representation design is provided for constructing a data payload portion of a data packet for a protocol that utilizes a generation-based or sliding-window RLNC, where recoding can be used at intermediate nodes. A data packet data payload comprises one or more symbol representations. Each symbol representation in turn comprises one or more symbols that can be systematic, coded or recoded. The use of this symbol representation design is not limited by transmission schemes. It can be applied to unicast, multiple-unicast, multicast, multi-path, and multi-source settings and the like.

Coding coefficient vectors must be implicitly or explicitly transmitted from the sender to the receiver, generally along with the coded data for successful decoding of the original data. One option is to attach each coding coefficient vector to the corresponding coded symbol as a header, thus also enabling recoding at intermediate nodes. Another option is to attach the current state of a pseudo-random generator for generating the coding coefficient vector, to reduce the size of the header. Adding a header to each symbol may result in a high overhead when the symbol size is small or when generation or sliding window size is large. Adding a joint header to the beginning of each generation may also cause synchronization to be re-initiated only at the beginning of each generation instead of every symbol. In what follows, a symbol representation is provided that allow for both of these options such that both a general representation with coding coefficients and a compact representation with a seed for generating the coding coefficients can be used, in order to reduce the header overhead.

3.1. Representation Setup

This section specifies a symbol representation that enables both a general form with coding vectors attached, and a compact form where a seed is attached instead for the first symbol in the symbol representation, and where subsequent coding vectors are deduced from the first one. Different maximum generation and window size are supported for RLNC encoding, recoding, and decoding.

To encode over a set of data symbols, a coding vector as described in Section 1.1 is first generated, comprising a number of finite field elements as specified by a generation/window size variable. In the case of systematic codes, systematic symbols correspond to unit coding vectors.

Figure 4 illustrates the general symbol representation design. Four header fields precede the symbol data: TYPE flag, SYMBOLS, ENCODER RANK, and SEED or CODING COEFFICIENTS. The TYPE Flag indicates if the symbol is systematic, coded, or recoded. SYMBOLS indicates the number of symbols in the "Symbol(s) Data" field. ENCODER RANK represents the current rank of the encoder, which is the number of symbols being linearly combined. SEED is used to generate the coding coefficient vector(s) using a pseudo-random number generator, for a compact form of the symbol representation. CODING COEFFICIENTS field is a list of SYMBOLS number of coding vectors used to generate the ensuing SYMBOL(S) DATA.

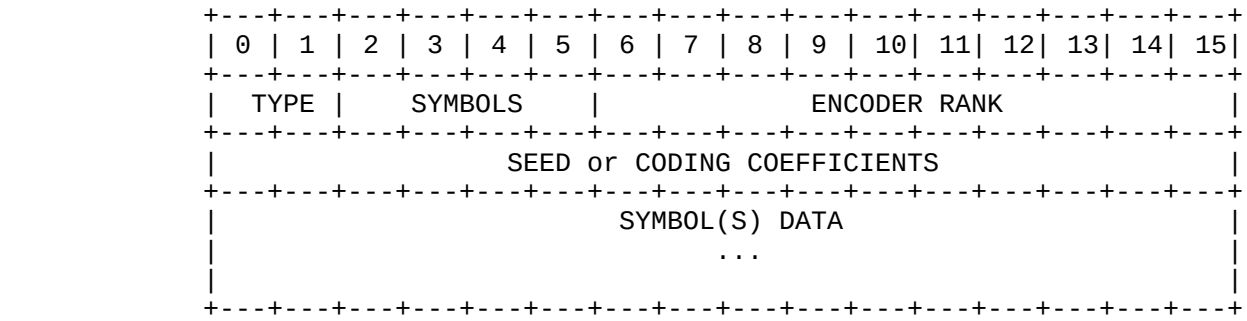

Figure 4: A general symbol representation design.

3.2. Field Types and Formats

The TYPE Flag indicates if the symbol is systematic, coded, or recoded, and has the following properties:

- 2 bits long.

- If the TYPE flag is ’1’, all symbols included in this symbol representation are systematic or uncoded, with symbol index starting from ENCODER RANK. This option allows for efficient representation of systematic symbols.

- If the TYPE is ’2’, all symbols included in this symbol representation are coded, with coding vectors generated using the included SEED and the ENCODER RANK. This option allows for compact and efficient representation of coded symbols, which may also subsequently be recoded.

- If the TYPE is ’3’, all symbols included in this symbol representation are either coded or recoded, with the coding coefficients included constituting ENCODER RANK coefficients each.

SYMBOLS indicates the number of symbols in the "Symbol(s) Data" field, and has the following properties:

- 4 bits long. A maximum number of 15 symbols are concatenated within each symbol representation.

- The special case of SYMBOLS = 0 indicates that zero symbols are included, and consequently the size of ’Symbol(s) Data’ is 0 bytes. This can, for example, be used to implement a flush functionality or ensure that protocol operations do not stop in certain case for purely event-driven protocols.

ENCODER RANK represents the current rank of the encoder, and has the following properties:

- MUST be no larger than generation/window size.

- If TYPE flag is ’1’, ENCODER RANK is the symbol index of the first data symbol in this symbol representation.

- If TYPE flag is ’2’ or ’3’, ENCODER RANK is the number of data symbols over which coding was performed for all coded symbols in this symbol representation.

- Coded symbols can be generated before a generation or window is filled. ENCODER RANK describes the number of original symbols included in the coded symbol(s).

SEED is used to generate the coding coefficient vector(s) using a pseudo-random number generator, for a compact form of the symbol representation, and has the following properties:

- The SEED field is only present when TYPE flag is '2'. If TYPE is '1' or '3', this field is absent.

- The pseudo-random generator MUST be seeded with this value and all coding coefficient vectors are produced by the same generator. For example, if ENCODER RANK is 12, then coding vector for the first symbol in this symbol representation is coefficients 0 through 11 generated by the pseudo-random generator seeded by SEED, and coding vector for the second symbol in this symbol representation is coefficients 12 through 23 generated by the pseudo-random generator seeded by SEED. If generation/window size is larger than ENCODER RANK, the remaining coefficients in the coding vector are zero.

- To ensure that SEED can be interpreted correctly at the receiver, the same pseudo-random number generator MUST be used by the sender and a recoding or receiving node. Otherwise, more than one SEED field would need to be used.

- 8 bits long. Thus, 256 different seed values can be served. One SEED is used per symbol representation, each of which can contain up to 15 symbols, all derived using the same SEED. For distinct ENCODER RANKs, different coding vectors would be generated from the same SEED, since only an ENCODER RANK number of coefficients from the random generator is grouped as a coding coefficient vector, before progressing to the next coding vector for the next symbol in the symbol representation. Consequently, the maximal number of coded symbols that can be generated for a generation is |SEED| * |SYMBOLS| * |ENCODER RANK| which in the best case is (2^8)*(2^4-1)*(2^10) ~ 2^22, which for all practical considerations can be considered as an infinite number of coded symbols. If all coded symbols that can be represented using a SEED is exhausted, symbols where the coding vectors is included can be sent instead.

- In the case where no random number generator is available, or where its use is not desired, the coding coefficients can be produced by other means, such as functions of the data, state of the network, or the like, and transmitted explicitly by setting the TYPE flag to '3'.

CODING COEFFICIENTS field is a list of SYMBOLS number of coding vectors used to generate the ensuing SYMBOL(S) DATA, and has the following properties:

- The CODING COEFFICIENT field is only present when TYPE flag is '3'. If TYPE is '1' or '2', this field is absent.

- Each coding vector comprises ENCODER RANK number of coding coefficients, each coding coefficient having a predetermined field size.

3.3. Small Encoding Window

In a first small encoding window symbol representation, ENCODER RANK is 10 bits long, and the maximum generation/window size is 2^10.

Figures 5 to 7 below illustrate systematic, coded, and recoded symbol representations within an encoding window of size 2^10. Systematic symbols are uncoded. Coded symbols are compact in form and comprises a seed for coding coefficient generation. Recoded symbols are general in form and comprises the coding coefficient vectors explicitly.

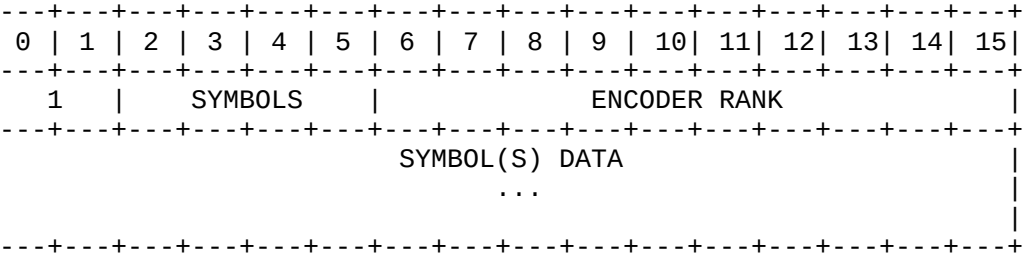

Figure 5: A systematic symbol representation.

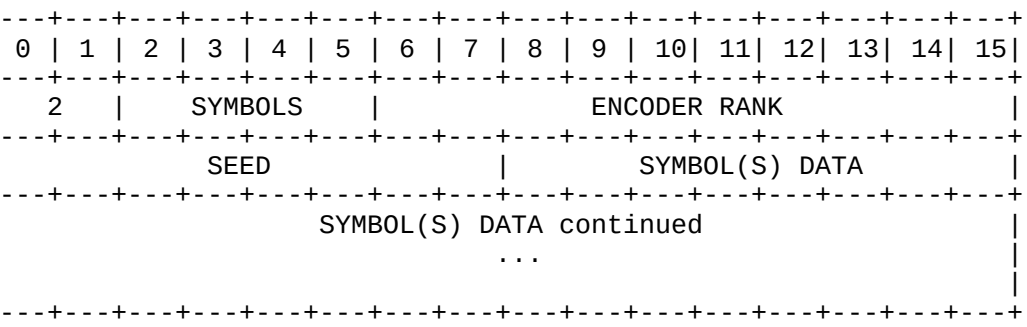

Figure 6: A compact, coded symbol representation.

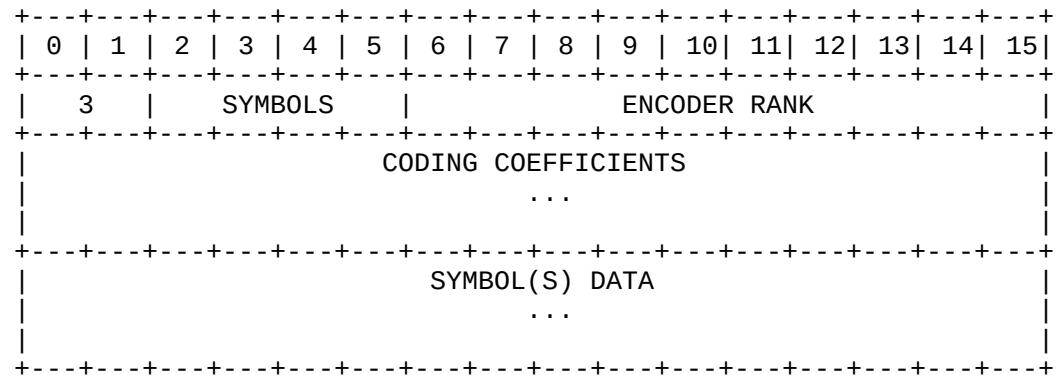

Figure 7: A recoded symbol representation.

3.3.1. Examples

The following examples show different symbol representations for an illustrative case where the symbol size is 2 bytes, generation/window size is 8, and field size is 2^8.

Example 1: Three systematic symbols with ID 0, 1 and 2. As the TYPE flag is '1' , SEED/CODING COEFFICIENTS is absent, and ENCODER RANK is the symbol index of the first data symbol with ID 0 in this compact symbol representation.

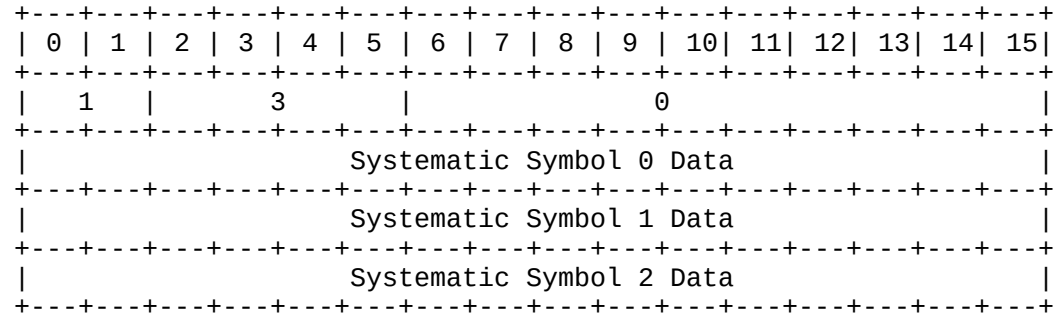

Figure 8: A symbol representation with 3 systematic, uncoded symbols.

Example 2: Two coded symbols using a compact representation. In this example, TYPE is '2', the SEED to the pseudo-random number generator shared by the sender and receiver is 4. The coding vector for Symbol A is coefficients 0 to 7 generated by the pseudo-random number generator, the coding vector for symbol B is coefficients 8 to 15 generated by the pseudo-random number generator.

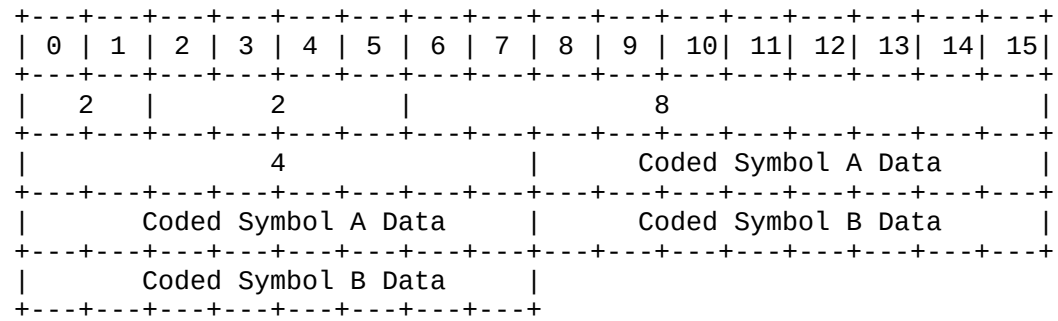

Figure 9: A symbol representation with 2 coded symbols.

Example 3: Two recoded symbols. Coefficients A0 to A7 constitutes the coding vector for Symbol A, coefficients B0 to B7 constitutes the coding vector for symbol B . In practical implementations, symbols sizes are much larger than 2, leading to amortization of the coding coefficient overheads.

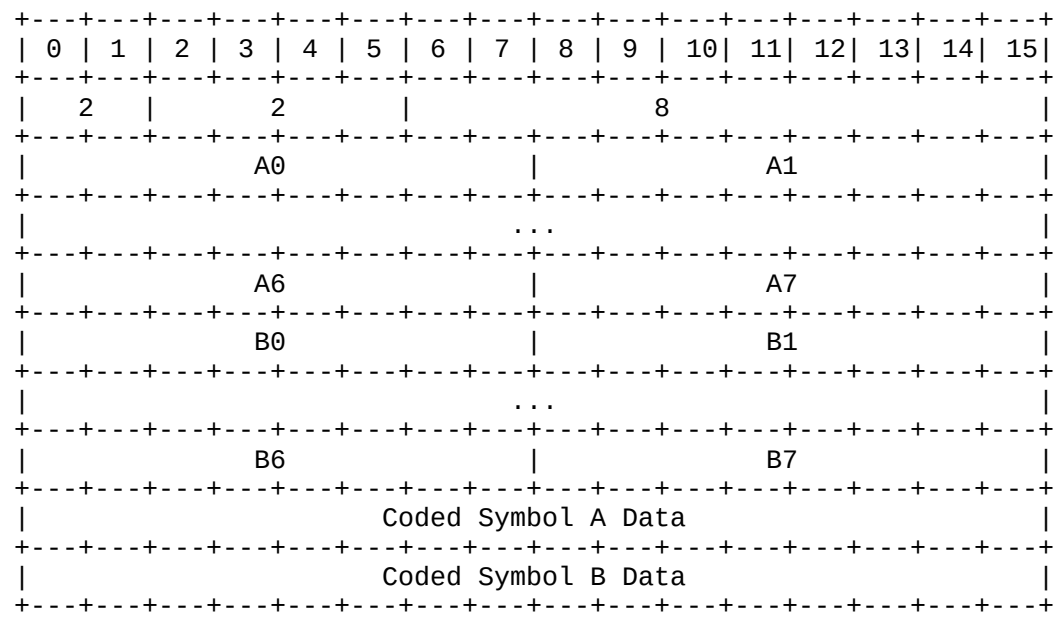

Figure 10: A symbol representation with 2 recoded symbols having coding coefficients attached.

3.4. Large Encoding Window

In a second large encoding window symbol representation, ENCODER RANK is 18-bit long, and the maximum generation/window size is 2^18.

Figures 11 to 13 below illustrate systematic, coded, and recoded symbol representations within an encoding window of size 2^18. Systematic symbols are uncoded. Coded symbols are compact in form and comprises a seed for coding coefficient generation. Recoded symbols are general in form and comprises the coding coefficient vectors explicitly.

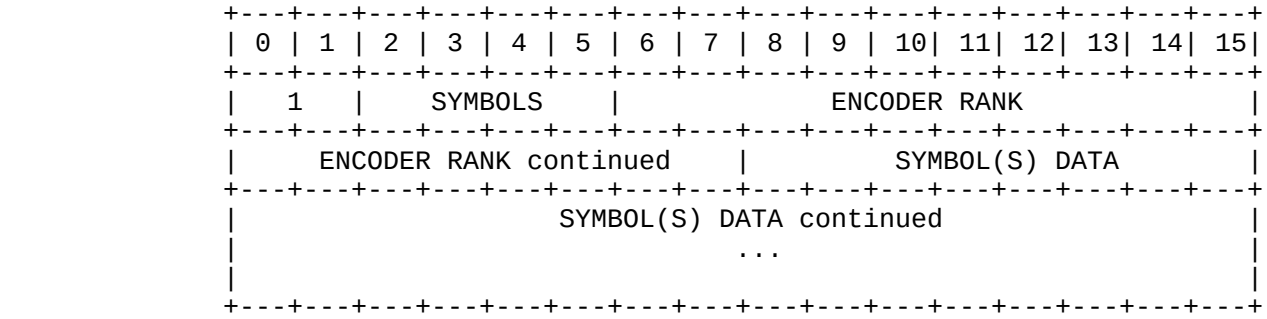

Figure 11: A systematic symbol representation.

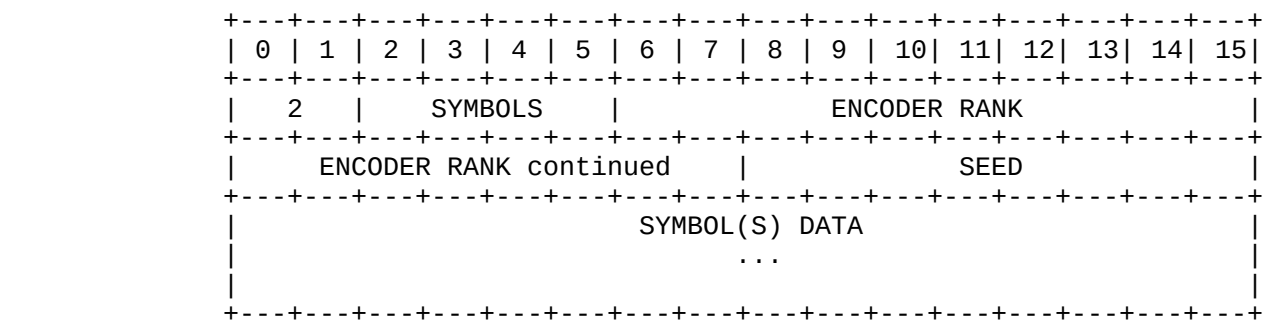

Figure 12: A coded symbol representation.

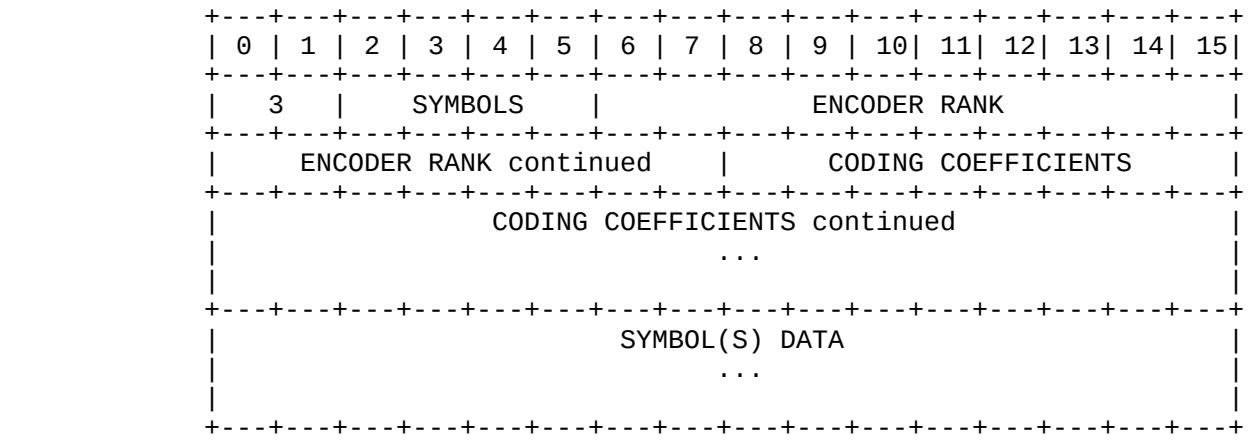

Figure 13: A recoded symbol representation.

4. Security Considerations

This document does not present new security considerations.

5. IANA Considerations

This document has no actions for IANA.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3453] Luby, M., Vicisano, L., Gemmell, J., Rizzo, L., Handley, M., and J. Crowcroft, "The Use of Forward Error Correction (FEC) in Reliable Multicast", RFC 3453, DOI 10.17487/RFC3453, December 2002, <https://www.rfc-editor.org/info/rfc3453>.

[RFC5445] Watson, M., "Basic Forward Error Correction (FEC) Schemes", RFC 5445, DOI 10.17487/RFC5445, March 2009, <https://www.rfc-editor.org/info/rfc5445>.

6.2. Informative References

[HK03] Ho, T., Koetter, R., Medard, M., Karger, D., and M. Effros, "The Benefits of Coding over Routing in a Randomized Setting", July 2003, <http://ieeexplore.ieee.org/document/1228459/>.

[HL08] Ho, T. and D. Lun, "Network Coding: An Introduction", April 2008.

[MS11] Medard, M. and A. Sprintson, "Network Coding: Fundamentals and Applications", October 2011.

[SS09] Sundararajan, J., Shah, D., Medard, M., Mitzenmacher, M., and J. Barros, "Network Coding Meets TCP", April 2009, <http://ieeexplore.ieee.org/document/5061931/>.

Authors' Addresses

Janus Heide
Steinwurf Aps
Aalborg
Denmark

Email: janus@steinwurf.com

Shirley Shi
Code On Network Coding LLC
Cambridge
USA

Email: xshi@alum.mit.edu

Muriel Medard
Code On Network Coding LLC
Cambridge
USA

Email: muriel.medard@codeontechnologies.com

Vince Chook
Inmarsat PLC
London
United Kingdom

Email: Vince.Chook@inmarsat.com

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: August 29, 2018

N. Kuhn, Ed.
CNES
E. Lochin, Ed.
ISAE-SUPAERO

February 25, 2018

Network coding and satellites
draft-kuhn-nwcrg-network-coding-satellites-02

Abstract

This memo presents the current deployment of network coding in some satellite telecommunications systems along with a discussion on the multiple opportunities to introduce these techniques at a wider scale.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 29, 2018.

Table of Contents

1. Introduction

Network coding schemes are inherent part of the satellite systems as the physical layer requires specific robustness to guarantee an efficient usage of the expensive radio resource. Further exploiting these schemes is an opportunity for a better end-user experience along with a better exploitation of the scarce resource.

In this context, this memo aims at:

- summing up the current deployment of network coding schemes over LEO and GEO satellite systems;
- identifying opportunities for further usage of network coding in these systems.

1.1. Glossary

The glossary of this memo is related to the network coding taxonomy document [I-D.irtf-nwcrg-network-coding-taxonomy].

The glossary is extended as follows:

- BBFRAME: Base-Band FRAME;
- PLFRAME: Physical Layer FRAME;
- PEP: Performance Enhanced Proxy;
- SATCOM: SATellite COMmunications;
- EPC: Evolved Packet Core;
- UMTRAN: Universal Mobile Terrestrial Radio Access Network;
- QoS: Quality-of-Service;
- QoE: Quality-of-Experience;
- VNF: Virtualized Network Function;
- CPE: Customer Premise Equipment;
- DTN: Delay Tolerant Network.

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. A note on satellite topology

The objective of this section is to provide both a generic description of the components composing a generic satellite system and their interaction. It provides a high level description of a multi-gateway satellites network. There exist multiple SATCOM systems, such as those dedicated to broadcasting TV or to IoT applications: depending on the purpose of the SATCOM system, ground segments are specific. This memo lays on SATCOM systems dedicated to Internet access that follows the DVB-S2/RCS2 standards.

In this context, Figure 1 shows an example of a multigateway satellite system. More details on a generic SATCOM ground segment architecture for a bi-directional Internet access can be found in [SAT2017]. This architecture may be mapped to cellular networks as follows: the ’network function’ block gather some of the functions of the Evolved Packet Core subsystem, while the ’access gateway’ and ’physical gateway’ blocks gather the same type of functions as the Universal Mobile Terrestrial Radio Access Network. This mapping

extends the opportunities identified in this draft since they may be also relevant for cellular networks.

It is worth noting that some functional blocks aggregate the traffic coming from multiple users, allowing the deployment of network coding schemes.

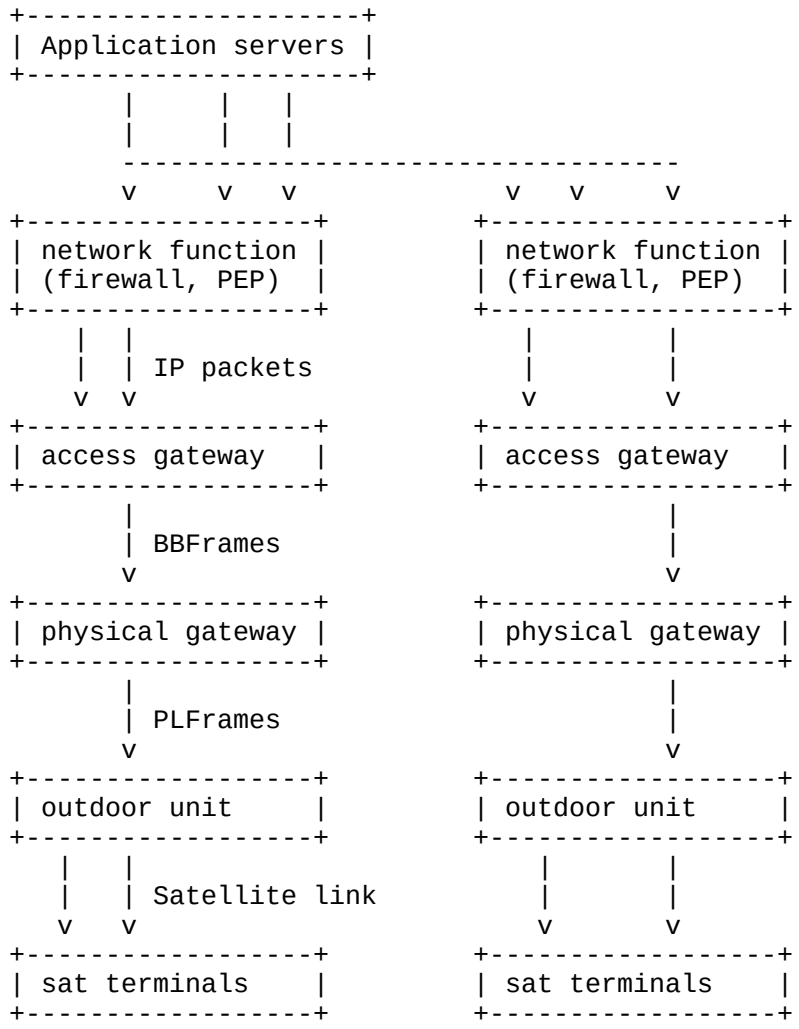

Figure 1: Data plane functions in a generic satellite multi-gateway system

3. Status of network coding in actually deployed satellite systems

Figure 2 presents the status of the network coding deployment in satellite systems. The information is based on the taxonomy document [I-D.irtf-nwcrg-network-coding-taxonomy] and the notations are the following: End-to-End Coding (E2E), Network Coding (NC), Intra-Flow Coding (IntraF), Inter-Flow Coding (InterF), Single-Path Coding (SP) and Multi-Path Coding (MP).

X1 embodies the source coding that could be used at application level for instance: for video streaming on a broadband access. X2 embodies the physical layer, applied to the PLFRAME, to optimize the satellite capacity usage. Furthermore, at the physical layer and when random accesses are exploited, FEC mechanisms are exploited.

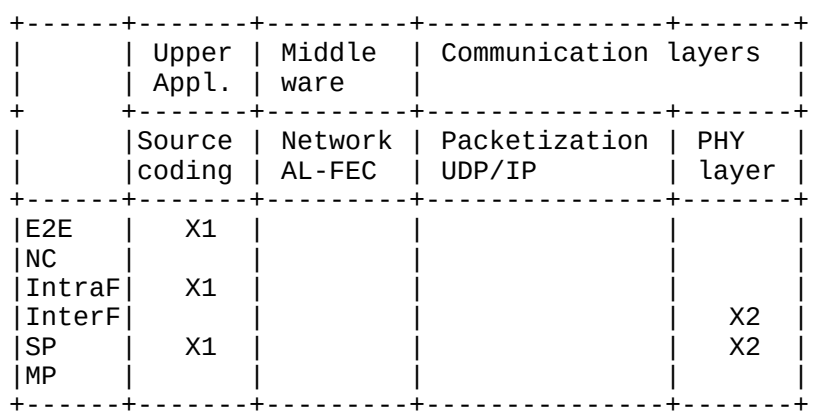

Figure 2: Network coding and satellite systems

4. Opportunities for more network coding in satellite systems

This section extends Section 3 by presenting potential opportunities for the deployment of network coding schemes inside satellite systems.

These opportunities are further detailed in Section 5 and listed in this section:

1. Two ways relay channel mode (more details in Section 5.1);

2. Reliable multi-cast (more details in Section 5.2);

3. Hybrid access (more details in Section 5.3);

4. Delay Tolerant Network architecture (more details in Section 5.4);

5. Dealing with varying capacity (more details in Section 5.5);

It is worth noting that these opportunities focus more on the middle ware (e.g. aggregation nodes) and packetization UDP/IP of Figure 2. Indeed, there are already lots of recovery mechanisms at the physical and access layers in currently deployed systems while E2E source coding are done at the application level. In a multigateway SATCOM Internet access, the specific opportunities are more relevant in specific SATCOM components such as the "network function" block or the "access gateway" of Figure 1.

5. Details on the use cases

This section details use-cases where network coding schemes could improve the overall performance of a SATCOM system (e.g. considering a more efficient usage of the satellite resource, delivery delay, delivery ratio).

5.1. Two way relay channel mode

This use-case considers a two-way communication between end users, through a satellite link. We propose an illustration of this scenario in Figure 3.

Satellite terminal A (resp. B) transmits a flow A (resp. B) to a server hosting NC capabilities, which forward a combination of the two flows to both terminals. This results in non-negligible bandwidth saving and has been demonstrated at ASMS 2010 in Cagliari.

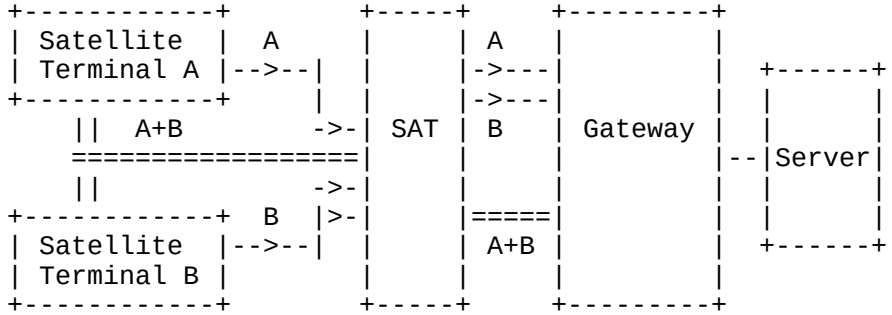

Figure 3: Network architecture for two way relay channel with NC

5.2. Reliable multi-cast

This use-case considers adding redundancy to a multi-cast flow depending on what has been received by different end-users, resulting in non-negligible scarce resource saving. We propose an illustration for this scenario in Figure 4.

A multi-cast flow (M) is forward to both satellite terminals A and B. On the uplink, terminal A (resp. B) does not acknowledge the packet Ni (resp. Nj) and either the access gateway or the multi-cast server includes the missing packets in the multi-cast flow so that the information transfer is reliable.

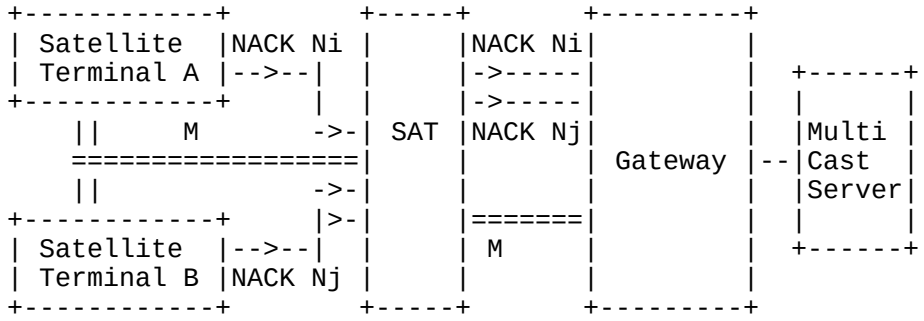

Figure 4: Network architecture for a reliable multi-cast with NC

5.3. Hybrid access

This use-case considers the use of multiple path management with network coding at the transport level to either increase the reliability or the total bandwidth. We propose an illustration for this scenario in Figure 5. This use-case is inspired from the Broadband Access via Integrated Terrestrial Satellite Systems (BATS) project. The architecture is also discussed in the MPTCP working group [I-D.boucadair-mptcp-dhc].

To cope from packet loss (due to either end-user movements or physical layer impairments), network coding could be introduced in both the CPE and at the concentrator.

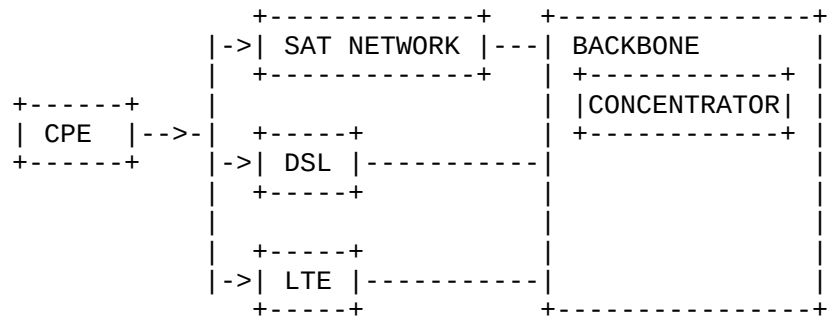

Figure 5: Network architecture for an hybrid access using NC

5.4. Delay Tolerant Network architecture

** EL: ** TBD with bundle layer as a candidate for NC

5.5. Dealing with varying capacity

This use-case considers the usage of network coding to overcome cases where the wireless link characteristics quickly change overtime and where the physical layer codes could not be made robust in time. The network coding schemes could be applied at the access gateway or the network function block levels to increase the reliability of the transmission. This use-case is relevant for, e.g. when mobile users are considered or when the chosen band induce a required physical layer coding that may change over time (Q/V bands, Ka band, etc.).

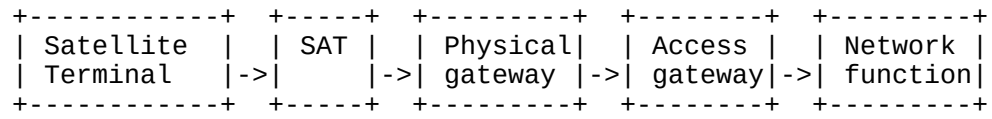

Figure 6: Network architecture for dealing with varying link characteristics with NC

6. Discussion on the deployability

This section discusses the deployability of the opportunities that are provided in Section 4.

SATCOM systems typically feature Proxy-Enhanced-Proxy RFC 3135 [RFC3135] which could be relevant to host network coding mechanisms and thus support the use-cases that have been discussed in Section 5. In particular the discussion on how network coding can be integrated inside a PEP with QoS scheduler has been proposed in RFC 5865 [RFC5865].

Related to the foreseen virtualized network infrastructure, the network coding schemes could be proposed as VNF and their deployability enhanced. The architecture for the next generation of SATCOM ground segments would rely on a virtualized environment. This trend can also be seen, making the discussions on the deployability of network coding in SATCOM extendable to other deployment scenarios [I-D.chin-nfvrg-cloud-5g-core-structure-yang]. As one example, the network coding VNF functions deployment in a virtualized environment is presented in [I-D.vazquez-nfvrg-netcod-function-virtualization].

7. Acknowledgements

8. Contributors

Many thanks to

9. IANA Considerations

This memo includes no request to IANA.

10. Security Considerations

This document, by itself, presents no new privacy nor security issues.

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3135] Border, J., Kojo, M., Griner, J., Montenegro, G., and Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations", RFC 3135, DOI 10.17487/RFC3135, June 2001, <https://www.rfc-editor.org/info/rfc3135>.

[RFC5865] Baker, F., Polk, J., and M. Dolly, "A Differentiated Services Code Point (DSCP) for Capacity-Admitted Traffic", RFC 5865, DOI 10.17487/RFC5865, May 2010, <https://www.rfc-editor.org/info/rfc5865>.

11.2. Informative References

[I-D.boucadair-mptcp-dhc]
Boucadair, M., Jacquenet, C., and T. Reddy, "DHCP Options for Network-Assisted Multipath TCP (MPTCP)", draft-boucadair-mptcp-dhc-08 (work in progress), October 2017.

[I-D.chin-nfvrg-cloud-5g-core-structure-yang]
Chen, C. and Z. Pan, "Yang Data Model for Cloud Native 5G Core structure", draft-chin-nfvrg-cloud-5g-core-structure-yang-00 (work in progress), December 2017.

[I-D.irtf-nwcrg-network-coding-taxonomy]
Adamson, B., Adjih, C., Bilbao, J., Firoiu, V., Fitzek, F., samah.ghanem@gmail.com, s., Lochin, E., Masucci, A., Montpetit, M., Pedersen, M., Peralta, G., Roca, V., Saxena, P., and S. Sivakumar, "Taxonomy of Coding Techniques for Efficient Network Communications", draft-irtf-nwcrg-network-coding-taxonomy-07 (work in progress), February 2018.

[I-D.vazquez-nfvrg-netcod-function-virtualization]
Vazquez-Castro, M., Do-Duy, T., Romano, S., and A. Tulino, "Network Coding Function Virtualization", draft-vazquez-nfvrg-netcod-function-virtualization-02 (work in progress), November 2017.

[SAT2017] Ahmed, T., Dubois, E., Dupe, JB., Ferrus, R., Gelard, P., and N. Kuhn, "Software-defined satellite cloud RAN", Int. J. Satell. Commun. Network. vol. 36, 2017.

Authors' Addresses

Nicolas Kuhn (editor)
CNES
18 Avenue Edouard Belin
Toulouse 31400
France

Email: nicolas.kuhn@cnes.fr

Emmanuel Lochin (editor)
ISAE-SUPAERO
10 Avenue Edouard Belin
Toulouse 31400
France

Email: emmanuel.lochin@isae-supaero.fr

Network Coding Research Group
Internet-Draft
Intended status: Informational
Expires: September 6, 2018

K. Matsuzono
H. Asaeda
NICT
C. Westphal
Huawei

March 5, 2018

Network Coding for Content-Centric Networking / Named Data Networking: Requirements and Challenges
draft-matsuzono-nwcrg-nwc-ccn-reqs-01

Abstract

This document describes the current research outcomes regarding Network Coding (NC) for Content-Centric Networking (CCN) / Named Data Networking (NDN), and clarifies the requirements and challenges for applying NC into CCN/NDN.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 6, 2018.

Table of Contents

1. Introduction

Information-Centric Networks in general, and Content-Centric Networking (CCN) [15] or Named Data Networking (NDN) [16] in particular, have emerged as a novel communication paradigm advocating to retrieve data through their names. This paradigm pushes content awareness into the network layer. It is expected to enable consumers to obtain the content they desire in a straightforward and efficient manner from the heterogenous networks they may be connected to. The CCN/NDN architecture has introduced innovative ideas and has stimulated research in a variety of areas, such as in-network caching, name-based routing, multi-path transport, content security, and so on. One key benefit of requesting content by name is that it removes the need to establish a session between the client and a specific server, and that content can thereby be retrieved from multiple sources.

In parallel, there has been a growing interest from both academia and industry to better understand fundamental aspects of Network Coding (NC) toward enhancing key system performance metrics such as data throughput, robustness and reduction in the required number of transmissions through connected networks, point-to-multipoint connections, etc. Typically, NC is a technique mainly used to encode packets to recover lost source packets at the receiver, and to effectively get the desired information in a fully distributed manner. In addition, NC can be used for security enhancements [2][3][4][5].

NC aggregates multiple packets with parts of the same content together, and may do this at the source or at other nodes in the network. As such, network coded packets are not connected to a specific server, as they may have evolved within the network. Since NC focuses on what information should be encoded in a network packet, rather than the specific host where it has been generated, it is in line with the CCN/NDN core networking layer (described in more detail later on). NC has already been implemented for information/content dissemination (e.g. [6][7][8]). NC provides CCN/NDN with the highly beneficial potential to effectively disseminate information in a completely independent and decentralized manner. [9] first suggested to exploit NC techniques to enhance key system performances in ICN, and others have considered NC in ICN use cases such as content dissemination [10], seamless mobility [11], joint caching and network coding [12][13], low-latency video streaming [14], etc.

In this document, we consider how NC can be applied to the CCN/NDN architecture and describe the requirements and potential challenges for making CCN/NDN-based communications better using the NC technology. Please note that providing specific solutions (e.g., NC optimization methods) to enhance CCN/NDN performance metrics by exploiting NC is out of scope of this document.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

2.1. Definitions

The terminology regarding NC used in this document is described below. It is aligned with RFCs produced by the FEC Framework (FECFRAME) IETF Working Groups as well as recent activities in the Network Coding Research Group [18].

- Random Linear Coding (RLC): Particular case of Linear Coding using a set of random coding coefficients.

- Generation, or (IETF) Block: With Block Codes, the set of content data that are logically grouped into a Block, before doing encoding.

- Generation Size: With Block Codes, the number k of content data belonging to a Block.

- Encoding Vector: A set of coding coefficients used to generate a certain coded packet through linear coding. The number of nonzero coefficients in the Coding Vector defines its density

- Finite Field: Finite fields, used in Linear Codes, have the desired property of having all elements (except zero) invertible for + and * and all operations over any elements do not result in an overflow or underflow. Examples of Finite Fields are prime fields {0..p^m-1}, where p is prime. Most used fields use p=2 and are called binary extension fields {0..2^m-1}, where m often equals 1, 4 or 8 for practical reasons.

- Finite Field size: The number of elements in a finite field. For example the binary extension field {0..2^m-1} has size q=2^m.

- Block Coding: Coding technique where the input Flow(s) must be first segmented into a sequence of blocks, FEC encoding and decoding being performed independently on a per-block basis.

- Sliding Window Coding or Convolutional Coding: General class of coding techniques that rely on a sliding encoding window. This is an alternative solution to Block Coding.

- Fixed or Elastic Sliding Window Coding: Coding technique that generates repair data on-the-fly, from the set of source data present in the sliding encoding window at that time, usually by using Linear Coding. The sliding window may be either of fixed size or of variable size over the time (also known as "elastic sliding window").

- Feedback: Feedback information sent by a decoding node to a node (or from a consumer to a publisher in case of End-to-End Coding). The nature of information contained in a feedback packet varies, depending on the use-case. It can provide reception and/or decoding statistics, or the list of available source packets received or decoded, or the list of lost source packets that should be retransmitted, or a number of additional repair packet needed to have a full rank linear system.

Concerning CCN/NDN, the following terminology and definitions are used.

- Consumer: A node requesting content. It initiates communication by sending an interest packets.
- Publisher: A node providing content. It originally creates or owns the content.
- Forwarding Information Base (FIB): A lookup table in a content router containing the name prefix and corresponding destination interface to forward the interest packets.
- Pending Interest Table (PIT): A lookup table populated by the interest packets containing the name prefix of the requested data, and the outgoing interface used to forward the received data packets.
- Content Store (CS): A storage space for a router to cache content objects. It is also known as in-network cache.
- Content Object: A unit of content data delivered through the CCN/NDN network.
- Content Flow: A sequence of content objects associated with the unique content name prefix.

2.2. NDN/CCN Background

Armed with the terminology above, we briefly explain the key concepts of CCN/NDN. Both protocols are similar in principle, and different on some implementation choices.

In a CCN network, there are two types of packets at the network level: interest and data. The consumer request a content by sending an "interest" message, that carries the name of the data. On difference to note here in CCN and NDN is that in later versions of CCN, the interest must carry a full name, while in NDN it may carry a name prefix (and receive in return any data with a name matching this prefix).

Once a router receives an "interest" message, it performs a series of look-up: first it checks in the Content Store if it has a copy of the requested content available. If it does, it returns the data and the transaction has successfully completed.

If it does not, it performs a look-up of the PIT to see if there is already an outgoing request for the same data. If there is not, then

it creates an entry in the PIT that lists the name included in the interest, and the interfaces from which it received the interest. This is used later to send the data back, since interest packets do not carry a source field that identifies the requester. If there is already a PIT entry for this name, then it is updated with the incoming interface of this new request and the interest is discarded.

After the PIT look-up, the interest undergoes a FIB lookup to select an outgoing interface. The FIB lists name prefixes and their corresponding forwarding interfaces, to send the interface towards a router that possesses a copy of the requested data.

Once a copy of the data is retrieved, it is send back to the requester(s) using the trail of PIT entries; intermediate node remove the PIT state every time that an interest is satisfied, and may store the data in their content store.

Data packets carry some information to validate the data, in particular that the data is indeed the one that corresponds to the name. This is required since authentication of the object is crucial in CCN/NDN. However, this step is optional at intermediate routers, so as to speed up the processing.

The key aspect of CCN/NDN is that the consumer of the content does not establish a session with a specific server. Indeed, the node that returns the content is not aware of the network location of the requester and the requester is not aware of the network location of the node that provides the content. This in theory allows the interests to follow different paths within a network, or even to be sent over totally different networks.

3. Advantage given by NC and CCN/NDN

Both NC for large scale content dissemination [7] and CCN/NDN can contribute to effective content/information delivery while working jointly. They both bring similar benefits such as throughput/ capacity gain and robustness enhancement. The difference between their approaches is that, the former considers content flow as algebraic information to combine [17], while the latter focuses on content/information itself at the networking layer. Because these approaches are complementary, it is natural to combine them. The CCN/NDN core abstraction at networking layer through name makes network stack simple as it enables applications to take maximum advantage of multiple simultaneous connectivities due to its simpler relationship with the layer 2 [15].

CCN/NDN itself, however, cannot provide reliable and robust content dissemination. This requires some specific CCN/NDN transport (i.e.,

strategy layer) [15]. NC can enable the CCN/NDN transport system to effectively distribute and cache data associated with multi-path data retrieval. Furthermore, NC may further enhance CCN/NDN security [23]. In this context, it should be natural that there is much room for considering NC integration into CCN/NDN transport exploiting in-network caching and multi-path transmission [9] and seamless mobility [11] [28].

From the perspective of NC transport mechanism, NC is divided into two major categories: one is coherent NC, and the other is non-coherent NC [30]. In coherent NC, source and destination nodes exactly know network topology and coding operations at intermediate nodes. When multiple consumers are trying to receive the same content such as live video streaming, coherent NC could enable the optimal throughput by making the content flow sent over the constructed optimal multicast trees [24].

However, it requires fully adjustable and specific name-based routing mechanism for CCN/NDN, and an intense computational task for central coordination. In the case of non-coherent NC that often utilizes RLC, they do not need to know network topology and intermediate coding operations. Since non-coherent NC works in a completely independent and decentralized manner, this approach is more feasible especially in the large scale use cases that are intended with CCN/NDN. This document thus focuses on non-coherent NC with RLC.

4. Requirements

This section presents the NC requirements for ICN/CCN in terms of network architecture and protocol. The current document focuses on NC in a block coding manner.

4.1. Content Naming

Naming content objects is as important for CCN/NDN as naming hosts is for today’s Internet [19]. Before performing network coding for specified content in CCN/NDN, the overall content should be split into small content objects to avoid packet fragmentation that could cause unnecessary packet processing and degrades throughput. The size of content objects should be within the allowable packet size so as to avoid packet fragmentation in CCN/NDN network, and then network coding should be applied into a set of the content objects.

Each coded packet MAY have a unique name as the original content object has in CCN/NDN, since PIT/FIB/CS operations need a unique name to identify the coded data. As a way of naming coded packet, the encoding vector and the identifier of generation can be used as a part of the content object name [10]. For instance, when the block

size (also called generation size) is k and the encoding vector is [1,0,0,0], the name would be like /CCN.com/video-A/k/1000. This naming scheme is simple and can support the delivery of coded packets with exactly the same operations in the FIB/PIT/CS as for original source packets. However, such a naming way requires the consumer to know the naming structure (through a specific name resolution scheme for instance) in order for nodes to specify the exact name of generated coded data packet to retrieve it. From this point of view, it could shift the generation of the encoding vector from the content producer onto the content requester.

If a naming schema such as above is used, it would be valuable to reconsider whether Interest should carry full names (as in CCN) or prefixes (as in NDN) as multiple network coded packets could match a response to a specific prefix for a given generation, such as /CCN.com/video-A/k. In the latter case allowing partial name matching, the content requestor may not be able to obtain degrees of freedom. Thus, extensions in the TLV header of the Interest would be used to specify further network coding information so as to limit coded packets to be received (for instance, by specifying the encoded vectors the content requestor receives (also called decoding matrix) as in [9]). However, it may incur a largely increased size of TLV header. Without such coding information, the forwarding node would need to maintain some records regarding interest packets sent before, in order to provide new degrees of freedom.

Coded packet MAY have a name that indicates that it is a coded packet, and move the coding information into a metadata field in the payload (i.e., the name includes only data type, original or coded packet, etc). This however would preclude network coding on packets without prior decoding them (for instance, in the CS of forwarding nodes). It would not be beneficial for applications or services that may not need to understand the packet payload. Due to the possibility that multiple coded packets may have a same name, as described above, some mechanism needs for the content requestor to obtain innovative coded packets. It would also require some mechanism to insert the multiple innovative packets into the CS. If the coding information of coded packet are encrypted together with the payload (for instance, at source coding), the content requestor or forwarding nodes would incur extra computational overhead for decryption of the packet to interpret the coding information.

4.2. Transport

The pull-based request-response feature of CCN/NDN is the fundamental principle of its transport layer; one Interest retrieves at most one Data packet. It is important to not violate this rule, as it would

open denial of service attacks issues, and thus the following basic operation should be considered to apply NC to CCN/NDN.

4.2.1. Scope of Network Coding

It should be discussed whether the network can update data packets that are being received in transit, or if only the data that matches an interest can be subject to network coding operations. In the latter case, the network coding is performed on an end-to-end basis (where one end is the consumer, and the other end is any node that is able to respond to the Interest). In the former case, NC happens anywhere in the network that is able to update the data. As CCN/NDN has mechanisms in place to ensure the integrity of the data during transfer, NC in the network introduce complexities that would require special consideration for the integrity mechanisms to still work.

Similarly, caching of network coded packets at intermediate node may be valuable, but may prevent the node caching the coded content to validate the content.

4.2.2. Consumer Operation

To attain NC benefits associated with in-network caching, consumers need to issue interests directing the router (or publisher) to forward innovative coded packets if available. The reason why this directive is needed is that delay-sensitive applications such as live-video streaming may want to sequentially get original packets rather than coded packets cached in routers due to real-time constraint. Issuing such an interest is possible by using optional TLV (Type Length Value) header contained in Interest TLV packet format which allows network elements to add or modify information on the fly. Consumer can put an instruction into it, and for instance, if routers detect that it is better for consumer to get coded packets rather than original packets, routers can modify it to do so. After receiving interests having the instruction in optional header, the router with useful coded packets forward them.

As another solution, consumer issues interests specifying unique names for each coded packets. In this case, a unified naming scheme considering both original and coded packets is required. Moreover, in the case of NC end-to-end approach, publishers need to get feedback from the corresponding receivers to adjust some coding parameters. To deal with this, a receiver may have to request a specific interest name to reach the corresponding publisher and put required information into the optional header.

4.2.3. Router Operation

Routers need to appropriately handle PIT entries to accommodate interests for coded packets as well as original packets. Moreover, in order to decode as necessary, nodes need to know the coding vector used for each coded packet (note: since all the data for a specific content may not come through the same path/network, intermediate nodes may never be able to decode). In a typical case, the coding vector used for each coded packet is attached to the header of coded data. In regard to this point, the generation size (also called block size) for NC should be set to a reasonable value so that the total coded packet size including header needed for expressing the coding vector information and data message fits into the allowable packet size. It may be useful to use compression techniques for coding vectors [20][21].

Router may try to forward useful independent coded packets toward downstream nodes in order to respond to received interests for coded packets. Routers thus need to determine whether or not they can generate useful coded packets for consumers. Assuming that the size of the Finite Field in use is not relatively small, re-encoding using enough cached packets has a strong probability of making independent coded packets [24]. If router does not have enough cached packets to newly produce independent coded packets, it relays received interests to upstream nodes to receive a new original or independent coded packet and pass it to downstream nodes. In another possible case, when receiving interests for only original packets, routers may try to decode and get all the original packets and store them (if there are fully available cache capacity), enabling faster response to the interests. Since there is a tradeoff between NC encoding/decoding calculation cost and cache capacity, and the usage efficacy of re-encoding or decoding at router, router should need to determine how to response to receiving interests according to the use case (e.g., delay-sensitive or delay-tolerant application) and the router situation such as available cache space and computational capability.

Some proposed schemes [10]require that the router maintain a tally of the interests for a specific name and generation, so as to know how many degrees of freedom have been provided already for the NC packets. Scalability and practicality of maintaining such scheme at intermediate routers should considered.

To enable fast loss recovery cooperating with in-network caching, a transport mechanism of in-network loss detection and recovery [28][14] at router as well as consumer-driven mechanism should be considered.

4.2.4. Publisher Operation

The procedure for splitting an overall content into small content objects is responsible for the original publisher. When applying NC for the content, the publisher performs NC over the content objects, and naming processing for the coded packets. If the producer takes the lead in determining the used encoding vectors and generating the coded packets, there are the two possible end-to-end cases; 1) content requestors obtain the names of coded packets through a certain mechanism, and send the correspond interests toward the publisher to get the coded packets already generated at the publisher, and 2) the publisher determines the encoding vectors after receiving interests specifying them. In the former case, although content requestors cannot flexibly specify an encoding vector for generating the coded packet to retain, but the latency for getting the coded data can be reduced compared to the latter case where additional NC operations need after receiving interests. According to application requirement for latency, such NC operation strategy should be considered.

4.3. In-network Caching

Caching is an essential technique to improve throughput and latency in various applications. In-network caching CCN/NDN essentially supports at network level is highly beneficial by exploiting NC to enable effective multicast transmission [29], multipath data retrieval[10] [11], fast loss recovery [14], and so on. However, there are several issues to be considered.

As a general issue, there are limitations of cache capacity, and caching policy affects on consumer’s performances [22] [25] [26]. It is thus highly significant for routers to determine which packets should be cached and discarded. Since delay-sensitive applications often do not require in-network cache for a long period due to their real-time constraints, routers have to know the necessity for caching received packets to save the caching volume. This could be possible by putting a flag into optional header of data packets at publisher side. When receiving data packets with the flag meaning no necessity for cache, routers just have to forward them to downstream nodes. On the other hand, when receiving original packets or coded packets without the flag, router may cache them based on a specified replacement policy.

One key aspect of in-network caching is whether or not intermediate nodes can cache NC packets without first decoding them. If in-network caches store coded packets, they need to be able to validate that the packets are not compromised, so as to avoid cache pollution attacks. Without having all the packets in a generation, the cache

cannot decode the packets to check if it is authenticated. Caching of coded packets would require some mechanism to validate coded packets. In addition, when coded packets have a same name, it would also require some mechanism to identify them.

4.4. Seamless Mobility

This subsection presents how NC can achieve seamless mobility [11] [28] and clarify the requirements. A key feature of CCN/NDN is that it is sessionless and that multiple interests can be send to different copies of the content in parallel. CCN/NDN enables a consumer to retrieve the content from multiple sources that are distributed and asynchronous.

In this context, network coding provide a mechanism to ensure that the Interests sent to multiple copies of the content retrieve innovative packets, even in the case of packet losses on some of the paths/networks to these copies. NC adds a reliability layer to CCN in a distributed and asynchronous manner. One key benefit is that the link between the consumer and the multiple copies acts as a virtual logical link, upon which rate adaptation mechanism can be performed.

This naturally applies to mobility event, where the consumer may connect between multiple access points before a mobility event (make-before-break handoff). In such mobility event, the consumer is connected first to the previous access point, then to both the previous and next access points, then finally only to the next access points. With CCN, the consumer only sends interests on the available interfaces. Requesting network coded packets ensures that during the phase where it is connected to the previous and the next APs at the same time, it does not receive duplicate data, but does not miss on any content either. By combining NC with CCN, the consumer receives additional degrees of freedom with any innovative packet it receives on either interface.

Further discussion is [TBD].

4.5. Security and Privacy

This subsection describes the requirement for security and privacy provided by NC in CCN/NDN, such as data integrity especially when intermediate nodes perform re-encoding, as in the case of hash restrictions for original data packets, and so on.

Network coding impacts the security mechanisms of CCN/NDN. In particular, CCN/NDN is designed to prevent modification of the Data packets. Because Data packets for a specific name can be self-

authenticated, they can be validated on the delivery path, and can also be cached at untrusted intermediate nodes. Network coding may bring up issues if intermediate nodes are allowed to modify packets by performing additional network coding operations. Intermediate nodes may also be caching network coded packets without having the ability to perform validation of the content and therefore open themselves to cache pollution attacks.

In CCN/NDN, content objects can be encrypted to support access control or privacy. If the coding information of coded packet is included in the encrypted data payload, extra computational overhead occurs.

5. Challenges

This section presents several primary challenges and research items to be considered when applying NC into CCN/NDN.

5.1. Adopting Convolutional Coding

Several block coding approaches have been proposed so far, but there is still no sufficient discussion and application of convolutional coding approach (e.g., sliding or elastic window coding) in CCN/NDN. Convolutional coding is often appropriate to situations where a fully or partially reliable delivery of continuous data flows is needed, especially when these data flows feature realtime constraints. As in [31] on an end-to-end basis, it would be advantageous for continuous content flow to adopt sliding window coding in CCN/NDN. In this case, the publisher needs to appropriately set coding parameters and let content requestor know the information, and content requestor needs to send interest (i.e., feedback information) about the data reception status. Since CCN/NDN advocates hop-by-hop communication, it would be worth discussing and investigating how convolutional coding can be applied in a hop-by-hop fashion and the benefits. In particular, assuming that NC could occur at intermediate nodes with some useful data packets stored in the CS as described in the previous section, both the encoding window and CS management would be required, and the feasibility and practicality should be considered.

5.2. Rate and Congestion Control

Adding redundancy using coded packets may cause further network congestion and adversely affect overall throughput performance. In particular, in a situation where fair bandwidth sharing is more desirable, each streaming flow must adapt to the network conditions to fairly consume the available link bandwidth. It is thus indispensable that each content flow cooperatively implements congestion control to adjust the consumed bandwidth to stabilize the

network condition (i.e., to achieve low packet loss rate, delay, and jitter).

5.3. Security and Privacy

A variety of security and privacy concerns would exist in NC and CCN/NDN. This subsection focuses on the description of security and privacy challenges related to NC for CCN/NDN. [TBD]

5.4. Routing Scalability

This subsection focuses on the challenges of routing mechanisms such as scalability and protocol overhead, and so on.

6. Security Considerations

This document does not impact the security of the Internet. Security considerations related to NC for CCN/NDN are described in the previous Section.

7. References

7.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

7.2. Informative References

[2] Cai, N. and R. Yeung, "Secure network coding", Proc. International Symposium on Information Theory (ISIT), IEEE, June 2002.

[3] Lima, L., Gheorghiu, S., Barros, J., Mdard, M., and A. Toledo, "Secure Network Coding for Multi-Resolution Wireless Video Streaming", IEEE Journal of Selected Area (JSAC), vol. 28, no. 3, April 2002.

[4] Gkantsidis, C. and P. Rodriguez, "Cooperative Security for Network Coding File Distribution", Proc. Infocom, IEEE, April 2006.

[5] Vilea, J., Lima, L., and J. Barros, "Lightweight security for network coding", Proc. ICC, IEEE, May 2008.

[6] Dimarkis, A., Godfrey, P., Wu, Y., Wainwright, M., and K. Ramchandran, "Network Coding for Distributed Storage Systems", IEEE Trans. Information Theory, vol. 56, no.9, September 2010.

[7] Gkantsidis, C. and P. Rodriguez, "Network coding for large scale content distribution", Proc. Infocom, IEEE, March 2005.

[8] Seferoglu, H. and A. Markopoulou, "Opportunistic Network Coding for Video Streaming over Wireless", Proc. Packet Video Workshop (PV), IEEE, November 2007.

[9] Montpetit, M., Westphal, C., and D. Trossen, "Network Coding Meets Information-Centric Networking: An Architectural Case for Information Dispersion Through Native Network Coding", Proc. Workshop on Emerging Name-Oriented Mobile Networking Design (NoM), ACM, June 2012.

[10] Saltarin, J., Bourtsoulatze, E., Thomos, N., and T. Braun, "NetCodCCN: a network coding approach for content-centric networks", Proc. Infocom, IEEE, April 2016.

[11] Ramakrishnan, A., Westphal, C., and J. Saltarin, "Adaptive Video Streaming over CCN with Network Coding for Seamless Mobility", Proc. International Symposium on Multimedia (ISM), IEEE, December 2016.

[12] Wang, J., Ren, J., Lu, K., Wang, J., Liu, S., and C. Westphal, "An Optimal Cache Management Framework for Information-Centric Networks with Network Coding", Proc. Networking Conference, IFIP/IEEE, June 2014.

[13] Wang, J., Ren, J., Lu, K., Wang, J., Liu, S., and C. Westphal, "A Minimum Cost Cache Management Framework for Information-Centric Networks with Network Coding", Computer Networks, Elsevier, August 2016.

[14] Matsuzono, K., Asaeda, H., and T. Turletti, "Low Latency Low Loss Streaming using In-Network Coding and Caching", Proc. Infocom, IEEE, May 2017.

[15] Jacobson, V., Smetters, D., Thornton, J., Plass, M., Briggs, N., and R. Braynard, "Networking Named Content", Proc. CoNEXT, ACM, December 2009.

[16] Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., Claffy, K., Crowley, P., Papadopoulos, C., Wang, L., and B. Zhang, "Named data networking", ACM Comput. Commun. Rev., vol. 44, no. 3, July 2014.

[17] Koetter, R. and M. Medard, "An Algebraic Approach to Network Coding", IEEE/ACM Trans. on Networking, vol. 11, no 5, Oct. 2003.

[18] Adamson, B., Adjih, C., Bilbao, J., Firoiu, V., Fitzek, F., Lochin, E., Masucci, A., Montpetit, M., Pedersen, M., Peralta, G., Roca, V., Saxena, P., and S. Sivakumar, "Network Coding Taxonomy", draft-irtf-nwcrg-network-coding-taxonomy-05 (work in progress), September 2017.

[19] Kutscher, et al., D., "Information-Centric Networking (ICN) Research Challenges", RFC 7927, July 2016.

[20] Thomos, N. and P. Frossard, "Toward one Symbol Network Coding Vectors", IEEE Communications letters, vol. 16, no. 11, November 2012.

[21] Lucani, D., Pedersen, M., Heide, J., and F. Fitzek, "Fulcrum Network Codes: A Code for Fluid Allocation of Complexity", available at http://arxiv.org/abs/1404.6620, April 2014.

[22] Perino, D. and M. Varvello, "A reality check for content centric networking", Proc. SIGCOMM Workshop on Information-centric networking (ICN'11), ACM, August 2011.

[23] Wu, Q., Li, Z., Tyson, G., Uhlig, S., Kaafar, M., and G. Xie, "Privacy-Aware Multipath Video Caching for Content-Centric Networks", IEEE Journal of Selected Area (JSAC) vol. 38, no. 8, June 2016.

[24] Wu, Y., Chou, P., and K. Jain, "A comparison of network coding and tree packing", Proc. ISIT, IEEE, June 2004.

[25] Podlipnig, S. and L. Osz, "A Survey of Web Cache Replacement Strategies", Proc. ACM Computing Surveys vol. 35, no. 4, December 2003.

[26] Rossini, G. and D. Rossi, "Evaluating CCN multi-path interest forwarding strategies", Elsevier Computer Communication, vol.36, no. 7, April 2013.

[27] Chai, W., He, D., Psaras, I., and G. Pavlou, "Cache Less for More in Information-centric Networks", Journal Computer Communications, vol. 37. no. 7, April 2013.

[28] Carofiglio, G., Muscariello, L., Papalini, M., Rozhnova, N., and X. Zeng, "Leveraging ICN In-network Control for Loss Detection and Recovery in Wireless Mobile networks", Proc. ICN ACM, September 2016.

[29] Ali, M. and U. Niesen, "Coding for Caching: Fundamental Limits and Practical Challenges", IEEE Communications Magazine vol. 54, no. 8, August 2016.

[30] Koetter, R. and F. Kschischang, "An algebraic approach to network coding", IEEE Trans. Netw. vol.11, no.5, October 2008.

[31] Tournoux, P., Lochin, E., Lacan, J., Bouabdallah, A., and V. Roca, "On-the-Fly Erasure Coding for Real-Time Video Applications", IEEE Trans. Multimeda vol.13, no.4, August 2011.

Authors' Addresses

Kazuhisa Matsuzono
National Institute of Information and Communications Technology
4-2-1 Nukui-Kitamachi
Koganei, Tokyo 184-8795
Japan

Email: matsuzono@nict.go.jp

Hitoshi Asaeda
National Institute of Information and Communications Technology
4-2-1 Nukui-Kitamachi
Koganei, Tokyo 184-8795
Japan

Email: asaeda@nict.go.jp

Cedric Westphal
Huawei
2330 Central Expressway
Santa Clara, California 95050
USA

Email: cedric.westphal@huawei.com

NWCRG
Internet-Draft
Intended status: Informational
Expires: September 22, 2018

V. Roca (Ed.)
INRIA
J. Detchart
ISAE - Supaero
C. Adjih
INRIA
M. Pedersen
Steinwurf ApS
I. Swett
Google
March 21, 2018

Generic Application Programming Interface (API) for Sliding Window FEC Codes
draft-roca-nwcrg-generic-fec-api-01

Abstract

This document introduces a generic Application Programming Interface (API) for sliding window FEC codes. This API is meant to be compatible with any sliding window FEC code. It defines the core procedures and functions meant to control the codec (i.e., implementation of the FEC code), but leaves out all upper layer aspects that are the responsibility of the application or protocol making use of the codec. As a consequence, this is not an API for a FEC Scheme since certain mechanisms that must be defined by any FEC Scheme (e.g., signalling and FEC Payload IDs) are the responsibility of the caller instead of being addressed by the codec. A goal of this document is to pave the way for a future open-source implementation of such codes, another goal is to simplify the development of content delivery protocols that rely on sliding window FEC codes for robust transmissions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 22, 2018.

Table of Contents

1. Introduction

Forward Erasure Correction (FEC) codes are a key element of communication systems, used to efficiently recover from packet losses during content delivery sessions. Among the FEC codes working at the network and higher layers, one can broadly distinguish block codes and sliding window codes. Block FEC codes require the data flow coming from the application to be segmented into blocks of a predefined maximum size, before generating a certain number of repair packets. With the second type of FEC codes, an encoding window continuously slides over the set of source data and repair packets are generated at any time by computing for instance a linear combination of data present in the encoding window. This fondamental difference seriously impacts the way they can be used by a content delivery protocol or application.

This document introduces a generic Application Programming Interface (API) for sliding window FEC codes. This API is meant to be usable by any sliding window FEC code and FEC Scheme independently of the protocol that may rely on it. This API defines the core procedures and functions meant to control the codec (i.e., implementation of the FEC code), but leaves out all upper layer aspects that are the responsibility of the application making use of the codec.

This API is meant to be usable by any sliding window FEC code. independently of the FEC Scheme or network coding protocol that may rely on it This API defines the core procedures and functions meant to control the codec (i.e., implementation of the FEC code), but leaves out all upper layer aspects that are the responsibility of the application making use of the codec. For instance, those restricted to end-to-end use-cases as well as those compatible with in-network re-encoding use-cases. Additionaly, this API is not impacted by the intra-flow versus inter-flow nature of the use-case, nor is it impacted by the single-path versus multi-paths nature of the use-case, since those are usage considerations under the responsibility of the caller.

A goal of this document is to pave the way for a future open-source implementation of such codes.

2. Definitions and Abbreviations

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

This document uses the following definitions and abbreviations:

XXX

3. AL-FEC Codes and Mechanisms Considered by the Generic API

This generic FEC API is meant to be used with:

- sliding window codes, that manage an encoding window (of fixed or variable size) that slides over the set of source symbols at the sender. On the opposite, block codes (e.g., Reed-Solomon, LDPC, Raptor(Q)) are out of scope;
- codes that are restricted to use-cases that involve a single encoding point and a single decoding point (i.e., FEC operations are carried out either within the end-hosts or middle-boxes), as well as codes that can be used with use-cases that involve in-network re-coding operations;
- use-cases that are limited to an intra-flow coding (simple case), as well as use-cases that involve inter-flow coding. This second case is more complex to address (e.g., with questions such as how to identify a packet of a flow?) however this is the responsibility of the application or protocol using this codec and not the codec itself. This aspect is therefore transparent to the API;
- use-cases that are limited to single-path communications and use-cases that consider multi-path communications. Here also this is a usage consideration that is transparent to the API;
- use-cases that involve a dynamic adaptation of the codec parameters (e.g., its code rate because the communication path losses is known thanks to feedbacks and an appropriate strategy can be defined);
- fixed code rate or not FEC codes, including rateless codes where the number of repair symbols that can be generated is huge (in theory illimited);
- ideal (MDS) or non ideal (non-MDS) codes. However most of the time, sliding window codes are non-ideal codes, meaning that slightly more than l repair symbols may be required to recover all the l lost source symbols;

A key question is to determine what mechanisms are included in the codec and what mechanisms are left to the responsibility of the caller (i.e., an application or a protocol making use of this codec). More precisely, an FEC Scheme (such as the RLC FEC Scheme [RLC] in case of FECFRAME [fecframe-ext]) defines all the internal code details in order to enable interoperable implementations, but also signaling considerations that are essential to use them in a specific context.

3.1. Mechanisms Considered or Ignored by the API

Applying FEC in a given use-case to improve robustness involves many mechanisms. However they are not all the responsibility of the codec and can be instead implemented within the application or protocol that uses the codec. For instance the following mechanisms are considered out of scope of the API, being implemented by the caller, without any impact on the codec:

- code rate adjusment (e.g., thanks to communication path experienced loss feedback);
- signaling header creation / parsing (TBC, see discussion below);
- packet management;
- packet transmission / reception;
- tunnel management (if any);
- congestion control;
- selective ACK creation / parsing;
- memory management;

The following mechanisms are within scope of the API:

- session management (sender and receiver);
- encoding window management (sender and receiver): XXX: TO BE DISCUSSED;
- set/get/compute coding coefficient (sender and receiver);
- build coded symbol (sender);
- decode with received source or repair symbol (receiver);

4. Generic API Proposal

The following sections describe the generic API, following a C-language formalism. Everything is prefixed by XXX_, which stands for XXX.

4.1. General definitions (Sender and Receiver)

```
<CODE BEGINS>
// all the definitions of interest

/**
 * Function return value, indicating whether the function call succeeded or not.
 *
 * STATUS_OK = 0      Success
 * STATUS_FAILURE,    Failure. The function called did not succeed to perform
 *                    its task, however this is not an error. This can happen
 *                    for instance when decoding did not succeed (which is a
 *                    valid output).
 * STATUS_ERROR,      Generic error type. The detailed error type is returned
 *                    in a global variable, TBD_errno.
 */
typedef enum {
        STATUS_OK = 0,
        STATUS_FAILURE,
        STATUS_ERROR
} TBD_status_t;

<CODE ENDS>
```

API proposal

4.2. Encoder

<CODE BEGINS>

<CODE ENDS>

Encoder API proposal

4.3. Decoder

<CODE BEGINS>

<CODE ENDS>

Decoder API proposal

5. Security Considerations

TBD

6. IANA Considerations

N/A.

7. Acknowledgments

The authors would like to thank TBD.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6363] Watson, M., Begen, A., and V. Roca, "Forward Error Correction (FEC) Framework", RFC 6363, DOI 10.17487/RFC6363, October 2011, <https://www.rfc-editor.org/info/rfc6363>.

8.2. Informative References

[fecframe-ext] Roca, V. and A. Begen, "Forward Error Correction (FEC) Framework Extension to Sliding Window Codes", Transport Area Working Group (TSVWG) draft-ietf-tsvwg-fecframe-ext (Work in Progress), March 2018, <https://tools.ietf.org/html/draft-ietf-tsvwg-fecframe-ext>.

[RLC] Roca, V., "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Scheme for FECFRAME", Transport Area Working Group (TSVWG) draft-ietf-tsvwg-rlc-fec-scheme (Work in Progress), March 2018, <https://tools.ietf.org/html/draft-ietf-tsvwg-rlc-fec-scheme>.

Appendix A. Existing APIs

Editor's comment: we list a few existing APIs for reference and inspiration purposes. They will be removed in future versions of this document.

A.1. Morten API proposal

A.1.1. Encoder

```
<CODE BEGINS>
// Copyright Steinwurf ApS 2011.
// Distributed under the "STEINWURF RESEARCH LICENSE 1.0".
// See accompanying file LICENSE.rst or
// http://www.steinwurf.com/licensing

class encoder
{
public:
    /// Factory for encoders. The factory is used to build and initialize
    /// encoders. If needed, e.g. for efficiency reasons, it is possible to
    /// re-initialize already built encoders in order to reuse them.
    class factory
    {
    public:
        /// Constructor
        factory();

        /// @return The current specified symbol size in bytes.
        uint64_t symbol_size() const;

        /// @param symbol_size Sets the size of a symbol in bytes.
        void set_symbol_size(uint64_t symbol_size);

        /// @param field Set the finite field to use.
        void set_field(finite_field field);

        /// @return The finite field used.
        finite_field field() const;

        /// @return A new encoder.
        encoder build();

        /// @param encoder Initialize a encoder with the factory settings. After
        ///        calling initialize the encoder will be ready for use.
        void initialize(encoder&);
        [...]
    };
```

```
public:
    /// @return The total number of symbols available in memory at the encoder.
    ///         The number of symbols in the coding window MUST be less than
    ///         or equal to this number. The total range of valid symbol
    ///         indices is:
    ///
    ///             for (uint64_t i = 0; i < stream_symbols(); ++i)
    ///             {
    ///                 std::cout << i + stream_lower_bound() << "\n";
    ///             }
    ///
    uint64_t stream_symbols() const;

    /// @return The index of the oldest symbol known by the encoder. This symbol
    ///         may not be inside the window but can be included in the window
    ///         if needed.
    uint64_t stream_lower_bound() const;

    /// @return The upper bound of the stream. The range of valid symbol indices
    ///         goes from [encoder::stream_lower_bound(),
    ///         encoder::stream_upper_bound()). Note the stream is a half-open
    ///         interval. Going from encoder::stream_lower_bound() to
    ///         encoder::stream_upper_bound() - 1.
    uint64_t stream_upper_bound() const;

    /// @return The size of a symbol in the stream in bytes.
    uint64_t symbol_size() const;

    /// Adds a new symbol to the front of the encoder. Increments the number of
    /// symbols in the stream and increases the encoder::stream_upper_bound().
    ///
    /// @param symbol Pointer to the symbol. Note, the caller must ensure that
    ///        the memory of the symbol remains valid as long as the symbol is
    ///        included in the stream. The caller is responsible for freeing the
    ///        memory if needed. Once the symbol is popped from the stream.
    /// @return The stream index of the symbol being added.
    uint64_t push_front_symbol(const uint8_t* symbol);

    /// Remove the "oldest" symbol from the stream. Increments the
    /// encoder::stream_lower_bound().
    /// @return The index of the symbol being removed
    uint64_t pop_back_symbol();

    /// @return The number of symbols currently in the coding window. The
    ///         window must be within the bounds of the stream.
    uint64_t window_symbols() const;

    /// @return The index of the "oldest" symbol in the coding window.
```

```
    uint64_t window_lower_bound() const;

    /// @return The upper bound of the window. The range of valid symbol indices
    ///         goes from [encoder::window_lower_bound(),
    ///         encoder::window_upper_bound()). Note the window is a half-open
    ///         interval. Going from encoder::window_lower_bound() to
    ///         encoder::window_upper_bound() - 1.
    uint64_t window_upper_bound() const;

    /// The window represents the symbols which will be included in the next
    /// encoding. The window cannot exceed the bounds of the stream.
    ///
    /// Example: If window_lower_bound=4 and window_symbol=3 the following
    ///          symbol indices will be included 4,5,6
    ///
    /// @param window_lower_bound Sets the index of the oldest symbol in the
    ///        window.
    /// @param window_symbols Sets number of symbols within the window.
    void set_window(uint64_t window_lower_bound, uint64_t window_symbols);

    /// @return The size of the coefficient vector in the current window in
    ///         bytes. The number of coefficients is equal to the number of
    ///         symbols in the window. The size in bits of each coefficients
    ///         depends on the finite field chosen. A custom coding scheme can
    ///         be implemented by generating the coding vector manually.
    ///         Alternatively the built-in generator can be used. See
    ///         encoder::set_seed(...) and encoder::generate(...).
    uint64_t coefficient_vector_size() const;

    /// Seed the internal random generator function. If using the same seed
    /// on the encoder and decoder the exact same set of coefficients will
    /// be generated.
    /// @param seed_value A value for the seed.
    void set_seed(uint64_t seed_value);

    /// Generate coding coefficients for the symbols in the coding window
    /// according to the specified seed (see encoder::set_seed(...)).
    /// @param coefficients Buffer where the coding coefficients should be
    ///        stored. This buffer must be encoder::coefficient_vector_size()
    ///        large in bytes.
    void generate(uint8_t* coefficients);

    /// Write an encoded symbol according to the coding coefficients.
    /// @param symbol The buffer where the encoded symbol will be stored.
    ///        The symbol buffer must be encoder::symbol_size() large.
    /// @param coefficients The coding coefficients. These must have the
    ///        memory layout required (see README.rst). A compatible format can
    ///        be created using encoder::generate(...)
```

```
    void write_symbol(uint8_t* symbol, const uint8_t* coefficients);

    /// Write a source symbol to the symbol buffer.
    /// @param symbol The buffer where the source symbol will be stored. The
    ///        symbol buffer must be encoder::symbol_size() large.
    /// @param index The symbol index which should be written.
    void write_source_symbol(uint8_t* symbol, uint64_t index);
    [...]
};
<CODE ENDS>
```

Morten API proposal

A.1.2. Decoder

```
<CODE BEGINS>

// Copyright Steinwurf ApS 2011.
// Distributed under the "STEINWURF RESEARCH LICENSE 1.0".
// See accompanying file LICENSE.rst or
// http://www.steinwurf.com/licensing

class decoder
{
public:
    class factory
    {
    public:
        /// Constructor
        factory();

        /// @return The current specified symbol size in bytes.
        uint64_t symbol_size() const;

        /// @param symbol_size Sets the size of a symbol in bytes
        void set_symbol_size(uint64_t symbol_size);

        /// @param field Set the finite field to use
        void set_field(finite_field field);

        /// @return The finite field used.
        finite_field field() const;

        /// @return A new decoder.
        decoder build();

        /// @param decoder Initialize a decoder with the factory settings. After
        ///        calling initialize the decoder will be ready for use.
```

```
        void initialize(decoder&);
        [...]
    };

public:
    /// @return The total number of symbols known at the decoder. The number of
    ///         symbols in the decoding window MUST be less than or equal to
    ///         this number. The total range of valid symbol indicies is
    ///
    ///             for (uint64_t i = 0; i < stream_symbols(); ++i)
    ///             {
    ///                 std::cout << i + stream_lower_bound() << "\n";
    ///             }
    ///
    uint64_t stream_symbols() const;

    /// @return The index of the oldest symbol known by the decoder. This symbol
    ///         may not be inside the window but can be included in the window
    ///         if needed.
    uint64_t stream_lower_bound() const;

    /// @return The upper bound of the stream. The range of valid symbol indices
    ///         goes from [decoder::stream_lower_bound(),
    ///         decoder::stream_upper_bound()). Note the stream is a half-open
    ///         interval. Going from decoder::stream_lower_bound() to
    ///         decoder::stream_upper_bound() - 1.
    uint64_t stream_upper_bound() const;

    /// @return The size of a symbol in the stream in bytes.
    uint64_t symbol_size() const;

    /// Adds a new symbol to the front of the decoder. Increments the number of
    /// symbols in the stream and increases the decoder::stream_upper_bound().
    ///
    /// @param symbol Pointer to the symbol. Note, the caller must ensure that
    ///        the memory of the symbol remains valid as long as the symbol is
    ///        included in the stream. The caller is responsible for freeing the
    ///        memory if needed. Once the symbol is popped from the stream.
    /// @return The stream index of the symbol being added.
    uint64_t push_front_symbol(uint8_t* symbol);

    /// Remove the "oldest" symbol from the stream. Increments the
    /// decoder::stream_lower_bound().
    /// @return The index of the symbol being removed
    uint64_t pop_back_symbol();

    /// @return The number of symbols currently in the coding window. The
    ///         window must be within the bounds of the stream.
```

```
    uint64_t window_symbols() const;

    /// @return The index of the "oldest" symbol in the coding window.
    uint64_t window_lower_bound() const;

    /// @return The upper bound of the window. The range of valid symbol indices
    ///         goes from [decoder::window_lower_bound(),
    ///         decoder::window_upper_bound()). Note the window is a half-open
    ///         interval. Going from decoder::window_lower_bound() to
    ///         decoder::window_upper_bound() - 1.
    uint64_t window_upper_bound() const;

    /// The window represents the symbols which will be included in the next
    /// decoding. The window cannot exceed the bounds of the stream.
    ///
    /// Example: If window_lower_bound=4 and window_symbol=3 the following
    ///          symbol indices will be included 4,5,6
    ///
    /// @param window_lower_bound Sets the index of the oldest symbol in the
    ///        window.
    /// @param window_symbols Sets number of symbols within the window.
    void set_window(uint64_t window_offset, uint64_t window_symbols);

    /// @return The size of the coefficient vector in the current window in
    ///         bytes. The number of coefficients is equal to the number of
    ///         symbols in the window. The size in bits of each coefficients
    ///         depends on the finite field chosen. A custom coding scheme can
    ///         be implemented by generating the coding vector manually.
    ///         Alternatively the built-in generator can be used. See
    ///         decoder::set_seed(...) and decoder::generate(...).
    uint64_t coefficient_vector_size() const;

    /// Seed the internal random generator function. If using the same seed
    /// on the decoder and decoder the exact same set of coefficients will
    /// be generated.
    /// @param seed_value A value for the seed.
    void set_seed(uint64_t seed_value);

    /// Generate coding coefficients for the symbols in the coding window
    /// according to the specified seed (see decoder::set_seed(...)).
    /// @param coefficients Buffer where the coding coefficients should be
    ///        stored. This buffer must be decoder::coefficient_vector_size()
    ///        large in bytes.
    void generate(uint8_t* coefficients);

    /// Decodes a coded symbol according to the coding coefficients.
    ///
    /// Both buffers may be modified during this call. The reason for this
```

```
    /// is that the decoder will directly operate on the provided memory
    /// for performance reasons.
    ///
    /// @param symbol Buffer representing a coded symbol.
    ///
    /// @param coefficients The coding coefficients used to
    ///        create the encoded symbol
    void read_symbol(uint8_t* symbol, uint8_t* coefficients);

    /// Add a source symbol at the decoder.
    ///
    /// @param symbol Buffer containing the source symbol's data.
    /// @param index The index of the source symbol in the stream
    void read_source_symbol(uint8_t* symbol, uint64_t index);

    /// The rank of a decoder indicates how many symbols have been
    /// partially or fully decoded. This number is also equivalent to the
    /// number of pivot elements we have in the stream.
    ///
    /// @return The rank of the decoder
    uint64_t rank() const;

    /// @return The number of missing symbols at the decoder
    uint64_t symbols_missing() const;

    /// @return The number of partially decoded symbols at the decoder
    uint64_t symbols_partially_decoded() const;

    /// @return The number of decoded symbols at the decoder
    uint64_t symbols_decoded() const;

    /// @param index Index of the symbol to check.
    ///
    /// @return True if the symbol is decoded (i.e. it corresponds to a source
    ///         symbol), and otherwise false.
    bool is_symbol_decoded(uint64_t index) const;
    [...]
};
<CODE ENDS>
```

Morten API proposal

A.2. Jonathan API proposal

```
<CODE BEGINS>
/** a status for function calls **/
typedef enum {
    STATUS_OK,
```

```
    STATUS_ERROR,
    /* ... */
} status_t;


/** defines the galois field used (at least the size, maybe we need to separate
the implementations (Lookup Tables or Xor-based)) **/
typedef enum {
    GF_16,
    GF_64,
    GF_256
} galois_field_t;


/*
 * coding coefficient generators: specifies the algorithm used to generate the c
oefficients for the linear combinations
 *
 * NOTE: the choice of the finite field could done here (RLC_GF256, RLC_GF_16, .
..) rather than using a different structure
 */
typedef enum {
    RLC,
    VDM
    /* ... */
} coding_coefficients_generator_identifier_t;


 /***
  **
  ** encoder side
  **
  ***/


/**
 * NOTE for the callbacks in the sw_encoder: this is a proposition, we could als
o use 2 structures (source_t and repair_t) to avoid sending too many parameters
in the callbacks.
 **/


/**
 * context: a context as a generic pointer (defined by the application if needed
 and given in sw_encoder_set_callbacks)
 * src: the source data unit to consider
 * src_id: the id of the source unit set by the encoder
 * src_sz: the size in bytes of the data unit
 **/
typedef void (*sw_encoder_callback_source_ready)(void *context,void* src, uint32
_t src_id, size_t src_sz);


/**
 * context: a context as a generic pointer (defined by the application if needed
 and given in sw_encoder_set_callbacks)
```

```
 * rep: the repair data unit generated by the encoder
 * rep_id: the id of the repair unit given by the encoder
 * rep_sz: the size in bytes of the repair data unit
 * src_ids: the array of the source unit ids used to generate the repair unit
 * src_coefs: the coefficients used for each source units to generate the repair
 unit
 * nb_src_in: number of src units in the linear combination (repair unit) (also
the number of elements of src_ids and src_coefs)
 **/
typedef void (*sw_encoder_callback_repair_ready)(void *context,void* rep, uint32
_t rep_id, size_t rep_sz, uint32_t* src_ids, uint8_t *src_coefs, size_t nb_src_i
n);


/** a structure containing all we need to encode **/
typedef struct sw_encoder sw_encoder_t;


/**
 * @brief Init a sliding window encoder by giving a galois field size, a coeffic
ient generator function, and the maximum size of the window
 * @param galois_field                    The size of the galois field used to
create the coefficients and to encode.
 * @param ccgi                                    The function used to generate
 the coefficients (depends on the gf_size)
 * @param max_wnd_size                    set the maximum of source units t con
sider inside the coding window (the oldest units will be destroyed)
 * @return a sw_encoder_t structure.
 **/
sw_encoder_t* sw_encoder_init(galois_field_t galois_field, coding_coefficients_g
enerator_identifier_t ccgi, uint32_t max_wnd_size);


/**
 * @brief Set the callbacks to get the encoded (repair) data
 * @param encoder                              The encoder initialized
 * @param context                             A generic context defined by the ap
plication (will be given in the callback)
 * @param src_callback                    The function to be called when a sour
ce data unit has been processed by the encoder
 * @param rep_callback                   The function to be called when a repai
r data unit has been generated
 **/
status_t sw_encoder_set_callbacks(sw_encoder_t* encoder, void* context, sw_encod
er_callback_source_ready src_callback, sw_encoder_callback_repair_ready rep_call
back);


/**
 * @brief Gives a source data unit  and its id to the encoder
 * @param encoder     The encoder structure.
 * @param src                 The data to add
 * @param sz             The size in bytes of the data unit
 **/
status_t sw_encoder_add_source(sw_encoder_t* encoder, void* src, size_t sz);


/**
 * @brief Removes the corresponding source data unit from the encoding window by
 giving and id
 * @param encoder     The encoder structure
 * @param id             The id of the data unit
 * @return                STATUS_OK if the data unit has been found and rem
```

oved, STATUS_ERROR if the data unit doesn’t exist

```
 **/
status_t sw_encoder_remove_source(sw_encoder_t* encoder, uint32_t id);


/**
 * @brief generates a repair data unit and calls the corresponding callback.
 * @param encoder             The encoder structure.
 **/
status_t sw_encoder_generate_repair(sw_encoder_t* encoder);


/* ... */
status_t sw_encoder_set_control_parameter(sw_encoder_t* encoder, uint32_t type,
void* value, uint32_t length);


/* ... */
status_t sw_encoder_get_control_parameter(sw_encoder_t* encoder, uint32_t type,
void* value, uint32_t length);


/**
 * @brief Release an encoder structure
 * @param encoder        The encoder structure
 **/
status_t  sw_encoder_release(sw_encoder_t* encoder);


 /***
  **
  ** decoder side
  **
  ***/


/**
 * NOTE for the callback in the sw_decoder: this is a proposition, we could also
 use an opaque structure (source_t) to avoid sending too many parameters in the
callback.
 **/
/**
 * context: a context as a generic pointer (defined by the application if needed
 and given in sw_encoder_set_callbacks)
 * src: the source data unit to consider
 * src_id: the id of the source unit used in the decoder
 * src_sz: the size in bytes of the data unit
 **/
typedef void (*sw_decoder_callback_source_ready)(void *context,void* src, uint32
_t src_id, size_t sz);


/** a structure containing all we need to decode **/
typedef struct sw_decoder sw_decoder_t;
```

```
/**
 * @brief Init a sliding window decoder by giving a galois field size, a coeffic
ient generator function, and the maximum size of the window
 * @param galois_field                      The size of the galois field used to
create the coefficients and to encode.
 * @param ccgi                                      The function used to generate
 the coefficients (depends on the gf_size)
 * @return a sw_decoder_t structure.
 **/
sw_decoder_t* sw_decoder_init(galois_field_t galois_field, coding_coefficients_g
enerator_identifier_t ccgi);


/**
 * @brief Set the callback to get the encoded (repair) data
 * @param decoder                                The decoder initialized
 * @param context                               A generic context defined by the ap
plication (will be given in the callback)
 * @param callback                               The function to be called
 **/
status_t sw_decoder_set_callback_source_ready(sw_decoder_t* decoder, void* conte
xt, sw_decoder_callback_source_ready callback);


/**
 * NOTE for the next decode functions: we could also use 2 opaque structures to
represent the source and repair units (source_t or repair_t)
 **/


/**
 * @brief decode some source data units by giving to the decoder a new source da
ta unit
 * @param decoder                                the decoder structure
 * @param src                                       the new source data unit
 * @param src_id                                the id of the new source data unit
(given by an encoder)
 * @param src_sz                                the size in bytes of the new source
 data unit
 **/
status_t sw_decoder_decode_with_source(sw_decoder_t* decoder, void* src, uint32_
t src_id, size_t src_sz);


/**
 * @brief decode some source data units by giving to the decoder a new repair da
ta unit
 * @param decoder                                the decoder structure
 * @param rep                                       the new repair data unit
 * @param rep_id                                the id of the new repair data unit
(given by an encoder)
 * @param rep_sz                                the size in bytes of the new repair
 data unit
 * @param src_ids:                               the array of the source unit ids
used to generate this repair unit
 * @param src_coefs:                        the coefficients used for each source u
nits to generate this repair unit
 * @param nb_src_in:                        number of src units in the linear combi
nation (repair unit) (also the number of elements of src_ids and src_coefs)
 **/
status_t sw_decoder_decode_with_repair(sw_decoder_t* decoder, void* rep, uint32_
t rep_id, size_t rep_sz, uint32_t* src_ids, uint8_t * src_coefs, size_t nb_src_i
n);
```

```
/* ... */
status_t sw_decoder_set_control_parameter(sw_decoder_t* decoder, uint32_t type,
void* value, uint32_t length);
```

```
/* ... */
status_t sw_decoder_get_control_parameter(sw_decoder_t* decoder, uint32_t type,
void* value, uint32_t length);


/**
 * @brief Release a decoder structure
 * @param decoder        The decoder structure to release
 **/
status_t sw_decoder_release(sw_decoder_t* decoder);
<CODE ENDS>
```

Jonathan API proposal

A.3. Cedric API proposal

For DRAGONCAST/DragonNet/GardiNet (<https://gitlab.inria.fr/GardiNet/liblc/>):

- an API could be globally pretty similar;
- there is a maintained set of symbols of the "codec" where online Gaussian Elimination is performed. But this same set, is used to also re-code packets for generation. For this to work, one uses as pivot the highest index (instead of the lowest in standard RREF), in order to avoid adding symbols with higher indices in the decoding process.
- another set of differences would be that the protocol has more control over the coding process than our current codec proposal. The reason is that DRAGONCAST (re)codes for several neighbors, and in such scenario, there is no "obvious" decision that can be made, for instance:
  - which source symbols (indices) should be present in a generated packet: -> tradeoff: helping the maximum number of nodes (emphasis on "new" undecoded source symbol indices) -vs- helping the neighbor which is the most late in the decoding process (emphasis on "old" source symbols)
  - which symbols should be kept in the decoding process (or dropped): -> tradeoff: helping coding by keeping old symbols (be able to generate symbols for late neighbors) vs keeping up with decoding (and never throwing away a new symbol with high indices). (I. Amdouni discussed such issues in <https://tools.ietf.org/html/draft-amdouni-nwcrg-cisew-00> for instance).
- In the current implementation, the packet generation process is done in the protocol which directly "peeks" in the set of symbols in the codec, and creates a linear combination with the ones that suits it.
- Technically the callbacks from the "codec" are:

* notification of a source symbol is decoded;
* notification that the set of symbols is full (protocol can remove symbols it sees fits, especially decoded symbols);
* for the "over-the-air" reflashing application, the codec can ask the protocol if an already removed source symbol is available (on the assumption that it has been written somewhere else);

A.4. Ian API proposal

A.5. Vincent API proposal

A.5.1. General

```
<CODE BEGINS>
/**
 * The fec_codec_id_t enum identifies the FEC code/codec being used.
 * Since a given fec_codec_id can be used by one or several FEC schemes (that sp
ecify
 * both the codes and way of using these codes), it is distinct from the FEC Enc
oding
 * ID.
 */
typedef enum {
        CODEC_NIL = 0,
        CODEC_RLC
} codec_id_t;


/**
 * Function return value, indicating whether the function call succeeded or not.
 * In case of failure, the detailed error type is returned in a global variable,
 * of_errno (see of_errno.h).
 *
 * STATUS_OK = 0      Success
 * STATUS_FAILURE,    Failure. The function called did not succeed to perform
 *                    its task, however this is not an error. This can happen
 *                    for instance when decoding did not succeed (which is a
 *                    valid output).
 * STATUS_ERROR,      Generic error type. The caller is expected to be able
 *                    to call the library in the future after having corrected
 *                    the error cause.
 * STATUS_FATAL_ERROR Fatal error. The caller is expected to stop using this
 *                    codec instance immediately (it replaces an exit() system
 *                    call).
 */
typedef enum {
        STATUS_OK = 0,
        STATUS_FAILURE,
        STATUS_ERROR,
```

```
        STATUS_FATAL_ERROR
} status_t;


/**
 * Throughout the API, a pointer to this structure is used as an identifier of t
he current
 * codec instance, also known as "session".
 *
 * This generic structure is meant to be extended by each codec and new pieces o
f information
 * that are specific to each codec be specified there. However, all the codec sp
ecific structures
 * MUST begin the same entries as the ones provided in this generic structure, o
therwise
 * hazardous behaviors may happen.
 */
typedef struct session {
        codec_id_t   codec_id;
        codec_type_t codec_type;
} session_t;


/**
* Generic FEC parameter structure used by set_fec_parameters().
*
* This generic structure is meant to be extended by each codec and new pieces of
 information
* that are specific to each codec be specified there. However, all the codec spe
cific structures
* MUST begin the same entries as the ones provided in this generic structure, ot
herwise
* hazardous behaviors may happen.
*/
typedef struct {
       /** SENDER and RECEIVER: maximum number of source symbols used for any re
pair symbol. */
       UINT32      coding_window_max_size;


        /** RECEIVER only: maximum number of source symbols kept in current line
ar
         * system. If the linear system grows above this limit, older source sym
bols
         * in excess are removed and the application callback called if set. Thi
s
         * value MUST be larger than the coding_window_max_size. */
    UINT32      linear_system_max_size;
   UINT32      encoding_symbol_length;
} parameters_t;
<CODE ENDS>
```

Vincent API proposal

A.5.2. Session Management

```
<CODE BEGINS>
/**
 * This function allocates and partially initializes a new session structure.
```

```
 * Throughout the API, a pointer to this session is used as an identifier of the
 * current codec instance.
 *
 * @param ses           (IN/OUT) address of the pointer to a session. This point
er is updated
 *                      by this function.
 *                      In case of success, it points to a session structure all
ocated by the
 *                      library. In case of failure it points to NULL.
 * @param codec_id      identifies the FEC code/codec being used.
 * @param codec_type    indicates if this is a coder or a decoder.
 * @param verbosity     set the verbosity level
 * @return              Completion status. The ses pointer is updated according
to the success return status.
 */
status_t     create_codec_instance (session_t**  ses,
                                    codec_id_t   codec_id,
                                    codec_type_t codec_type,
                                    uint32_t     verbosity);


/**
 * This function releases all the internal resources used by this FEC codec inst
ance.
 * None of the source symbol buffers will be free'ed by this function, even thos
e decoded by
 * the library if any, regardless of whether a callback has been registered or n
ot. It's the
 * responsibility of the caller to free them.
 *
 * @param ses           (IN) Pointer to the session.
 * @return              Completion status
 */
status_t     release_codec_instance (session_t*        ses);


/**
 * Second step of the initialization, where the application specifies code(c) sp
ecific parameters.
 *
 * At a receiver, the parameters can be extracted from the FEC OTI that is usual
ly communicated
 * to the receiver by either an in-band mechanism or an out-of-band mechanism, o
r set statically
 * for a specific use-case.
 *
 * @param ses           (IN) Pointer to the session.
 * @param params        (IN) pointer to a structure containing the FEC parameter
s associated to
 *                      a specific FEC codec.
 * @return              Completion status.
 */
status_t     set_fec_parameters (session_t*         ses,
                                 parameters_t*      params);


/**
 * This function sets a FEC scheme/FEC codec specific control parameter,
```

```
 * using a type/value method.
 *
 * @param ses           (IN) Pointer to the session.
 * @param type          (IN) Type of parameter. This type is FEC codec ID specif
ic.
 * @param value         (IN) Pointer to the value of the parameter. The type of
the object pointed
 *                      is FEC codec ID specific.
 * @param length        (IN) length of pointer value
 * @return              Completion status.
 */
status_t     set_control_parameter (session_t*    ses,
                                    UINT32        type,
                                    void*         value,
                                    UINT32        length);


/**
 * This function gets a FEC scheme/FEC codec specific control parameter,
 * using a type/value/length method.
 *
 * @param ses           (IN) Pointer to the session.
 * @param type          (IN) Type of parameter. This type is FEC codec ID specif
ic.
 * @param value         (IN/OUT) Pointer to the value of the parameter. The type
 of the object
 *                      pointed is FEC codec ID specific. This function updates
the value object
 *                      accordingly. The application, who knows the FEC codec ID
, is responsible
 *                      to allocating the appropriate object pointed by the valu
e pointer.
 * @param length        (IN) length of pointer value
 * @return              Completion status.
 */
status_t     get_control_parameter (session_t*    ses,
                                    UINT32        type,
                                    void*         value,
                                    UINT32        length);
<CODE ENDS>
```

Vincent API proposal

A.5.3. Callback Functions

```
<CODE BEGINS>
/**
 * Set the various callback functions for this session.
 * All the callback functions require an opaque context parameter, that must be
 * initialized accordingly by the application, since it is application specific.
 *
 * @param ses           (IN) Pointer to the session.
 *
 * @param decoded_source_symbol_callback
 *                      (IN) Pointer to the function, within the application, th
at
```

```
 *                       needs to be called each time a source symbol is decoded.
 *
 * @param available_source_symbol_callback
 *                       (IN) Pointer to the function, within the application, th
at
 *                       needs to be called each time a source symbol is decoded
and
 *                       all computations performed (i.e., the buffer does contai
n the
 *                       symbol value).
 *
 * @param source_symbol_removed_from_coding_window_callback
 *                       (IN) Pointer to the function, within the application, th
at
 *                       needs to be called each time a source symbol is removed
from
 *                       the left side of the coding window, at a SENDER because
this
 *                       window has slided to the right, or at a RECEIVER because
 this
 *                       old source symbol is now forgotten.
 *
 * @param context_4_callback
 *                       (IN) Pointer to the application-specific context that wi
ll be
 *                       passed to the callback function (if any). This context i
s not
 *                       interpreted by this function.
 *
 * @return               Completion status.
 */


status_t     set_callback_functions (of_session_t*        ses,
    void* (*decoded_source_symbol_callback) (void *context,
                                             UINT32      size,           /* size
 of decoded source symbol */
                                             UINT32      esi),           /* enco
ding symbol ID */
    void (*available_source_symbol_callback) (void       *context,
                                              void       *new_symbol_buf,/* symb
ol buffer */
                                              UINT32     size,           /* size
 of decoded source symbol */
                                              UINT32     esi),           /* enco
ding symbol ID */
    void (*source_symbol_removed_from_coding_window_callback)
                                             (void   *context,
                                              UINT32 old_symbol_esi),
    void*                context_4_callback);
<CODE ENDS>
```


A.5.4. Coding window functions

TBD

A.5.5. Coding coefficients functions

```
<CODE BEGINS>
/**
* SENDER:   this function specifies the coding coefficients chosen by the applic
ation if this is the way the codec
*      works. This function MUST be called before calling build_repair_symbol().
* RECEIVER: communicate the coding coefficients associated to a repair symbol an
d carried in the packet header.
*      This function MUST be called before calling decode_with_new_repair_symbol
().
*
* @param ses
* @param coding_coefs_tab      (IN) table of coding coefficients to be associate
d to each of the source symbols
*                              currently in the coding window. The size (number
of bits) of each coefficient
*                              depends on the FEC scheme. The allocation and rel
ease of this table is under the
*                              responsibility of the application.
* @param nb_coefs_in_tab       (IN) number of entries (i.e., coefficients) in th
e table.
* @return                     Completion status.
*/
status_t     set_coding_coefficients_tab (session_t*     ses,
                                          void*          coding_coefs_tab,
                                          UINT32         nb_coefs_in_tab);

/**
* SENDER:   this function enables the application to retrieve the set of coding
coefficients generated and used by
*      build_repair_symbol().
* RECEIVER: never used.
*
* @param ses
* @param coding_coefs_tab      (IN/OUT) pointer of a table of coding coefficient
s to be associated to each of the
*                              source symbols currently in the coding window. Th
e size (number of bits) of each
*                              coefficient depends on the FEC scheme. The alloca
tion and release of this table is
*                              under the responsibility of the application. Upon
 return of this function, this
*                              table is allocated and filled with each coefficie
nt value.
* @param nb_coefs_in_tab       (IN/OUT) pointer to the number of entries (i.e.,
coefficients) in the table.
*                              Upon calling this function, this number must be z
ero. Upon return of this function
*                              this number is initialized with the actual number
 of entries in the coeffs_tab[].
* @return                     Completion status (OF_STATUS_OK, FAILURE, ERROR o
r FATAL_ERROR).
*/
status_t     get_coding_coefficients_tab (session_t*     ses,
                                          void**         coding_coefs_tab,
                                          UINT32*        nb_coefs_in_tab);

/**
* The coding coefficients may be generated in a deterministic manner, (e.g., thr
ough the use of a PRNG and the
* repair symbol ESI used as a seed). This is the case with RLC codes.
*
```

```
* SENDER:   generate all coefficients. This function MUST be called before calli
ng build_repair_symbol().
* RECEIVER: generate all coefficients. This function MUST be called before calli
ng decode_with_new_repair_symbol().
*
```

```
* @param ses
* @param params                (IN) pointer to a codec specific structure contai
ning the required parameters.
*                              These parameters can include a repair symbol ESI
or key among other things.
* @return                     Completion status.
*/
status_t     generate_coding_coefficients (session_t*     ses,
                                           void*          params);
```

<CODE ENDS>

Vincent API proposal

A.5.6. Encoder specific functions

```
<CODE BEGINS>
/**
 * Create a single repair symbol, i.e. perform an encoding.
 * This function requires that the application has previously set the coding win
dow and if needed the coding coefficients
 * appropriately. After that, the application can call this function.
 *
 * @param ses
 * @param new_repair_symbol_buf (IN) The pointer to the buffer for the repair sy
mbol to build can either point to a buffer
 *                               allocated by the application, or let to NULL mea
ning that this function will allocate
 *                               memory.
 * @return                      Completion status.
 */
status_t     ccod_build_repair_symbol (session_t*      ses,
                                       void*           new_repair_symbol_buf);
```

<CODE ENDS>

Vincent API proposal

A.5.7. Decoder specific functions

```
<CODE BEGINS>
/**
 * Submit a received source symbol and try to progress in the decoding. For each
 decoded source
 * symbol, if any, the application is informed through the dedicated callback fu
nctions.
 *
 * This function usually returns OF_STATUS_OK, regardless of whether this new sy
mbol enabled the
 * decoding of one or several source symbols, unless an error occured. This func
tion cannot return
 * OF_STATUS_FAILURE.
 *
 * @param ses
 * @param new_src_symbol_buf    (IN) Pointer to the new source symbol now availa
ble (i.e. a new symbol received by
 *                              the application, or a decoded symbol in case of
a recursive call if it makes sense).
 * @param new_symbol_esi_or_key (IN) encoding symbol ID of the new source symbol
 or key if there is no notion of ESI.
 * @return                      Completion status.
 */
status_t     decode_with_new_source_symbol (session_t*   ses,
                                            void* const     new_src_symbol_buf,
                                            UINT32          new_symbol_esi_or_ke
y);


/**
 * Submit a received repair symbol and try to progress in the decoding. For each
 decoded source
 * symbol, if any, the application is informed through the dedicated callback fu
nctions.
 *
 * This function requires that the application has previously set the coding win
dow and the coding coefficients appropriately.
 * After that, the application can call this function. The
 * application keeps a full control of the repair symbol buffer, i.e., the appli
cation is in charge
 * of freeing this buffer as soon as it believes appropriate to do so (a copy is
 kept by the codec).
 *
 * This function usually returns OF_STATUS_OK, regardless of whether this new sy
mbol enabled the
 * decoding of one or several source symbols, unless an error occurred. This fun
ction cannot return
 * OF_STATUS_FAILURE.
 *
 * @param ses
 * @param new_repair_symbol_buf (IN) Pointer to the new repair symbol now availa
ble (i.e. a new symbol received by
 *                              the application or a decoded symbol in case of a
 recursive call if it makes sense).
 * @return                      Completion status.
 */
status_t     decode_with_new_repair_symbol (session_t*    ses,
                                            void* const   new_repair_symbol_buf)
;
<CODE ENDS>
```

Authors' Addresses

Vincent Roca
INRIA
Grenoble
France

EMail: vincent.roca@inria.fr

Jonathan Detchart
ISAE - Supaero
France

EMail: jonathan.detchart@isae-supaero.fr

Cedric Adjih
INRIA
France

EMail: cedric.adjih@inria.fr

Morten V. Pedersen
Steinwurf ApS
Denmark

EMail: morten@steinwurf.com

Ian Swett
Google
USA

EMail: ianswett@google.com

nwcrg
Internet-Draft
Intended status: Informational
Expires: September 4, 2018

I. Swett
Google
M. Montpetit
Triangle Video
V. Roca
INRIA

March 3, 2018

Coding for QUIC
draft-swett-nwcrg-coding-for-quic-00

Abstract

This document introduces means of integrating loss recovery coding in the proposed QUIC transport protocol. While no specific code is specified, the document defines how to integrate recent coding research to recover packets lost in QUIC sessions. This research targets the codes themselves as well as how to use them in real world protocols. Loss recover should improve the QUIC performance in sessions impacted by loss.

Table of Contents

1. Introduction

QUIC is a new transport that wants to improve network performance by enabling out of order delivery, partial reliability, and methods of recovery besides retransmission while also improving security. This document specifies a design to enable error correcting codes to be used to recover lost data in QUIC. Error correcting codes have the ability to recover packet losses in less than 1 round trip at the cost of more total data transmission and decoding delay. The design does not specify a code but allows to negotiate it hence assumes that more than one code could be offered concurrently as well as leaving open the possibility of new codes in the future. Without loss of generality in the document we consider that the encoding operations compute a linear combination of QUIC packets. Terms and definitions that apply to coding are available in RFC xxxx [nc-taxonomy]

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Design Considerations

2.1. Framing

A new QUIC frame type is defined within the current framework [quic-basics] to add the error correction feature. This new frame type contains inputs to the negotiated loss recovery algorithm as specified by the specific algorithm and coded information. THis frame payload begins with arguments that point to the negotiated function that generates the coding coefficients to be applied to the remaining payload. It also indicates the identity of the first packet in the coding window and its size as well as other necessary input. There are currently a number of options for identifying the coded packets. Firstly we can assign a new coded sequence number. If all packets are encoded in a session, then we can specify which packet is the first and with the window size it will be easy to recover the packets. If some packets are encoded and some are not then there could be gaps in the numbering that is not due to loss and we may need to specify which packets are inside the window using a form of run-length-encoding. We can specify a fixed-length field in order to identify the packets. Secondly, we could use the QUIC packet numbering which would reduce the overhead. This has the same functionality as using a sequence number except that in that case the gaps in numbering could be due to path migration and not losses Note: a reference will be provided in a later version.

2.2. Coding Symbol

One of the the design consideration is the definition of the code symbol. In order to minimize the impact on the QUIC design, the QUIC loss recovery (QLR) will be applied inside the QUIC encrypted packet payloads. Hence raw packets are used as symbols, the units of recovery are what the coding coefficients are applied to. Any packet payloads smaller than the coded payload will be implicitly padded with zeros as to prevent the detection of coding on any path. To allow for the coded packets to have more encrypted payload than other packets, any QUIC PADDING frames(type 0x00) [quic-basics] will be removed from the payload before applying the algorithm. This new frame type contains inputs to the negotiated loss recovery algorithm as specified by the specific algorithm and coded information.We want to keep the coding implementation simple, provide for code negotiation and stay independent of any encryption mechanism. It is thus proposed to apply loss recovery code before the encryption, hence to clear data. This allows the encryption operation to be unencumbered by coding. Hence raw packets will be used as coding symbols. The downside of this approach is that it does not enable non-participating middleboxes to add or remove encoding from packets but this is considered insignificant compared to the complexities of

interacting with security mechanisms. In addition, it is assumed that the the entire packet content will constitute a single source symbol. This choice is motivated by the desire to simplify the implementation. Coding should be applied to all QUIC packets except the 0RTT payloads. 0RTT payloads are sent prior to negotiation, and the QUIC negotiation mechanism does not allow sending extension frames prior to handshake completion.

2.3. Negotiation

There are already multiple candidates for the QUIC FEC and we assume that others may become available in the future as research continues.In order to stay as generic as possible and enable QUIC network operators to select their coding technology, it is assumed that a coding negotiation will be implemented to select one or more codes to be used over a QUIC session.This will be implemented using the one step negotiation of the new QUIC negotiation mechanism [quic-basics]. Each available coding algorithm should use the standard FEC frame [RFC3452] but reserve a different codepoint. We want to ensure that coding negotiation occurs during the QUIC handshake and can be used in all short header QUIC packets. Finally, to make the code selection as generic as possible, each algorithm should specifies a sequence of coding coefficients or a function to generate them, window sizes as necessary as well as meta information to ensure the conformant performance of the coding and decoding operations

3. References

3.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

3.2. Informative References

[nc-taxonomy] Roca et al., V., "draft-irtf-nwcrg-network-coding-taxonomy-07", 2018.

[quic-basics] Iyengar, J. and M. Thomson, "draft-ietf-quic-transport-09", 2018.

[RFC3452] Luby et al., M., "RFC 3452: Forward Error Correction (FEC) Building Block", 2002.

[RFC5053] Luby et al., M., "RFC 5053: Raptor Forward Error Correction Scheme for Object Delivery", 2007.

[RFC5510] Lacan et al., J., "RFC 5510: Reed-Solomon Forward Error Correction (FEC) Schemes", 2009.

[RLNC] Ho et al., T., "A Random Linear Network Coding Approach to Multicast", 2006.

[Tetrys] Detchart, J., Lochin, E., Lacan, J., and V. Roca, "draft-detchart-nwcrg-tetrys-03", 2016.

Appendix A. Appendix: Reference Algorithms

This ID does not mandate nor depends on any coding scheme. However, in order to have an initial implementation with good performance and not encumbered by intellectual property and proprietary implementations, it is suggested to use the Raptor (RFC 5053) as a reference algorithm. However, since the Raptor code perform badly with small blocks, depending on the application, another alternative is to use Reed-Solomon (RFC 5510) codes. It is assumed that other candidates that are free of IPR may become candidates in the future.

Appendix B. Appendix: Participating Middleboxes

The coding approach described in this document does allow on path elements that have the ephemeral keys to decrypt packets and add or remove FEC packets.

Appendix C. Appendix: APIS

It is planned that the QUIC coding mechanism will conform to any common API defined in the research group.

Authors' Addresses

Ian Swett
Google
Cambridge, MA
US

Email: ianswett@google.com

Marie-Jose Montpetit
Triangle Video
Boston, MA
US

Email: marie@mjmontpetit.com

Vincent Roca
INRIA
Grenoble, France
US

Email: vincent.roca@inria.fr